Implementation of Measurement Module For Seamless Vertical Handover

Selim İçkin

Thesis submitted for completion of Master of Science towards Electrical Engineering

Blekinge Institute of Technology
August 2010
Acknowledgements

I would like to express my gratitude to Docent Dr.-Ing. Markus Fiedler for his guidance supervision and patience throughout the evolution of my thesis. He provided a great knowledgable and experimental support that I needed to complete the thesis.

I would like to thank Alexandru Popescu and Karel De Vogeeler, my PhD student advisors, for their time and effort for providing me helpful and constructive feedbacks on my thesis work.

I would like to thank my family for their support during the completion of my masters degree over the past years and on my master thesis.
Table of Contents

Abstract ........................................................................................................................................................................ 6

1. Introduction .......................................................................................................................................................... 7

2. Technical Background and Motivation ........................................................................................................ 11
   2.1 Concept of Measurement .......................................................................................................................... 11
   2.2 Measurement Methods ............................................................................................................................. 12
   2.3 Traffic Measurement Tools ..................................................................................................................... 13
   2.4 Kernel Space Monitoring Tools ............................................................................................................. 15
   2.5 Why Kernel Space Implementation ....................................................................................................... 16
   2.6 Time .......................................................................................................................................................... 16
   2.7 Measurement Metrics .............................................................................................................................. 17
   2.8 ReorderingMetrics .................................................................................................................................. 19
   2.9 Comparison of the Methods .................................................................................................................... 23
   2.10 Packet Traffic Scenarios ....................................................................................................................... 23
   2.11 Importance of Including Departing Parameters .................................................................................. 25
   2.12 Inter Process Communication Mechanisms .......................................................................................... 26
   2.13 GRE Tunneling ....................................................................................................................................... 28

3. Problem Definition and Goals ..................................................................................................................... 29

4. Research Methodology .................................................................................................................................... 31
   4.1 Research Questions .................................................................................................................................. 31
   4.2 Applied Research Process ....................................................................................................................... 32
      4.2.1 Information Collection ...................................................................................................................... 32
      4.2.2 Information Analysis ......................................................................................................................... 32
      4.2.3 Implementation .................................................................................................................................. 33
      4.2.4 Experiments ....................................................................................................................................... 33

5. Implementation .................................................................................................................................................. 34
   5.1 Payload Dependent Approach .................................................................................................................. 34
      5.1.1 Development Environment ................................................................................................................ 34
      5.1.2 Test Module ....................................................................................................................................... 34
      5.1.3 Measurement Module ......................................................................................................................... 37
      5.1.4 Use of UDP Sockets .......................................................................................................................... 38
5.1.5 Receiving Data in Kernel Space.................................................................................................................. 38

5.2 Payload Independent Approach............................................................................................................................ 42
  5.2.1 Calculation of Delay Jitter.............................................................................................................................. 46
  5.2.2 Calculation of packet loss and reordering ........................................................................................................ 49
  5.2.3 Another Packet Loss Calculation.................................................................................................................... 52
  5.2.4 Reordering Detection and Calculation of DJ for Reordered Case..................................................................... 53

6. Results........................................................................................................................................................................... 55
  6.1 Payload Dependent Approach................................................................................................................................ 55
    6.1.1 Procedure of the Experiment........................................................................................................................... 55
    6.1.2 Reordering and Packet Loss Detection........................................................................................................... 56
    6.1.3 Reordering Detection................................................................................................................................. 56
    6.1.4 Implementation of the Jitter Calculation.......................................................................................................... 57
  6.2 Payload Independent Approach............................................................................................................................... 70
    6.2.1 Procedure of the Experiment........................................................................................................................... 70
    6.2.2 Packet Interval Approach (30 packets/Interval)........................................................................................... 71
    6.2.3 Time Interval Approach (1 second/Interval)................................................................................................. 76
    6.2.4 TCPDUMP versus Measurement Module.................................................................................................. 83
    6.2.5 Reordering Scenario Results......................................................................................................................... 85
    6.2.6 Method Validation Test..................................................................................................................................... 86
    6.2.7 Reorder Check.................................................................................................................................................. 87
    6.2.8 Packet Loss Results......................................................................................................................................... 89

7. Conclusion....................................................................................................................................................................... 90
  7.1 Experienced Problems............................................................................................................................................... 91
  7.2 Future Work.............................................................................................................................................................. 92

8. Appendices ...................................................................................................................................................................... 94
  8.1 Appendix A.
    8.1.1 Validation of the Delay Jitter Measurements (Payload Independent Approach)............................................. 94
    8.1.2 GRE Setup....................................................................................................................................................... 96
    8.1.3 Makefile......................................................................................................................................................... 96
    8.1.4 Used Linux Kernel Commands.................................................................................................................. 97
    8.1.5 Used Netem Commands............................................................................................................................... 97
Abstract

Research on heterogeneous seamless handover has become popular, since the wireless networking systems were introduced with mobility. The typical current vertical handover mechanism consists of an architecture built at Layer 3 and implemented to serve for different technologies [1]. Even though it has advantages in terms of simplicity, there still exist drawbacks such as the need of adaptation of the network architecture to different network technologies and systems. Providing transparency to the heterogeneous seamless handover can be provided by conducting the handover process at a higher layer. By that way, efficient handover decisions for vertical handover are made with more number of constraints that will lead to high performance, accessibility and low cost. Making this possible is by providing Quality of Experience (QoE) and obtaining current information of the throughput by measurements in the network. Analyzing and interpreting the statistics collected through measurements are vital in terms of decision making and to decide when to perform vertical handover. This thesis consists of the implementation of measurement module in two different approaches (Payload Dependent Approach and Payload Independent Approach) that will provide these statistics to the storage module in PERIMETER project.
Chapter 1

Introduction

High demand for multimedia rich applications in contemporary internet services brings the necessity of achieving good performance by using advanced mechanisms [12]. The development on infrastructure of wireless technology has brought improved performance, reduced cost and comprehensive accessibility to todays mobile networks. As this engineering and research process goes on, the architecture and technology behind the wireless technology are getting more complex. Both the fixed land line networks and the mobile networks need to care about user requirements such as Always Best Connected (ABC). However, additional work for mobile networks is needed to choose the best connection among other possible available connections on air. Therefore, the engineering process on the infrastructure should be done in a way that the end user should not experience any hazardous connection during an interaction with the mobile device. This leads wireless communication researchers to think more seriously about Quality of Experience together with the QoS in measurements for vertical handover.

As the number of subscribers that are using 3G applications on their mobile phones increase, there exists an exponential rise on the importance of keeping the high performance of the current wireless network connection stable for any kind of wireless technology. It's common not to expect a call drop during an important interactive video conference meeting with the boss while going from home to the office on a train early in the morning. Assume a Swedish passenger travelling in a high speed train and wants to watch the live final Worldcup between Sweden and Spain. He does not normally desire to get periodic connection problems and does not want to deal with handovers manually when he gets out of the train in the train station where an other alternative wireless network is detected. He would like ABC service to be handled without any notice during an interaction with the mobile. These connection problems occur most of the time due to the lack of bandwidth resources. The procedure of vertical handover plays a keyrole at this point. It's so critical for a wireless network system to analyze all the available resources on the medium independent from the time and connect to the network and service that provides the best performance without notifying the user. Once the mechanism is developed for the vertical handover scencarious on next generation mobile network systems, greater user satisfaction will be reached.

One way of looking onto this problem is by taking measurements and controlling the connectivity as desired. Measurement is considered to be one of the building blocks of the mechanism. It is to
monitor and probe the packets through the current network accurately and store it for the decision-making mechanism to be evaluated and perform vertical handover whenever needed. The main focus of the thesis will be the implementation of a measurement module that measures the delay jitter through the network by analyzing the packets with the emphasis on detection of potential reordering and packet loss.

The thesis interest is to implement a measurement module in order to produce quantitative descriptions of performance as the parameters; delay jitter and packet loss ratio to help the decision making mechanism in the PERIMETER project to decide when to perform vertical handover in between wireless networks without the need of adapting the protocol stack to the dedicated technologies. The measurement module is aimed to be the part of the QoE Delivery System of the PERIMETER project with the scope of providing user centric mobility in heterogeneous environments. The module has to perform end-to-end traffic measurements by probing the network on a routine basis or on demand. The collected statistics are analyzed and then stored in a database which will later be used for an event triggered mechanism called the QoE Event Manager, enabling vertical handover according to the QoE constraints. As to give an example, sudden increase in packet loss or considerably much increase in delay jitter should indicate that a given link is exhibiting worse performance as compared to another available link. In that case, the triggering of a vertical handover would be needed for the network to switch to another technology and link to provide favorable connection.

**Thesis Structure**

The thesis is organized as follows.

**Chapter 2  Technical Background and Motivation**

The topic of this chapter starts with the concept of measurement module, and provides an explanation of the already developed algorithms and methods regarding delay jitter and reordering detection. This includes previous researches together with the algorithms and calculations that are developed so far. The discussion continues to decide which measurement metrics to take into consideration to obtain healthy results. The requirements and ways to develop a measurement module in kernel space. Some theoretical information regarding the interprocess communication is given briefly. In addition to that, a brief theoretical information on GRE tunneling is also given.
Chapter 3  Problem Definition and Goals

The existing problem regarding the measurement through seamless communications and the aim of this thesis is depicted on this chapter.

Chapter 4  Research Methodology

This chapter addresses the methodology chosen to answer the research questions and the main goal of this research thesis. The methodology that is used in the approach in order to solve the stated research questions, and how the analysis were done will be discussed in this chapter.

Chapter 5  Implementation

In this chapter two forms of measurement module will be discussed. For the first part, the measurement module which measures the delay jitter and packet loss with respect to the values appended to the data field of the UDP header will be discussed (Payload Dependent Approach). And later the departure parameters are taken out from the data field and are put into the GRE header as extra header information independent from the data sent through the payload (Payload Independent Approach).

Chapter 6  Results

As refer to Chapter 5, two forms of analysis will be made on this chapter. Numerous methods (store buffer, ring buffer) are first tested without tunneling and with the Payload Dependent Approach. Later in the second part of this chapter, the results from the Payload Independent Approach are presented where these results are obtained from within the tunnel and completely in kernel space. The results involves the delay jitter and the packet loss as well as a few metrics regarding reordering. The results are presented on a packet and time interval basis.

Chapter 7  Conclusion

Chapter 7 concludes the thesis with a summary of the results of measurements that are obtained from both of the measurement modules (Payload Dependent Approach, Payload Independent Approach). It highlights the advantages of measurement module implementation in kernel level with tunneling feature.

Appendices

Additional information regarding technical terms and additional results are presented.
References

The references of relevance for this thesis are given in this final chapter.
Chapter 2

Technical Background and Motivation

The popularity of the research on heterogeneous seamless handover augmented the interest of computer network measurements because of the growth, complexity and diversity of the network based services [7]. In order to provide high Quality of Experience, the current information of the throughput in the network should be traced and well analyzed through observations. The observation results for a specific metric should be compared with reference to the stable results that are previously stabilized and regulated. That way, an observation is changed into a measurement. The accuracy of the measurements has great effect on the quality of the heterogeneous vertical handover.

The results of measurements should help someone to feel more comfortable while taking critical decisions [13]. Actual measurements help to produce and to interpret quantitative descriptions of the attributes to be worthwhile. Sometimes, it might get hard to interpret measurements due to possible undesired distortions in the environment. In daily life for instance, the erroneous performance measurements of the same products on different brands can lead to cost effective decisions. Measurement results should help in case of situations like deciding which specific product to buy or which telephone service to order, etc. Therefore the measurements should be accompanied with high accuracy.

*Delay jitter, packet loss* and *reordering* are the metrics that are of interest during the measurements taken in this thesis work. The important issue in the decision of reordering for the current network is to analyze the packet transmission rate, packet size, type of the transport protocol used, capacity of receive buffers and application purpose. For example, four packets that arrive in a sequence different to the expected sequence can be more or less neglected in one application, while it can cause a huge problem in another one. This applies for the packet loss as well, where the amount of lost packets might be neglected in one application while on the other hand should be considered to be as an important criteria that effects the QoE in all kind of communication especially in real-time applications.

2.1 Concept of Measurement

“*Measurement is a description of some attribute of an entity that can be expressed as a number or quantity*” [13]. Measurement affects the user perception by providing quantitative descriptions of attributes for services such like in networking [13]. “*Measuring the quality of service is used to*
evaluate the telecommunication services in ways that are useful to decision makers.” [13].
Thus, the measurement has to be done to lead to better evaluation and decision making for better quality of service.

2.2 Measurement Methods
Measurements are performed using various approaches, even though both of them rely on the same parameters and data during the process. The common approaches are active measurement and passive measurement. Throughout the thesis, I refer to active measurement as payload dependent and passive measurement as payload independent.

Active Measurement:
This is done by the injection of traffic into the network, observing and comparing the parameters at the source and destination in the network. In case of one link in-between source and destination without any traffic, it's expected from an active measurement tool to present more accurate results as compared to the connection of source and destination with multiple links. Practically, it is common that there are more than one alternative links due to the issues like load balancing and other actions as in Figure 1.

\[\text{Figure 1: Justification of a case when there is more than one route from source to a destination}\]

In addition to that, the disadvantage of active measurement is that it increases the load on the network and may cause wrong estimations of the measured parameters due to the interaction between the measurements and the real traffic. Examples of network-layer active measurement tools are ping, traceroute, etc [7]. An active measurement module is implemented in this thesis, however by using ping, it's not possible to add timestamps and modify the sequence numbers of the destined icmp packets. Thus, another user space module is implemented to inject IP packets with the packet sequence number and departure time attached to the data field per packet as illustrated in Figure 2.

Implementation Of Measurement Module For Seamless Vertical Handover
Once the packets are received on the destination side, the attached departing packet sequence number and the timestamp will be parsed and analyzed as compared to the same received parameters.

**Passive Measurement:**

This is another method of measurement that measures the parameters from the already running link without the need of extra fake traffic. This prevents from adding an extra load to the network and is very much preferred in large-scale network deployments. Along with the security and privacy drawbacks of this method, e.g., being able to see other's data, one other disadvantage of the passive measurement is that, involuntarily, it causes huge amount of data to be collected in the analysis. “A moderately loaded 100 Mbps full-duplex link generates 12 MB of data every second, that is 0.98 TB every day” [7]. In the thesis, a passive measurement approach is also implemented. This involves the payload-independent measurement module where the measurement parameters are appended to the GRE header of the IP packet rather than to the data field, in contrast to the implementation of the payload dependent measurement.

### 2.3 Traffic Measurement Tools

A number of traffic measurement tools have been developed for monitoring and probing the state of the current network and link. Some of them are implemented in user space while others are implemented in kernel space [8]. The metrics that are used in the tools are determined with respect to the application that is being used. Most of the time the tools that are implemented focused on measurements when the packets arrive in a sequential order that makes the calculation of delay jitters easier. One of the most widely used user-space tool is Wireshark [29] that analysis and prints out comprehensive results about the IP packets that arrive to any given interface. On the other hand, the Portable Multimedia Monitor (p3m) tool is an example of a kernel space measurement tool to analyze the throughput on the current network [8]. User space and kernel space network monitoring
tools have both advantages and drawbacks over each other. In fact, it is preferred to use both of the user-space and kernel-space as a kind of hybrid approach.

2.3.1 Advantages of User Space Measurement Modules

- **Fast Development of New Functionalities**
  User-space measurement modules are mostly preferred because it is faster to develop new functionalities.

- **Easy Debugging**
  It is considered to be easy to debug user-space tools from developers point of view. There is no limitation on the stack size, so its size grows with the usage. In other words, all the memory in the system can be used unlike in the kernel space where only limited amount of memory is available.

- **Fast Activation**
  In case of a software crash, it takes less time to bring up the tool as compared to the implemented kernel space tools.

2.3.2 Advantages of Kernel Space Measurement Modules

- **Fast Packet Processing**
  This is very much preferred in VoIP traffic, real-time video and in file transfers.

- **Minimal Jitter Observed**
  Real time characteristics of the network can be observed in an unbiased way as a result of fast and reliable packet processing.

- **Ease of Portability to Real Time Operating Systems**
  The kernel space modules can be loaded easily to the Kernel with a few lines of configuration commands to the Linux shell.

In Figure 3, the OSI reference model, execution environment and Internet model are illustrated. The application that functions at higher layers like (Layer 5-Layer 7), is considered to be an execution on userspace. Similarly, if the application that functions on lower layers like (Layer 1-Layer 3), it's favorable to be a kernel space application [7].
2.4 Kernel Space Monitoring Tools

To sum up the advantages of the kernel space as compared to user space development; When a system-call is made by a userspace to the kernel space, there is an unavoidable overhead in the user/kernel space communication, and this transition needs additional time to be performed in kernel space. Also, each time a packet is received or sent through the applications, the data is copied from user space to kernel space and vice versa [10]. Therefore, working on kernel space will cancel out all the drawbacks with less CPU load and will provide us a working environment with short response times.

There are numerous network monitoring tools in the market [26]. For monitoring QoS, there are few tool architecture that have been worked on.

- **HELIX**
- **MMDUMP**

  The above two tools are developed in user space and are reduced in efficiency [8].

- **P3M**

  Is a tool implemented in kernel space and server independent to make it more efficient in terms of extensibility adaptability. It consists of three kernel space modules: Video Packet Filter (VPF), MMPP and SKFIFO and three user space modules: plisten,
webserver, datastorage.

For researchers, monitoring of QoS should not be server-based due to its lack of flexibility [8]. Because, it basically supports one or several fixed protocols. In dynamic environments where a shift among many protocols are necessary, that is not widely preferred.

Since the thesis aims to implement in kernel level, the main focus and research will be on the solutions implemented in kernel space. However, not many kernel based tools can be found due to this complexity.

- **KUTE**
  Is a kernel based traffic engine [17]. It consists of a traffic generator and an analyzer that works completely in kernel space.

- **PKTGEN**
  Works completely in the Linux kernel and is used to send multiple UDP flows over Ethernet [28].

- **Click Project**
  There exists two modules in Click Project. It can send multiple UDP flows and measure packet arrival times [27].

### 2.5 Why Kernel Space Implementation?

The measurement module is developed in a way to provide maximum performance at the receiver that measures the delay jitter, detects reordering, calculates packet drop rate through the network. The only way to develop it is to use kernel space. Otherwise, the packets must pass through the complete network stack that decreases performance and is not desired especially for monitoring and probing applications. The receiver measurement modules implemented on kernel space interprets the packet inter arrival times more accurately.

This thesis's concern is to implement measurement module for tracking the real time traffic in the current network. Therefore, the thesis addresses the implementation mainly in kernel space.

### 2.6 Time

“Accuracy is the comparison done between the measured value and the true value” [7]. Reaching one hundred percent accuracy is desired while taking measurements. Even though, this is idealistic and hard to manage, the implementations should target one hundred percent accuracy.

*Clock synchronization* and *clock stability* has become the important subject of concern of passive and active network measurements [15]. The accuracy of the results based on the timestamps taken
from both ends can not be true unless they are synchronized. In server based tool, the error of synchronization is degraded by using the Network Time Protocol in Linux. `ntpd` and `ntpd` are useful commands that is used in synchronization of parties in server based systems. In server independent tools, it gets harder to synchronize.

Timestamp accuracy should also be taken into consideration. When a timestamp is needed to be taken when a packet arrives or departs, a function to get the timestamp of the moment is called. The time it takes to call the function and reach to the counter value affects the measurement results seriously [7]. In case of user space programming, this delay is expected to be large due to the system calls, thus resulting in wrong timestamp values. There comes again the importance of kernel space in measurement issues. Because, within kernel space, no system calls are needed.

There are other environmental reasons like temperature and age of the crystal that affects the stability of the oscillator and eventually the clock cycles and the timestamps. However, this is beyond the scope of this thesis.

### 2.7 Measurement Metrics

Delay, Loss, Jitter and Reordering are the four metrics that describes the Quality of Service in packet networking [4].

#### 2.7.1 Jitter (Packet Delay Variation)

The methods regarding the jitter calculation are as follows:

1) **Maximum Jitter = Maximum Latency − Minimum Latency**

Maximum delay variation is calculated as the substraction of the minimum latency from the maximum latency observed within packets per interval.

2) **$Jitter_n = |Arrival_n - Arrival_{n-1}| - |Arrival_{n-1} - Arrival_{n-2}|$**

Instantaneous delay jitter can be calculated among three consecutively received packets as described in the above equation, taking the difference between the time of the currently received packet and the previously received packet for two consecutively received packets.

3) **$Jitter_n = |Latency_n - Latency_{n-1}|$ where $Latency_n = |Arrival_n - Departure_n|$**

Therefore,

$$Jitter_n = |(Arrival_n - Arrival_{n-1}) - (Departure_n - Departure_{n-1})|$$

Instantenous latency is the time difference between the arrival time of the packet minus the departure time of the same one. Therefore the instanenous jitter is calculated by taking the
difference of the instantaneous latencies for two consecutively received packets.

4) $J_{\text{iter}}_n = |\text{Latency}_n - \text{AverageLatency}|$

Instantaneous delay jitter is can be calculated as the absolute value of the difference between the instantaneous latency per packet and the average latency that has been calculated per interval among the packets belong to that interval.

5) $J(i) = (15/16)J_{i-1} + (1/16)|\Delta D_i|$ where $J_i$ is the initial packet and $J_0$ is zero.

Another way of calculating the delay jitter for consecutive packets with exponential filter 1/16 is as follows. Refer to [6] for more information.

6) **Standard Deviation**

The difference of the timestamps taken from the departing and the reception side is subtracted. This difference is used as input to the standard deviation calculation. The output ($J$) is the delay jitter. This calculation is done and updated periodically for desired length of packet interval or time interval.

$R_n = $ Reception timestamp,

$S_n = $ Departure timestamp,

$J = $ Delay Jitter,

$N = $ Number of packets within the time/packet interval.

$$D_n = R_n - S_n$$

$$J = \sqrt{\frac{1}{N-1} \left[ \sum_{n=1}^{N} (D_n^2) - N\left(\overline{D}^2\right) \right]}$$

In the methods 1,3 and 4, two-point measurements are used where the measurement takes place in both sender and receiver. However, method 2 and method 3 seem similar. In method 2, the jitter measurement has been done only at the destination, where only the arrival time of a packet is taken into consideration, hence the method is called one point measurement.

Throughout the implementation, method 3 is implemented together with the standard deviation formula stated above. The results of the measurements and calculations are presented in Chapter 6 and Chapter 8.
Up to now, traffic generators and analyzers mostly supported method 2 where the measurement of inter-packet latency takes place in the receiver side. Transmitting packet latency, the time between departure time and arrival time, gives more precise information to the analyzers about the condition of the network, and is a must for one-way delay measurements. In case when the sender does not generate packets at constant rate, no transmitting packet latency difference (delay jitter during transfer) can be observed and it may seem like no negative effects are experienced on the current network (method 2). However the transmitting packet latency is very important to get an idea about the conditions along the network path. Throughout the implementation, the measurement regarding departure and receiving time is implemented in addition to the measurement regarding the receiving timestamps of the packets.

2.7.2 Packet Reordering

Packet reordering is the behaviour, with different kind of metrics attached and needs to be handled seriously in calculations of delay jitter and network analysis. Inherent local parallelism is said to be the main cause of reordering [5]. It generally occurs in the following scenarios:

- In case of load balancing and route instability, where there are more than one path from source to destination and the link metrics are not identical. This may cause the packets arrive out of order.
- Multiple processors at the switch ports may reorder the packets.
- Layer 2 handles retransmission for error correction. In this case, reordering may occur. However, this is not likely the case for specific networks, e.g., UMTS keeps the packets in order.
- The order of the packets in a buffer may not be handled correctly for any kind of technical reason.
- In case of non-identical buffers and service rates at different switches, routers or other communication end-user devices, reordering occurs, e.g., softphones.
- Asynchronous work done on processors and queues of the process in the devices. The varying packet sizes, QoS routines and firewall configurations on the network causes the packets with longer processing time to be queued on the queue more than the ones that have less processing time. (This also causes different packets to arrive at the destination not in the sequence as they were sent by the source.)
- Time complexity is a notion showing the duration of the execution of a method depending on the parameters on the input. Reordering measurements involve buffering and recording the
parameters like sequence numbers, delays and delay jitters of the arrived packets. Therefore the time complexity is directly affected by high amount of reordered packets.

The above mentioned reasons have been taken into consideration to answer the question why comprehensive reordering decision algorithms should be taken into account in analyzing the throughput of the networks like calculation of delay jitter and packet loss on networks. All the mechanisms with network complexity like QoS featured networks, ad-hoc routing and heterogeneous handover that brings local parallelism are considered to be the causes of reordering. Varying algorithms were developed to determine whether the reordering has occurred or not, as well as the degree of reordering, to give a wide ranging view on reordering to analyze the current situation of network and make precise decisions whether to decide on the necessity for the vertical handover: change the technology or the network.

Both TCP and UDP protocols are deeply affected by reordering. Late TCP packets are treated as lost, causing unnecessary retransmission over the network.

**Destination Packet Sequence Numbers**

```
0 1 2 4 5 6 7 8 9 3 10 11 12 13
```

In the above illustration packet number 3 is delayed by seven packets and arrives after the 9th packet. The TCP will consider the 3rd packet as lost and intend to retransmit depending on the window size.

In addition to that, TCP changes the window size according to the sequence of packets. In case of reordering, the TCP algorithm can not sense it precisely and acts as there was a packet loss. Therefore, it decreases the window size unnecessarily.

In UDP based applications like VoIP, the packet may arrive to destination after it has treated as lost. This result in a decrease in the perceived quality of voice. The reason is the late packet is typically 'jumped-over', i.e., thrown at the receiver.

**Source Packet Sequence Numbers**

```
0 1 2 3 4 5 6 7 8 9
```

**Destination Packet Sequence Numbers**

```
0 1 2 3 5 4 6 7 8 9
```

Reordering occurs in most of the cases in high-rate applications like video conferencing, VoIP, etc. According to the experiments done previously, it has been observed that reordering exceeds the order of three packets over European links [4]. It is as important to detect precisely when the packets are reordered as to avoid that the packets to become reordered, and make the delay jitter calculations to obtain statistics on the network.
2.8 Reordering Metrics

The thesis proceeds with some reordering metrics as follows.

Reorder density (RD)

In some cases, the reordering is analyzed with respect to the number of reordered packets in the frame called reorder density. Reorder density is a metric that counts the number of packets that have not arrived in the expected sequence. This metric provides satisfactory information about the degree of reordering [11].

\[ d_m = \text{distance of the } m\text{th received packet to the expected sequence number.} \]

If the receive_index for packet \( m \) has arrived at \( m + d_m \) with \( d_m \) is not equal to zero that means a reorder event has occurred.

This event is shown by \( R(m, d_m) \) [11]. The packet reordering in a sequence of packets is shown by the union of reordered events and denoted by \( R \) as given by the following equation.

\[ R = \text{Product} \{ r(m, d_m) \} \text{ where } d_m \text{ is not equal to zero for all } m. \]

If \( d_m < 0 \), packet \( m \) has arrived earlier than expected, and if \( d_m > 0 \), packet \( m \) is arrived later than expected. \( d_m = 0 \) denotes that the packet has arrived as expected.

Reorder Buffer-Occupancy Density (RBD)

If a packet arrives before the expected time, it is stored in the buffer and the buffer-occupancy metric becomes one. Buffer-occupancy is tracked in each arrival of a packet. The buffer-occupancy number remains as it is until the next packets complement the sequence gap between the early packet and one previous packet that arrived before the early packet. When an early packet is experienced, it is stored in the buffer, and the expected packet remains the same until it is received. The buffer occupancy becomes one. Once the expected packet arrives, the buffer occupancy is decreased back to zero considering the stored packet sequence number in the buffer to be received. Thus, the expected sequence number should be set by skipping the previously early received packet.

The density of occurrence of zeros and ones builds up the reorder buffer-occupancy density. The definition gets clear in Table 1 on page 22.

Reorder Extent

This metric is lateness-based. Reordering decision is made when a packet is detected to be late. If
the packets arrived in the expected time, it is assumed that the extent parameter is taken to be undefined. Extent metric is the difference between the expected sequence number and the actual packet arrival sequence number. Reordering is calculated as the density of each extent metric. Suppose $s[i]$ denotes the arrived packet in the destination where $i$ represents the expected sequence number. There exists a set of indices $j$ ($1 \leq j < i$) such that $s[j] > s[i]$. The reordering extent $e[i]$, of packet $s[i]$ is defined to be $e[i] = i - j$ for the smallest value of $j$ such that $s[j] > s[i]$ [16].

The example is given in the first row second column in Table 1. When packet 5 arrives, there $s[i] = 5$ where $i = 6$. The algorithm checks for $s[j] > s[i]$ and it is found to be that $s[j] = 7$ where $j = 5$. The extent 'e' holds the difference $i - j$ as variable 1. Consider packet 6 is received, $s[i] = 6$ where $i = 7$. The algorithm checks for $s[j] > s[i]$ where it finds $s[j] = 7$ at $j = 5$. There, the difference $i - j$ is calculated and stored in extent.

**n-Reordering**

n-Reordering defines the extent as the maximum distance between packets, from reordered packet to the earliest packet received that has a larger sequence number. However, if the packets are late in sets then only the first packet is considered to be reordered. The degree of n-reordering of the sample is 'm/N', where $m$ is the number of n-reordered packets in the sample and $N'$ is the size of the arrival sequence after discarding duplicate packets [16]. As can be seen from Table 1, the degree of reordering in the sequence of {1 2 3 4 7 5 6 8} turns out to be 1/8 since the next set of numbers after reordered packet 7 is consecutive.

<table>
<thead>
<tr>
<th>Arrival Sequence</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>5</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Index</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Displacement</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Reordering Density</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R = {(5, 6), (6, 1), (7, -2)}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RD[0] = 1/8, RD[0] = 5/8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RD[1] = 2/8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reorder Extent</td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Number of Packets with $e=1$ is 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Packets with $e=2$ is 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arrival Sequence</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Expected Sequence</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Buffer of Occupancy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ring Buffer Occupancy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RBD[0] = 6/8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RBD[1] = 1/8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-Reordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Degree of one-reordering is 1/8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-Reordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

*Table 1: Illustration of different reordering methods [16]*
2.9 Comparison of the Methods

In summary, RD measures the late-ness and earliness of the packets while RBD has a buffer and decides on reordering in a more tolerable way considering the resources for recovering from reordering. Reorder Extent and n-Reordering makes decisions only with respect to the lately arrived packets. RBD limits the algorithm to decide reordering after the late packets arrival. Thus, RBD, Reorder Extent and n-Reordering satisfies partial requirements on the other hand RD seems like to be more close to the solution where it satisfies this fundamental requirement for capturing reordering scenarios. Together with “capture reordering”, there are other essential requirements that must be satisfied by reordering metric like low sensitivity to packet loss and packet duplication, metric’s usefulness to evaluate behaviour and performance of a network, etc. The remaining requirements are desired attributes: simplicity, informativeness, evaluation complexity, robustness, and extensibility to cascaded networks. A comprehensive chart comparing the metrics mentioned above with respect to the requirements is shown in Table 2 that has been taken from [16].

<table>
<thead>
<tr>
<th>Requirement Metric</th>
<th>RD</th>
<th>RBD</th>
<th>Lateness Based Percentage Reordering</th>
<th>Reordering Extent</th>
<th>N-Reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capture Reordering</td>
<td>√</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>Low Sensitivity Loss and Duplication</td>
<td>√</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>Usefulness-TCP Flow Control</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
</tr>
<tr>
<td>Usefulness-Buffer/Resource Allocation</td>
<td>√</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Usefulness-Causes and Effects</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Simplicity / Informativeness</td>
<td>√</td>
<td>√</td>
<td>x</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>Robustness</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>√</td>
</tr>
<tr>
<td>Extention to Cascaded Networks</td>
<td>√</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

x: attribute is absent, √: attribute is present, h: attribute is partially present.

Table 2: Comprehension of the reordering metrics

2.10 Packet Traffic Scenarios

There are basically five types of cases that are experienced on packets from the delay jitter analysis point of view through the network. These are stated as the following: [6]
• **Constant Latency**
  ◦ The latency is approximately constant throughout the measurement interval.

  Departing Time (t)

  ![Figure 4: Constant Latency](image)

• **Alternating Latencies - Saw Tooth**
  ◦ The latency changes to a higher value and gets back to the original one periodically.

  Departing Time (t)

  ![Figure 5: Saw Tooth Model](image)

• **One Packet Having High Latency - Spike**
  ◦ There exists one spike of a high latency throughout a measurement interval.

  Departing Time (t)

  ![Figure 6: Spike Model](image)
- **Latency Changes Once to a Different Value - Step**
  - The latency changes to a different value than before and then stays constant.
  - Departing Time (t)
    - Departing Packets
    - Received Packets
    - Figure 7: Step Model
  - Reception Time (t)

- **Latency Gradually Increasing and Decreasing – Steps**
  - The latency rises to a certain value and shifts back to the previous value, a periodic step of latencies in other words.
  - Departing Time (t)
    - Departing Packets
    - Received Packets
    - Figure 8: Steps Model
  - Reception Time (t)

The measurement module supports two point measurement by putting a departed timestamp to the payload of the IP packet before it is sent. When the receiver receives the packet, the measurement module calculates the delay jitter to compare the latency of the currently received packet as compared to the previously received packet.

### 2.11 Importance of including parameters at the sender side

There can be three possibilities where we get constant delays by looking at only the arriving packets' timestamps. First possibility is the normal ideal scenario where there are no faults on the network between the sender and receiver. First possibility can be the case when packets can be considered to be clumping at the sender or spreading out at the receiver or vice versa as illustrated in Figures 9, 10.
and 11.

Figure 9: The packets are clumped at the source and spread at the destination.

Figure 10: The packets are spread at the source and clumped at the destination.

Figure 11: Ideal packet transfer scenario

Therefore, there is a strong need that we need to take the departure time of the packets into consideration. In the rest of the thesis, the calculations and considerations are made according to the variations between the departure and arrival times of a packet. This difference will be taken as a parameter in the delay jitter calculations.

2.12 Inter Process Communication Mechanisms

Together with the implementation and design of the measurement module in kernel space, the packet traffic generator class is also implemented. However, the implementation of this traffic generator has been done at the user space. The reason behind was trying to simulate a real-life scenario, in which we consider that all packets arrive to kernel space from user space. This traffic generator class creates the IP packets with desired parameters like sequence number and timestamps. Moreover, the measurements and calculations can be interpreted and analyzed on the kernel level with the help of the measurement module by comparing the parameters.

In order to create and send an IP packet to the kernel space, and making the kernel space module to
get the packet and parse it, a piece of software is required. In the following lines, some terms regarding inter process communication will be discussed.

Figure 12 shows all the communication possibilities between user space and the kernel space.

- Communication over the UDP sockets
  Are basically used in user space to kernel space communication applications

- UNIX Sockets:
  Used in user space to user space communication applications.

- PIPES
  This provides the communication over the named pipes. This is also used in user space communication

- GENERIC NETLINK

Figure 12: User Space and Kernel Space Communication

Implementation Of Measurement Module For Seamless Vertical Handover
Provides communication over the generic netlink sockets between user space and kernel space.

- **KERN**
  
  Is basically used in kernel space to kernel space communications by direct function call.

For the payload-dependent measurement module implementation, UDP Socket Communication is used. It will be discussed in more details in Chapter 5.1.4

### 2.13 GRE Tunneling

Tunneling provides the transport of packets from one protocol within another protocol [18]. Generic Routing Encapsulation (GRE) is a Cisco-proprietary stateless tunneling that provides multicast traffic and IPv6 traffic and widely preferred inside and outside Linux environment. Stateless means that it does not keep information of the status of the remote end point of the tunnel. In other words, it does not care whether the remote end point is up or down. It encapsulates most of the network layer protocols inside the IP tunnels and carries the passenger protocols. It creates a virtual point-to-point link to Cisco routers, as well as the Linux machines over an IP internetwork. GRE Encapsulation can be considered as a virtual layer on the OSI model that resides in the network layer, and makes the separation between IPv4 and IPv6 protocols [24]. GRE is a packaging protocol that can package any protocol's packets into generic data packages that are able to be carried by any kind of protocol.

When a tunnel is needed for various protocols, GRE tunnelling is preferred rather than an IP-IP tunneling. Even though, the GRE has issues of weak authentication, it can be supported by IP-Sec [25]. Another benefit of the GRE tunneling is the possibility of doing/implementing the measurements in the Linux GRE tunneling module (ip_gre.c) itself. This way, a passive measurement mechanism independent of the packet payload is possible. The applicability of this test is also done by embedding all the measurement code into the GRE module. The procedure concerning the configurations of GRE tunnel on two Linux boxes is found in Appendix A: Chapter 8.1.2.
Chapter 3

Problem Definition and Goals

The products and systems are expected to be developed with the predefined QoS values and specifications and then provided to the user. Various definitions have been made for QoS in the previous researches where one of them is: “The degree to which a system, component or process meets customer or user needs or expectations” [14]. According to Garvin in this reference, the concept is identified by five perspectives: transcendent, product-based, user-based, manufacturing-based, and value-based. These perspectives should both fit to users and developers in order to synchronize and negotiate while both sides are satisfied.

However, in dynamic test mediums and environments, there exists other sophisticated variables and constraints that are not taken into consideration rather than the QoS values and specifications during development of a system. That may corrupt the outputs of the developed systems and resulting the end user to complain about during the experience [3]. In that manner, there comes the importance of a subjective way of testing the quality via QoE where the end user has a chance to experience the faults and report as a feedback for the improvement of the product development. Therefore, the additional dynamic control mechanism on a live network that takes measurements on the network is strongly needed for QoE purposes. The measurement module is the part of the PERIMETER's QoE Delivery System that conducts network measurements. The monitoring and probing network interfaces is adequate within the kernel because there exists the root of the network connectivity in kernel. Eventually, the kernel tools are the ones performing actual measurements and this lets us obtain more accurate results for the analysis. So to speak, to monitor and probe the current state of the network as a snapshot, a measurement module in kernel space is being implemented. Two metrics are of specific interest and thus measured and investigated throughout the implementation:

- Delay Jitter:

  A sophisticated algorithm is used to determine the delay jitter. In an ideal environment where there is no packet loss, computing the delay jitter is easy. Though when network performance deteriorates and retransmissions occur, packets arrive out of order and sequences become mixed up. Therefore a comprehensive delay jitter assessment algorithm needs to be developed. In the measurement module, calculation of delay jitter has been implemented and tested with different methods.
• **Packet Drop Rate (packet loss ratio):**

The PDR can be measured with the help of sequence numbers of each packet. This information can be observed by looking at the payload or the header of the packet depending on whether the measurement is being done as an active or a passive one. A missing sequence number can have two reasons. Either the packet has gotten lost/dropped or it has been delayed and will arrive at a later time and out of order, the latter of which is called the reordering state. This involves the implementation of detection of reordered packets as well as the lost packets on their journey throughout the network.
Chapter 4

Research Methodology

There exist a few tools but still an implementation of a measurement module is needed to analyze and store the statistics of delay jitter and packet loss to be used in the PERIMETER project. This thesis aims at implementing and providing statistics for decision making mechanisms for vertical handover in seamless communication that will be used in PERIMETER project.

The thesis has been constructed mainly in two parts. In order to apply a qualitative research, in the first part, comprehensive research has been done which also consists of the literature overview to answer some of the research questions (Q2 to Q3 in Chapter 4.1). In the second part of the thesis, the quantitative study is conducted based on the qualitative study made on the first part. The aim of the second part is to motivate the already existing methods based on the first part as well as testing the newly developed algorithms with actual outcomes of the implementation to show the need of answering the remaining research questions (Q1, Q4 to Q5 in Chapter 4.1). The quantitative part consists of implementation and testing the methods that were researched and developed.

4.1 Research Questions:

The research questions declared below will be answered in the thesis.

1. What are the performance metrics associated with QoS measurements?
2. What are the existing techniques and tools that are used in QoS measurements?
   2.1 Compare the tools used so far in kernel space with the ones in user space.
   2.2 What are the advantages of kernel space network programming?
   2.3 Why are the network measurement tools preferred to be implemented in kernel space?
3. What are the methods and solutions used in the implementation of a QoS measurement module?
   3.1 Compare the pro and cons of such.
   3.2 What is the best applicable solution in QoS measurements?
   3.3 What are the cases that can be experienced on the latency of each packet during packet transfer and the methods to detect reordering packet loss and calculation of delay jitter?
4. How can a measurement module be implemented in kernel space to analyse the packets and
stores the data on the current network?

By finding solutions to the above stated research questions below, analysers and researches will get a better understanding of the previous researches and methods, will get a better picture with comparisons of such, and catch a better point of view to analyze the network traffic. In the measurement module, the most desired of the previous methods were implemented together with the newly developed ones to calculate delay jitter and make comparisons among them. In addition to those, it provides to observe reordering and packet loss scenarios in details with varying notifications and measurement results.

4.2 Applied Research Process

![Drawing 1: Illustration of the steps through this work.](image)

4.2.1 Information Collection

The research starts with the collection of material about the tools, methods and algorithms developed so far to calculate delay jitter, detecting reordering and tracking the lost packets through the current network. Understanding the basis of such is important to take steps in the implementation in the best way. The material consists of whitepapers, articles, specifications (RFC's), projects regarding the operation and methods to find out solutions in the aim of reaching the goal of the thesis. The collected information is presented in parts: the concept of measurement, the parameters need to obtain information through the network, the delay jitter calculations and algorithms, reordering detection algorithms, and the previously implemented tools.

4.2.2 Information Analysis

The analysis of the collected information and finding out the necessary points is vital. Together with the analysis done before starting the thesis, it is also important to analyse the implemented results at the end of the thesis. Analysis is known as one of the most important source of error in network
measurements [7]. Smoothing out the time differences of the delays among the packets by taking huge amount of data at an interval would be nothing but hiding or distributing the faulty information to the average to make it disappear in the overall. Therefore, in this implementation, it's also important to choose which approach, e.g., the length of the interval, etc., to choose in the analysis of the results.

4.2.3 Implementation

This step discusses how the measurement module is implemented based on the collected and the analysed information. This involves the parsing of necessary information from packet and analysing the needed parameters, and processing the parameters to obtain comparable results for different methods.

4.2.4 Experiments

The experiment part is actually the testing part of the measurement module in kernel space. Another testing module is developed in user space to verify and validate whether the measurement module is working as expected. The results obtained with the execution of the measurement module are deeply analysed and compared depending on different packet transfer scenarios. This involves the delay jitter values obtained by different methods, detecting reordered packets with different algorithms and verifying the accuracy of the detection of the dropped packets through the network. The active measurement module is tested with the user space sender module and the passive measurement module is tested by using the ping command together with shaping tools like Netem.
Chapter 5

Implementation

This chapter describes the implementation of the measurement module that will provide analysis on the current network to the decision mechanisms for vertical seamless handover between wireless and cellular networks with focus on always best connected. The analysis of the measurement module basically consists of the calculation of the measured timestamps of the departed and received packets as well as the packet sequence numbers. The data for delay jitter values will be later stored in a separated database module for future analysis for decision mechanism of the vertical handover systems.

5.1 Payload Dependent Approach

 Basically, there is one measurement module implemented in kernel space and one testing module implemented in user space that sends packets to kernel space for measurement module to receive and analyse as mentioned in previous chapters. The module is implemented while the departed packet parameters (timestamp and packet sequence number) are appended to the payload field of the UDP packet. (See Figure 2 in Chapter 2.3.)

The measurement module is compiled in kernel 2.6.29-generic.

5.1.1 Development Environment

- Ubuntu 8.04 kernel 2.6.29-generic
  - Linux is used to get benefit of its wide distribution, open source policy and its stable O/S. Linux supports the dynamic loading of kernel modules, so that the code running on kernel space can be tested easily.
- GNOME desktop environment.
- GNU C Library
- Gedit, the text editor

5.1.2 Test Module

The test module is a user space module that was written to send packets to the measurement module that resides in kernel space. The layer where the traffic generator is located in the network stack is important in terms of its influence on network stack implementation. The lower it's located, the less it's influenced. The timestamp is added to the payload buffer together with the sequence number of the packet. Then the packet is sent to the specific IP address and port via UDP socket.
communication. The following procedure is followed to send the packets to kernel space.

- Create the socket
- Binding the socket to well known address and port.
- Send the packet out form the network interface to the defined destination address, e.g., `Sendto()`.

In addition to the above steps, the packet sequence number and the timestamps are also included by the help of the following methods:

- `gettimeofday()` to get the timestamp at the moment when the packet is sent.
- `ultostr()` to change the unsigned long to a string/ascii in base 10.
- `sprintf()` to append different fields to the payload field

In the tester module, different buffers are allocated for the sequence number, the departure timestamp in microseconds and the departure timestamp in seconds. These allocated buffers are then appended to each other to obtain one unique buffer by using the `sprintf()` method. After the final buffer is created, the information is ready to be sent to the specified socket address. The following stated command is used to run the testing module and it takes two extra arguments. Below is the part of the code-piece showing how the packets are created with desired parameters attached to the payload field of UDP packet and sent to the kernel space measurement module.

```c
n=sprintf (buffer, "%i %i", h, c); // the variable c is appended to the buffer h
p=sprintf(buffer1, "%s %i ", buffer,time_mili); //the microseconds part of the departure timestamp info is appended to the buffer created in the above line.
r=sprintf(buffer2, "%s %i ", buffer1,time); // the second part of the departure timestamp is appended to the buffer1 created in the above line.

//we send the payload to the defined socket address.
//s is the socket ; buffer2 points to the buffer to be sent; 3rd parameter is the size of the buffer; the rest of the parameters are address and address size.
ret = sendto(s, buffer2, strlen(buffer2), 0, (struct sockaddr *)&addr, sizeof(addr));

//if sent is successful sendto() shall return the number of bytes sent. Otherwise, -1 shall be returned and errno set to indicate the error.
if (ret == -1) { perror("sendto"); exit(1); }
```

Many different combinations of packet transfer scenarios can be established by 'playing around' with the above code. In addition to that, below is the code piece showing how the initialization of the buffers are made in creating the final payload.
Once the class is compiled, it's ready to be executed with the following line. It takes two arguments as defined in the implementation.

`/tester.exe “ip_address_of_the_receiver” “port_number_of_the_receiver”`

The arguments needed for execution is defined as the following:

```c
addr.sin_family = AF_INET;
ret = inet_aton(argv[1], &addr.sin_addr); // first argument is the ip address of the remote box
if (ret == 0) { perror("inet_aton"); exit(1); }
addr.sin_port = htons(atoi(argv[2])); // second argument is the port number and is set to 5555
```

Some of the functions used in the above code are explained briefly in the below lines

- **`int inet_aton(const char *cp, struct in_addr *addr)`**
  
  This function converts the specified string in the first parameter to a network address, and stores the address in the second parameter. Each byte of data between the dots in the first parameter are interpreted to the four bytes of the internet address.

- **`htons()`**
  
  This function converts a 16-bit number `u_short` from host byte order to big-endian TCP/IP network byte order

- **`atoi()`**
  
  Converts ascii to integer

Inside the tester file, there exits three types of functions that lets the tester to send packets to the receiving side.

- The reordered packet sequence
- The ordered packet sequence

Implementation Of Measurement Module For Seamless Vertical Handover

36
• The packet loss scenario sequence.

Therefore, it gives flexibility to the tester which scenario to choose and test.

5.1.3 Measurement Module

The process of handling the incoming packets in kernel level is not as simple as in user space. In this subsection, it will be explained in detail.

This kernel module listens to the IP packets which were being sent from the test module on a specific port. Once the module detects the first packet, the process continues with parsing the IP packet by decapsulating to reach the information encapsulated in the payload. The information consists of the departed sequence number of the packet and the departed time of the packet in nanoseconds. Although the second showing the packet departing time is added to the payload, this parameter is not used at the moment. The reason is the measurements are done in microseconds range. It might be needed for highly congested real time network traffic. It's appended to create the backbone for future work implementations. The measurement module calculates delay jitter occurred in-between the arrived packets and also capable of revealing reordering and packet drop rate with various methods mentioned in Chapter 2.7.

The performance metrics associated with QoE measurements are,

• Latency is measured in microseconds.
• The Delay Jitter is measured in microseconds.
• Packet Drop Rate is calculated according to the measurements taken of the amount of lost packets through a time interval.

The implementation of the measurement for delay jitter is possible by using the departure time of the packets that are placed into the payload in the sender module and comparing the parsed metrics at the receiving side with the ones obtained in the receiving side of the tunnel. This way of measurement is called the payload dependent measurement. There is also another way where the parameters are appended to the GRE header in one end of the tunnel and parsed in the receiving end of the tunnel. This does not involve changing the payload of the sent packets. This way of measurement is called payload independent measurement. In payload independent measurement there is no need for a traffic generator. In this technique, the current network situation inside the tunnel will be traced and analyzed.
5.1.4 Use of UDP Sockets

It is expected that there will be multiple interfaces per application to the measurement module during the analysis of the packet on the current network. In order to handle these interfaces at the same time, measurement module is implemented with the callback mechanism. This provides the kernel module to receive packets asynchronously. The callback function can be extended further to more than one for handling more functions like control purposes. Currently, only one callback function is declared in the measurement module implementation for data reception. The measurement module registers the callback function for the socket that it listens for data messages. The callback function's duty is to insert the callback handler function into the work queue. The work queue handler function dequeues the messages from the socket receive queue list. There is the method called handle_pkt(skb) which takes the dequeued element in the head of the list, parses and makes desired analysis. The function handle_pkt will be widely discussed in the next subchapter.

5.1.5 Receiving Data in Kernel Space

When the module is loaded to kernel the first method to be called is

```c
static int __init server_init( void )
```

The server opens a socket, binds it to a port and waits for messages sent to it. Come to one that uses the server sent the sender information to respond to the client.

Initialization Functions

- **sock_create ()**
  It creates the socket, specifies that the symbols "SOCK _DGRAM" and "IPPROTO _UDP" is a UDP socket. The socket is created as the following:
  ```c
  sock_create(PF_INET, SOCK_DGRAM, IPPROTO_UDP, &udpsocket)
  ```

- **bind ()**
  In the structure "sockaddr_in" specified receive port binds the struct function 'bind () "to the socket.

```c
server.sin_port = htons( (unsigned short)SERVER_PORT);
//bind the port to the socket
servererror = udpsocket->ops->bind(udpsocket, (struct sockaddr *)
&server, sizeof(server ));
```

Implementation Of Measurement Module For Seamless Vertical Handover
Other Functions

- `sock_release()`

The sockets that were allocated and used to send data over internet must be released when not used anymore. This function should also be called in case of server error, and is the reason we utilize the below if check.

```c
if (servererror) {sock_release(udpsocket);
    return -EIO;}
```

OR

/* When the module is unloaded from the kernel, this method is called (ie: >rmmod measurement.ko) */
static void __exit server_exit( void )
{
    if (udpsocket)
        sock_release(udpsocket); //releases the sockets
    if (wq) { //clears the workqueue.
        flush_workqueue(wq);
        destroy_workqueue(wq);
    }
    printk("EXIT MODULE"); //prints out the message informing the unload.
}

Let's examine in more details the contents of the below method:

```c
static int __init server_init( void )
{
    // Implementation...
}
```

Afer the socket is created and bound as described in the above lines, another function `callback` is called as soon as a data is received on the socket.

- `Callback()` Method

The Linux kernel invokes this function when a data is received on specific socket. Struct sock is the argument of the callback function depending on the data. Callback function is responsible from dequeuing the messages from the socket receive message queue, process the data and free the memory occupied by the message.

Below is the callback function used in the implementation.

```c
static void cb_data(struct sock *sk, int bytes){
    wq_data.sk = sk;
    queue_work(wq, &wq_data.worker);
}
```

Implementation Of Measurement Module For Seamless Vertical Handover

39
How the callback function is associated with the socket is shown with the code snippet as follows:

```c
//The Linux kernel invokes this callback function when data is received on the
//specified socket
udpsocket->sk->sk_data_ready = cb_data;

/* create work queue */
INIT_WORK(&wq_data.worker, call_main);

wq = create_singlethread_workqueue("myworkqueue"); //creates the thread and puts
into queue.
if (!wq){return -ENOMEM;} return 0;
```

The task of processing some data is divided into two parts.

First part which prepares the data:

- Is executed in interrupt context where all other processes are masked and disabled. This part receives the data and passes them to the second part.
- To implement the second part, the Linux kernel introduces three mechanisms:
  - kernel timers
  - tasklets
  - work_queues

Kernel timers and tasklets are in interrupt context but they are not allowed to sleep. Work_queues are executed in process context and are able to sleep. Each work queue has multiple processes which runs the functions in the queue. The executed function may sleep, and the process context reschedules the function after a sleep.

Work queues are useful when definite fast function calls are needed. It allows kernel code to call a function as soon as possible or at a later time. There exists a default work queue that can be reached by any module in Linux kernel on the other hand there can be work queues specifically created on each module. It's not preferred to use the default work queue since many different functions are registered to it and is most of the time busy. When a function needs a long period of time for execution, it's suggested to use the newly created work queue on its own module.

There are functions called work queue handler functions that handle the functions executed from the work queue. There can only be one instance of a work queue handler function in the work queue at a given time. In case of using socket API together with work queue, this might be problematic. The number of packets that arrive into the socket is unknown until the work queue is executed. This
might cause the socket callback function to dequeue the packets from the socket receive work queue, even though this is not desired to happen. To avoid this, only the work queue handler function is triggered which is responsible to dequeue all the messages in the socket receive message queue. Normally, the work queue handler function has the following interface:

When the work queue handler function is executed, it needs the base address of the struct work queue wrapper. Then, the address of the whole wq_wrapper struct is returned by the following function.

```c
struct wq_wrapper * foo = container_of(data, struct wq_wrapper, worker);
```

where data is the pointer showing where the type 'struct wq_wrapper' belongs to and member is the name of the pointer in the struct.

The next step is to dequeue the packets on the queue list, analyze according to the measurement needs and then free the socket. This is done with the following lines.

```c
skb = skb_dequeue(&foo->sk->sk_receive_queue);
handle_pkt(skb); //the main function which parses the ip packet and calls other functions for calculations
/* the below functions are used to adjust memory boundaries, is explained in the thesis in more details. */
/* free the initial skb */
kfree_skb(skb);
```

UDP header remains attached in the data after the dequeuing process. Therefore the actual message, the payload, starts after 8 bytes, source port 2 bytes, destination port 2 bytes, length 2 bytes and checksum 2 bytes.

Even though payload dependent measurement module is not preferred to use due to the need of active traffic measurement tools, it is implemented for the purpose of testing the developed algorithms. It is flexible and easy to use for testing the algorithms. Appending the desired timestamp and the sequence number to the payload of the packet by the user space sender module, lets the tester to send the packets in the desired sequence number and with the desired delays to the receiving module. Detailed work on the implementation of the sender and receiver modules together with the configuration of the tunneling modules will be discussed in the next chapter.
thesis continues with the explanation of the measurement Module implementation with the payload independent approach in the next chapter.

5.2 Payload-Independent Approach

The aim is to implement a module that carries out passive measurements of the delay jitter to have an insight about the current status of the network independent of the packet contents sent through the network. Experimenting with a direct, i.e. without routers/switches in-between peers, connection between the sender and receiver Linux boxes, prevents the possible delay that might occur due to the routing/switching process.

This is done by embedding the measurement algorithms that are implemented in the Payload-Dependent measurement module to the already developed GRE tunneling modules. The GRE tunneling modules (ip_gre.ko) should be loaded into both the sender and the receiver, so that the GRE tunnel can be established in-between. The configurations made on both sides for tunneling are shown in Appendix A: Chapter 8.1.2. The sender and receiver Linux boxes are installed with kernel 2.6.29. The reason behind this is to make the measurement module to suit with the newly released Android 1.6 since Android 1.6 mobile phones have kernel version 2.6.29 running as the operation system. The measurement module is implemented on top of the features of the GRE tunneling module. It establishes the Generic Encapsulation Encoder tunneling between the sender and receiver side of the tunnel.

There are basically two methods that are of interest in this module:

- static int ipgre_tunnel_xmit()
- static int ipgre_rcv()

![One way delay jitter calculation](image)

*Figure 13: GRE Tunneling and a set of key methods*
The first method (ipgre_tunnel_xmit) above handles the packets by pushing the packets inside the GRE tunnel once the packets entered into the tunnel, see Figure 13. Here, the sequence numbers of the packets at the beginning of the tunnel are set. Furthermore, the timestamp is created and appended to the GRE header together with the sequence number as an extension to the GRE header format. On the other hand, the second method handles the receiving process of packets. At the end of the tunnel, this method of the module will be the one which will make the actual measurement and calculation of the delay jitter. Once the above defined metrics are appended and sent through the tunnel towards the receiver, these metrics are parsed and traced within this method.

The GRE module is a GRE over IP protocol decoder. In Linux, in order to establish the GRE tunneling, the ip_gre module is used. The packets sent through the tunnel can be detected and be observed according to the following parameters. As the packets are sent from the sender towards the receiver along the tunnel, the parameters for the TX and RX fields are observed to be updated as in Table 3.

<table>
<thead>
<tr>
<th>gre/ip</th>
<th>remote</th>
<th>194.47.148.199</th>
<th>local</th>
<th>194.47.148.95</th>
<th>ttl</th>
<th>255</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX:</td>
<td>Packets</td>
<td>Bytes</td>
<td>Errors</td>
<td>CsumErrs</td>
<td>OutOfSeq</td>
<td>Mcasts</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TX:</td>
<td>Packets</td>
<td>Bytes</td>
<td>Errors</td>
<td>DeadLoop</td>
<td>NoRoute</td>
<td>NoBufs</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Output that shows the parameters regarding the GRE Tunnel. Can be accessed by the command: ip -s tunnel in Linux Machine.

By putting together the above metrics that are measured both at the inlet of the tunnel and at the outlet of the tunnel, the calculations of delay jitters including reordering and packet loss are possible. The detailed description on how the calculations are implemented will be discussed in Chapter 5.2.

GRE configurations need to be made properly to establish the tunnel. The GRE_SEQ flag that is located on the 3rd bit of the GRE header needs to be set, otherwise the sequence numbers from the above mentioned methods can not be obtained. The GRE sequence number is defined and located in the 12th byte of the GRE header which can be reached and set by the following variable in the ipgre_tunnel_xmit method of the code.

tunnel->o_seqno;

Implementation Of Measurement Module For Seamless Vertical Handover
The next metric we need to set before sending the packet through the tunnel is the timestamp. The GRE module has not defined the timestamp field in GRE header structure. Therefore, it's necessary to add it manually by allocating two times four bytes of extra space. (the first four byte block represents seconds and the latter block represents microseconds).

Originally the 16th byte is allocated for Routing. Since it is optional, it is possible to use this space as the timestamp field. Next, this allocated space should be used for the timestamp information. Before the packet is transmitted through the tunnel, the timestamp value is appended to the GRE header in the ipgre_tunnel_xmit method with the following code line.

```
addend += 4; //allocate buffer for timestamp for the seconds portion.
addend += 4; //allocate buffer for timestamp for microseconds portion.
```

At the outlet of the tunnel, the metrics are obtained from the GRE header in the following code line in the ipgre_rcv method of the measurement module.

```
do_gettimeofday(&this_time);
tunnel->gre_dept_time_sec = this_time.tv_sec; //obtain the seconds on the departure side by adding this line
*ptr = htonl(tunnel->gre_dept_time_sec);
ptr--;

tunnel->gre_dept_time = this_time.tv_usec; //obtain the microseconds on the departure side by adding this line
*ptr = htonl(tunnel->gre_dept_time);
ptr--;
```

The above `ntohl()` function converts the unsigned integer that is located on the specified pointer offset from network byte order to host byte order [20]. The function is used to make the conversion between Little Endian and Big Endian. This conversion plays a role when the data types are stored in the memory and the way they are transmitted through internet.

The sequence number of the packets are added to the GRE header in the same way as described above.

There is also the ipip.h header file that keeps the fields of the GRE header. All the fields are able to get modified within this file. Therefore, the following timestamp variable lets us use `tunnel->gre_dept_time and tunnel->gre_dept_time_sec` as global variables.

```
unsigned long gre_dept_time_sec= htonl(*(__be32*)(h + offset));
unsigned long gre_dept_time_microsec= htonl(*(__be32*)(h + offset));
```
The departed packets' sequence numbers and the departure times are obtained at the end of the tunnel. In addition to these values, the time when the packet is received should be measured with 

```c
unsigned long gre_dept_time_sec;
unsigned long gre_dept_time_microsecs;
```

TimeStamp Counter (TSC) and Get Time Of Day (GTOD) are the two basic functions that are used in timestamp operations. TSC gets the time through an assembler call, while the GTOD gets the timestamp via a system call. In the thesis, GTOD is used in the measurement implementations.

Calculations are made while taking the input parameters as follows:

- **Departure_Time, the time when the packet was departed.**
- **Received_Time, the time when the packet was received**
- **Expected packet sequence number**
- **Actual received packet sequence number**

Measuring delay jitter in the implementation part is not that easy. Normally, clock synchronization has to be adjusted before taking measurements as has been discussed earlier in Chapter 1. The calculations that are made during the implementation are based on comparisons as reference to the initial obtained timestamps, thus the clock synchronization is considered to be maintained as long as it is assumed that the clocks are not drifting against each other.

Method 3 that has mentioned on Chapter 2 is implemented along one time interval periodically, however in a more comprehensive way. It is not implemented instantaneously but by considering the one-way-delays of all packets that are received in one complete time interval, and the calculation is finalized by taking the standard deviation of these obtained delays. Eventually, the delay variation per time interval is calculated. If the number of packets received per interval is less than three, then no delay variation can be calculated.

Departure time of the current packet will be subtracted from the reception time of the same packet. The difference values are used as inputs to the standard deviation calculation.

There are a few ways included in the code regarding the delay jitter calculation through finding the standard deviation. The first method is called the ring buffer method and it's preferred to be implemented because it's more efficient in terms of memory allocation, and faster for real time network analysis scenarios [2]. None of the previous data is stored in an array or in a storage buffer.
for further calculations. The standard deviation is calculated after each time interval. The other way that is implemented is more reliable especially in reordering scenarios. The departing and arrival parameters are stored separately in an array according to the received packets' sequence numbers. After that, the calculation of the standard deviation is made by using these stored parameters. But since the buffer is flushed and reused again in every one hundred packets, the buffer allocation is minimized. Nevertheless, the time complexity for this method is expected to be more as compared to the ring buffer method implementation since there is no need to allocate a buffer for storing parameters of the departure.

This delay jitter calculator method has basically three parameters obtained from the source. Since do_gettimeofday() method returns two variables tv_usec for microseconds and tv_sec for seconds, we treat these two variables separately and then combine them to obtain unique timestamp in milliseconds resolution. These two variables, i.e., seconds and microseconds, are sufficient enough for calculations in our experiments due to lack of congestion and utilization in the network. During the departure of a packet, the timestamp is stored in the extended field of GRE header by allocating two four bytes of data as mentioned in the code snippet previously. The difference between the timestamps that belong to the departed and the received packets are used as the reference to compare it with the difference of these metrics with the next packets.

5.2.1 Calculation of Delay Jitter

The parameters needed for the calculation is obtained with the following method.

```
static void store_packet_params (unsigned long recv_time_usecs, unsigned long recv_time_secs, unsigned long dept_time_usec, unsigned long dept_time_sec, unsigned long ul_seqno)
```

Basically, there are five parameters: the reception time in seconds level, the reception time in milliseconds level, the departure time in seconds, the departure time in milliseconds level and the departure sequence number. The departure sequence number is important since we should know which departure timestamp belongs to which departure sequence number. These parameters are passed in from the ip_gre_rcv() method which is invoked as soon as the packet arrives to the outlet of the tunnel.

After having the parameters at hand, the procedure showing how the delay jitter gets calculated is shown below.

The least significant bits of the reception time are clipped (recv_time_secs%1000) and then multiplied by 1000 ((thissstime_secs%1000)*1000)). A timestamp consisting of six digits is
obtained. Later, the timestamp data in milliseconds resolution is stored in the recv_time_secs will be clipped so that the most three significant digits are obtained. Finally, the current receive time is obtained (current_rcv_time). The current departure time is calculated within the same manner. The difference is found in the timestamp regarding the departure times.

\[
\text{current_rcv_time} = (((\text{recv_time_secs}\%1000)*1000)+\text{recv_time_usec}/1000));
\]
\[
\text{current_dept_time} = (((\text{dept_time_sec}\%1000)*1000)+\text{dept_time_usec}/1000));
\]

Once the first packets arrive to the outlet of the tunnel, the departure and the reception timestamps are stored.

\[
\text{if(ul_seqno000==1) } \{
\text{initial_receive_time} = (((\text{thissstime_secs}\%1000)*1000)+\text{thissstime}/1000))/\text{milliseconds}
\}
\]

Thus, the synchronized received and departure times for the later packets can be calculated. See the following code sniplet. The newly received packet's parameters are subtracted from the parameters that are obtained from the first packet. This is done for obtaining the clock synchronization by setting the initial time to \( T_0 \).

\[
\text{current_synched_rcv_time} = \text{current_rcv_time-}\text{initial_receive_time};
\]
\[
\text{current_synched_dept_time} = \text{current_dept_time-}\text{initial_dept_time};
\]

The time interval that consists of one second (1000 ms) is determined with the following code line.

\[
\text{current_interval} = (\text{current_synched_rcv_time}/1000);
\]

The next step is to find the transmitting time of each packet by taking difference between the departure and the received times as follows:

\[
\text{if(current_synched_dept_time}\geq \text{current_synched_rcv_time) }
\{
\text{differences_of_curr} = \text{current_synched_dept_time-}\text{current_synched_rcv_time};
\}
\]
\[
\text{else}
\{
\text{differences_of_curr} = \text{current_synched_rcv_time-}\text{current_synched_dept_time};
\}
\]

The magnitude check is being done prior to the calculation, since the data are stored as the type unsigned long.

When the new packet is arrived to the current interval, the recent mean (first moment) is recalculated as shown in line 10 and line 12. Since the metrics are type unsigned long, we have positive numbers that needs to be treated specially. This depends on the new magnitude of the transmitting delay of that packet. If the new magnitude is less than the mean it is subtracted, on the
other hand if it is larger it is added to the mean and a new mean value is calculated. Variables used in the below code snippet is described as follows:

**delta:** the time difference between the current transmission delay and the mean of transmitting delays since the beginning of the time interval.  
**m2:** the delta is multiplied by the new delta with respect to the new mean calculated.

```c
1  if(current_interval==oldinterval) {
2    count_packets++; //the packet number is increased by one packet
3    if(differences_of_curr>mean_new) //the current transmitting delay is compared to the mean.
4        delta=differences_of_curr-mean_new;
5    if(differences_of_curr<mean_new)
6        delta=mean_new-differences_of_curr;
7    if(count_packets>1) // to avoid divide by zero exception for line 10.
8    {
9        if(differences_of_curr>mean_new)
10           mean_new=(mean_new+(delta/(count_packets-1)));
11        if(differences_of_curr<mean_new)
12           mean_new=(mean_new-(delta/(count_packets-1)));
13        printk("DT: %u.", current_synched_dept_time);
14        printk("RT: %u.", current_synched_rcv_time);
15        printk("delta: %u.", delta);
16        printk("m2: %u.", m2);
17        printk("DC: %u.", differences_of_curr);
18        printk("MeaN: %u.", mean_new);
19        if(differences_of_curr>mean_new)
20           m2=m2+(delta*(differences_of_curr-mean_new)); //uses the new value of the mean.
21        if(differences_of_curr<mean_new)
22           m2=m2+(delta*(mean_new-differences_of_curr));} //uses the new value of the mean. }
23    }
24    old_synched_rcv_time=current_synched_rcv_time;
25    old_synched_dept_time=current_synched_dept_time; }
```

As the interval varies, the mean value obtained is utilized into the calculation of the standard deviation. The variance is calculated with help of the m2 value and the number of packets counted on the relevant interval. Finally, the delay variation (the delay jitter) is calculated by taking the square root of the calculated variance.
As can be seen, the mean value of the transmitting delay is altered each time the new packet has arrived. Thus, this prevents the storage of all the parameters in a buffer and allows for the calculations considering all the stored parameters at the end of each interval.

5.2.2 Calculation of Reordering and Packet Loss Metrics

Reordering Density:

The reordering density is measured by the method `static void reordering_density_check(unsigned long seqnumber, unsigned long expected_seqnumber)`. The distance of the received packets to the expected packets are calculated in this method. The calculations are made similar as explained in the Reordering Density part of Chapter 2. As the packet arrives along with the departed sequence number, it is checked with respect to the expected sequence number. If the difference is zero, then that packet is treated as the expected packet so the instantaneous reordering density is set to zero. However, if the departed sequence number and the expected sequence number are not found to be the same, then the difference in between the packets are taken into account. Thus, the calculation of the reordering density metric can take place for the whole time interval period as the other packets arrive. The code snippet shown below is used to count the number of packets with the concerned packet reordering distances.
When a packet is received, the expected sequence number is checked with respect to the received packet's sequence number and the \textit{discontinuity} flag is set to false (line 2). The number of packets that are zero distance away from the expected sequences are increased (line 3). The index that belongs to the expected sequence number is set to false, which represents that the expected sequence number is received and marks that we are not interested on that packet anymore (except for the detection of the duplicate packets). However if the expected packet has not arrived, it will be marked with true into array index of the expected packet number.

At line 8, the condition holds, when the expected sequence number is not equal to the actual received sequence. Therefore, \textit{discontinuity} flag is set to true.

There are two alternatives when the received packet sequence number is not equal to the expected packet sequence number.

- Received sequence number is greater than the expected sequence number
- Received sequence number is less than the expected sequence number.

```c
if(seqnumber>expected_seqnumber) {
  difference=seqnumber-expected_seqnumber; //find the reordering distance.
  if(expected_seqnumber=true) //mark if the exp. seq number has not reached.
    discontinuity_size[difference]++; //reordering density track.
} else if(seqnumber<expected_seqnumber) {
  difference=expected_seqnumber-seqnumber;
  discontinuity_size[difference]++;
  if([seqnumber]==true) {
    [seqnumber]=false; //delete the mark since it has received.
    printk("The previously lost %uth packet has arrived.",seqnumber);
    packet_loss_counter=0; //reset the counter.
  }
} else {
  packet_loss_counter++;
  printk("PacketLostCounter:%u", packet_loss_counter);
}
```

Implementation Of Measurement Module For Seamless Vertical Handover
At line 9, the expected packet number which is not received is marked as true and its packet number is copied inside an array f[], in addition to that the distance between the expected packet and the received packet is calculated and the index that belongs to the distance is incremented by one (see line 15).

Line 17 is the condition when the expected packet's sequence is greater than the actual received packet sequence number. There are two conditions:

- There was a missing packet and has just arrived,
- The previously arrived packet has arrived again, resulting in a duplicate packet.

After the reordering distance is measured for the incoming packet (line 19), the received packet's sequence number is checked according to the packet's index number in the array f[]. If it has not been received previously, it will be treated as a reordered packet that has arrived late. The mark indicating a missing packet in the array f[] should be set back to false for further checks. However, if it has already received, it will be treated as duplicate packet and nothing needs to be done in the duplicate packet scenario. There is also a variable called packet_loss_counter that tracks the number of packets received between the missing packet sequence and the expected packet sequence number. If this exceeds a threshold of four packets, or the reordering distance between the actual received packet and the current expected packet sequence is more than four packets, that traced packet (expected packet) will be treated as lost and will not be traced anymore. However, if the missing packet is arrived within the next four packet interval, the packet_loss_counter will be reset.

In every 20 packets, the reordering density check is being made (line 37, 38). The array that holds the missing packets are checked within this period (line 40). If there still exist missing packets, they will be counted as lost, and the lost_packet counter will be incremented accordingly. The expected sequence number is incremented by one packet after receiving a single packet.
Finally, the packets are counted and now we have a wider view of the current network. The results are expected as follows:

<table>
<thead>
<tr>
<th>Received Seq NO:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected Seq NO:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Results         :</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>(rd[expected_seq_number])</td>
<td>: 0 : 4/11, 1 : 6/11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Results show the absolute difference between the expected and received sequence number. There are four packets arrived in sequence while six packets are reordered.

Although the reordering density is caused by the loss of only one packet, it is treated as reordered with one packet with a 6 / 12 ratio that is considered to be quite high. However, this can be detected by writing a smarter algorithm.

### 5.2.3 Another Packet Loss Calculation (Fast simple but not reliable)

The most reliable way of calculating the packet drop rate for a long interval is to store all the packet properties like departure and receiving timestamps in a struct or in an array with their sequence numbers. The index content of the non-received packets will be set to zero. After a certain interval, the search mechanism will track the zero's and their indices of sequence numbers. This information is used both to calculate the packet drop rate by counting the number of zero's and to detect reordering. However, this consumes buffer space and is a slow processing time is expected. In traffic scenarios, it's sometimes hard to keep track of all missing packets since new buffer must be allocated for each expected non-arriving packet. Therefore, during the thesis, another algorithm is utilized.

The following algorithm does not need buffering or allocation of a storage for the lost packets to be traced in the interval. It calculates the packet drop rate in the simplest and fastest way. When the expected packet has not arrived, the packet loss counter is increased by one. As long as that packet is received later on the interval, packet loss counter is decreased by one. In this algorithm, see below code sniplet, duplicate packet free scenario is assumed. The results for this code piece can be seen in Chapter 6.
5.2.4 Reordering Detection and Calculation of DJ for Reordered Case

In the original version of the tunneling code, the packets that are out of sequence were forced to be dropped. This property of the code was preventing us to detect the reordered packets and measure the appropriate delays along with the received packets' departing sequence numbers and the timestamps. Therefore the packet drop function that is called when the out-of-sequence packets are detected is commented out.

```c
if (tunnel->parms.i_flags&GRE_SEQ) {
    if (!(flags&GRE_SEQ) ||
        (tunnel->i_seqno && (s32)(seqno - tunnel->i_seqno) < 0)) {
        stats->rx_fifo_errors++;
        stats->rx_errors++;
        // goto drop; // dont go to drop !!!
    }
}
```

The reason that we prevent the packets to be dropped for the analysis is the following. It's observed from the results of the delay jitter calculation, the NETEM tool generates delay jitter less than the specified value when the delay jitter that is introduced causes the packet to get
reordered. While checking the tcpdump timestamps, this can also be verified. Even though, there are still drawbacks for tcpdump, such as nanosecond time resolution, packet drop count and other counts and packet sequence numbers, it can be a reliable tool when no packet reordering is assumed. When this fact is accumulated with the fact of dropped packets on the netem tool buffer, the gap between the expected results and the set value on the netem tool increases even more. It can be claimed that the DJ values are reliable and can be measured if no packet drops are observed. The result is illustrated in Chapter 6.1.3

<table>
<thead>
<tr>
<th>Received Seq NO: 0 1 3 2 4 6 5 7 8 9 10 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected Seq NO: 0 1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td>Results : 0 0 +1 -1 0 +1 -1 0 0 0 0 +2</td>
</tr>
<tr>
<td>No of Results (rd[expected_seq_number]) : 0 : 7/12, 1 : 4/12, 2 : 1/12</td>
</tr>
</tbody>
</table>

There are seven packets arrived as expected, four packets arrived with a RD of 1 and one packet arrived with RD of 2.
Chapter 6

Results

Two versions of the measurement module are developed; an active measurement module (payload-dependent) and a passive measurement module (payload-independent). For the active measurement, a tester application is implemented in user space to send the packets for varying traffic scenarios. The kernel space measurement module receives the packets sent by this tester application, parses the payload data field of the packets to decapsulate the timestamps and sequence numbers.

6.1 Payload Dependent Approach

The test bed for the experiments and the final results regarding the Payload Dependent Approach is presented in this section of the chapter.

6.1.1 Procedure of the Experiment

• Load the measurement module into the kernel at the destination.
• Run the test module at the source.
• Examine the results (Hint: use the kernel commands in Appendix A: Chapter 8.1.4.
• Change the sequence of packets in the test module for different scenarios.
• Run the test module again.
• Examine and compare the results with the previous ones.
6.1.2 Reordering and Packet Loss Detection

Let's start with the following result observed after the activation of the code piece that is responsible from notifying the user in cases like packet loss in the measurement module. In the below example the threshold level for the number of packets after a loss of an expected packet is six. In other words, the algorithm waits for five packets for the expected but lost packet to arrive. If not arrived after the fifth packet, it notifies the packet loss case.

This is the result observed after the activation of the code piece that is responsible of notifying the user in cases like reordering in the measurement module.

A detection of a reordering and a packet loss, in fact the packet drop rate can be calculated just by looking at the packet sequence number.

6.1.3 Reordering Detection

At the end of the time interval, the module checks the number of packets arrived in $\gamma$ units. If there are some missing packets, the decision for packet loss and reordering is made according to the threshold value $\gamma$ that holds the predetermined packet sequence number.

Assume that the packet two is dropped during the propagation of the packet from the transmitter to the receiver in the first time interval. It's assumed that the measurement module has the threshold
set to $\gamma=10$ packets. After the packet with the sequence number 13 is received, the threshold value $\gamma$ is reached. The decision for the loss of packet two can now be made. Similarly, the decision for the loss of packet 17 will be made in the third time interval when the 28th packet is reached, that is shown by the crossover at the corresponding sequence number. By this functionality, the difference between packet loss and reordering can be clearly detected by assuming a scenario that packet two arrives after 9th packet, i.e., a reordering scenario, in time interval 1 in the below packet sequence.

**Time Interval 1:** 1-2 3 4 5 6 7 8 9 10

**Time Interval 2:** 11 12 13 14 15 16 17 18 19 20

**Time Interval 3:** 21 22 23 24 25 26 27 28 29 30

Thus, the packet loss scenario can end up with reordering within the threshold limits. As there exists packet loss, the measurement module keeps track of those statistics and calculates the packet drop rate by getting the ratio of dropped packets to the number of expected packets up to that time.

**Reorder Density Method Implementation**

Not exactly, but a similar method to reorder density that is mentioned earlier regarding reordering detection is also implemented. It tracks the reordering distance of the received packets. In other words, it compares the incoming packet distance to the expected packet distance as shown below: See Chapter 5.2.4.

| Zero Distance: 21, One Distance: 3, Two Distance: 1, Three Distance: 0, Lost Packets: 0 |

The above example shows that there are 21 packets arrived as expected. There are three packets arrive with distance one and one packet arrived with distance two. Distance two means that the number of packets in between the expected packet and the actual received packet is two. By the above report, the analyzer gets a wider view of how much the packets are reordered with the number of packet distances. However, these notifications and the result we can obtain can not be reliable enough to conclude and give a decision about the network. Therefore, we need to look into more metrics like delay and delay jitter among the packets.

**6.1.4 Implementation of the Jitter Calculation based on Standard Deviation**

There are different ways of calculating delay jitter, but throughout the thesis, it is aimed to find one that calculates and gives out the most reliable results with a low overhead and in the fastest possible way. The delay jitter calculation is implemented with ring buffer method [2]. It's preferred to be implemented because it's more efficient in terms of memory allocation, and faster in real time.
network analysis scenarios. None of the previous data is stored in an array or in a storage buffer for further calculations. It calculates the standard deviation after a specific number of packets are received while keeping the first and second moment updated. In this section, the comparison between the ordered and reordered packet scenarios are made by analyzing the delay jitters.

This method calculates the variation of the arrival times of the packets and has three parameters. To start with, only arrival times of the packets were used as parameters during testing. The reception time of a packet is used as the reference to compare it with the other packets. The arriving time differences in between packets are illustrated (T1, T2, T3, etc.) as follows:

![Illustration 1: Illustrating one complete time interval.](image)

**Ring Buffer Method**

The tests for this method is done through a user space tester module. This module creates and sends UDP packets to the module in kernel space in microseconds resolution. Therefore the results are in microseconds resolution. The user space tester module is used as the traffic generator.

The following data is collected and results are interpreted according to the ring buffer method discussed in [2]. In order to test whether the implementation provides accurate results, some prior experiments are done. Below is a bunch of data collected from a random interval. As a result of the calculations, the mean is found to be 624 microseconds, while the standard deviation is 1070 microseconds. This is compared with the expected values. The expected mean according to the data is 702 microseconds with a standard deviation of 922 microseconds. Hereby, the measurement results are accepted. There exists approximately ten percent error rate in the calculations. This error is due to the round-off and the square root algorithm that is implemented seperately since most of the math functions can not be called from within kernel.

Implementation Of Measurement Module For Seamless Vertical Handover
After comparing the expected and obtained results with the above sample, the method is verified. It's assured with more experiments that the implementation is giving more accurate results. Another experiment is as follows:

Illustration 2: Illustration of the packet reception times. Clumping of six packets are observed between packet #2 and packet #7. Packet receiving time is given as in microseconds.

Illustration 3: Illustration of the packet reception times. Packet receiving time is given as in microseconds.
The increase of the arrival times of the packets is approximately linear and constant resulting in a mean of 28 microseconds with a standard deviation of 32 microseconds where the expected mean is 31 microseconds with an expected standard deviation of 28 microseconds. This method is quite fast but is harder to manipulate the implementation when the reordered distance of the packets are large. Even though the results seem like to be considerable with ten percent error rate in the ring buffer method, the calculation of the standard deviation is tested with another method of implementation to obtain more accurate results.

The below data is taken from the implementation of the ring buffer method. The upper bar represents the departure times and the lower one represents the arrival times. Since they are not taken from the same machine, departure and reception times are not adjacent. But they show synchronized behaviour.

**Illustration 4:** Departure and reception times are plotted.
The mean of differences belongs to the interpacket delays is 220 microseconds.

Below is an other sample taken from a ordered sequence between 30th and 60th packets. The mean is calculated as 186 microseconds with a standard deviation of 261 microseconds.

*Illustration 5: Illustrating the interpacket delays, i.e.,
 \[ Jitter (n) = \text{abs}(\text{Arrival}_n - \text{Arrival}_{n-1}) - (\text{Departure}_n - \text{Departure}_{n-1}) \]

The mean of differences belongs to the interpacket delays is 220 microseconds.

Below is an other sample taken from a ordered sequence between 30th and 60th packets. The mean is calculated as 186 microseconds with a standard deviation of 261 microseconds.

*Illustration 6: Interpacket delays of packets are illustrated.

The mean and standard deviation calculated by this method varies between the following values
depending on the traffic and utilization. The test has been done and standard deviation is calculated with 9 consecutive intervals, and the results are reported in the following table:

<table>
<thead>
<tr>
<th>Interval (30 packets)</th>
<th>Mean (microseconds)</th>
<th>Standard Deviation (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7873</td>
<td>2309</td>
</tr>
<tr>
<td>2</td>
<td>261</td>
<td>185</td>
</tr>
<tr>
<td>3</td>
<td>3685</td>
<td>946</td>
</tr>
<tr>
<td>4</td>
<td>159</td>
<td>129</td>
</tr>
<tr>
<td>5</td>
<td>2033</td>
<td>6683</td>
</tr>
<tr>
<td>6</td>
<td>3428</td>
<td>2997</td>
</tr>
<tr>
<td>7</td>
<td>144</td>
<td>185</td>
</tr>
<tr>
<td>8</td>
<td>7454</td>
<td>544</td>
</tr>
<tr>
<td>9</td>
<td>2210</td>
<td>7363</td>
</tr>
<tr>
<td>10</td>
<td>1026</td>
<td>3787</td>
</tr>
<tr>
<td>11</td>
<td>740</td>
<td>2279</td>
</tr>
</tbody>
</table>

The values in column two varies due to the oscillations caused by the utilization and packet processing times for the CPU’s. The maximum mean of 7873 us and the minimum of 144 us show the expected boundaries.

During the implementation where the comparison of the differences between the departure and arrival times of consecutive packets are made, there is a tricky point that needs to be explained before going further with the results.

\[ \text{Jitter (n) = } \text{abs}([\text{Arrival}_n - \text{Arrival}_{n-1}] - [\text{Departure}_n - \text{Departure}_{n-1}]) \]

During the coding; the timestamps, sequence numbers, and all other numerical parameters obtained from the packets are allocated in the buffer as unsigned long. Therefore, even if the current difference between the parameters of interest are smaller than the ones in the previous packets, negative values can not be obtained.

In a normal sequence packet flow, the standard deviation and mean are obtained to be as expected in the implementation results. Consider the ordered packet sequence 0 to 11. The packet delay is shown as in Illustration 7. The mean is measured as 99 microseconds with a standard deviation of 29 microseconds.
Jitter(n) formula depicted in the previous page shows how the delay varies with respect to the previous packet. The smoothness of the variation of the delay can be observed with that equation.

**Reordering Method 1**

**Calculation of Variation of Arrival Times for Reordering**

The following illustration is showing a scenario in which the packets arrive out of sequence. It's clear that the reordering seems like to be occurred just because of a single packet arriving earlier than the expected packet.

*Illustration 7: Illustration of inter-packet delays in microseconds. PDV is calculated per packet.*

The following illustration is another example of the reordered sequence packets. Packets two and three, six and seven are reordered with reordering size of one packet.

*Illustration 8: Reordered packets with RD of 1 packet is illustrated.*
Method two is sufficient and is implemented to calculate the delay jitter with simple reordering scenario as above. We illustrated the reordering scenario with reordering size of one. Since there are not much time gaps between the reordered and the expected packets in this example, the following method which calculates the variation of the delay of the arrival time of the packets of consecutive packets is also satisfactory for this case. (See the Appendix A: Chapter 8.1.7).

Assume the following sequence of packets sent and received respectively.

**Sent Packet Sequence:** 1 2 3 4 5 6 7 8 9 10 11

**Received Packet Sequence:** 1 2 3 4 5 7 6 8 9 10 11

In that case, packet six is delayed by one packet distance and results in clumping since both sixth and eighth packet arrives at the same time.

---

![Illustration 9](image-url)

*Illustration 9: Illustration of a reordered packet scenario. Packet #41 and #42 are clumped*
The mean is 874 microseconds with a standard deviation of 2264 microseconds. In Illustration 10, 41st and 42nd packets arrive simultaneously and resulting a peak.

**Reordering Method 2**

**Calculation of Variation of both Departure and Arrival Times for Reordering**

This method has implemented as the most reliable and adaptable method for any kind of reordering scenarious. During the periodic calculations, in every twenty packets, the received packet parameters are stored in an array. After twenty packets are received, the procedure continues with the calculation of the differences of the received times among consecutive packets. This implementation seems simple at a first glance, however the reliable thing of this method is that it can handle the delay jitter as a result of standard deviation calculations in scenarios especially like in reordering. As soon as the packet arrives, the parameters as depart and received times are stored in the array according to their sequence numbers. The importance of the functionality of this method gets more significant if the distance between the reordered packets are high.

![Drawing 3: Reordered packet scenario](image)

The following two graphs show a sequence of the ordered packets. The validation of standard deviation and mean values are made by experimenting the method on ordered packet sequence. The results are as shown below:
Illustration 10: Ordered packet scenario is illustrated with higher utilization. Inter-packet delays are observed to be oscillating.

The mean of the above data is calculated as 610 microseconds with a standard deviation of 860 microseconds. There are fluctuations between packet 80 and 84 due to the change of utilization of the network.
The mean of the above data is calculated as 631 microseconds with a standard deviation of 633 microseconds. Similar behaviour with the previous graph is observed in this graph as well. Another reason behind is that the time resolution is microseconds.

Illustration 11: Ordered packet scenario is illustrated with higher utilization. Inter-packet delays are observed to be oscillating.

The mean of the above data is calculated as 631 microseconds with a standard deviation of 633 microseconds. Similar behaviour with the previous graph is observed in this graph as well. Another reason behind is that the time resolution is microseconds.
The mean value is 653 microseconds with a standard deviation of 547 microseconds. The delay variation is high until the 69th packet when the formula in the caption is used. This is due to the utilization of the network traffic since these experiments are conducted by taking one end of the tunnel in Malmo and the other in Karlskrona.

In case of a complicated reordering scenario in Illustration 14, the following results with the concerned results are obtained. The upper points represent the departure times of the packet in the sender module that is located in Malmö, on the other hand the lower points represent the receiving times of the sent packets. The reordered packet is the 40th packet and results a high difference with respect to the average difference within the interval.

Illustration 12: Inter-packet delays are illustrated.

Implementation Of Measurement Module For Seamless Vertical Handover

68
The mean is calculated to be **9931 microseconds** with a standard deviation of **13282 microseconds**.

**Illustration 13**: Reordering is observed in packets 39 and 40

**Illustration 14**: Departure and reception times of the packets are illustrated

The mean is calculated to be **9931 microseconds** with a standard deviation of **13282 microseconds**.
The reordered scenario results in higher values for mean and standard deviation as compared to ordered scenarios illustrated previously.

### 6.2 Payload Independent Approach

#### 6.2.1 Procedure of the Experiment

For the second part of the thesis, the methods for calculating delay jitter are embedded into the already implemented tunneling module module where the parameters are appended into the GRE header instead (in module ip_gre.ko). Along with this passive measurement module, there is no need to add parameters and metrics into the payload of the sent packet from the user space. This lets us to work completely in kernel space environment. The procedure for passive measurement is the following:

- Load the Measurement_Destination module to the destination kernel.
- Load the Measurement_Source module to the source kernel.
- Configure and establish the GRE.
- Adjust the delay jitter as desired with the Netem tool (Netem commands in Chapter 8.1.5).
- F-Ping between the ends of the tunnel and assure that the tunnel is established. (See Chapter 6.2.3)
- Examine the results (Hint: Kernel commands in Appendix A: Chapter 8.1.4).
- Change the delay jitter settings with the Netem tool.
- F-Ping the source kernel to the destination kernel (among the ends of the GRE tunnel).
- Examine and compare the results with the expected results.

Experiment Design:

**Experiment Design for Payload Independent measurement module.**

- Establish the GRE tunnel among peers as described in Chapter 8.1.3
- Tunnel measurement modules on both source and destination

The Payload Independent measurement module is tested to calculate delay jitter with the tested methods on Payload Dependent measurement module in the previous chapters. The tunneling modules are loaded to kernel in both sides, destination and source. The measurement is done at the destination side independent from the packets sent from the source. The packet sequence numbers and departed timestamp is put to the header on the source side. Through the packet's journey
through the tunnel in real life, it's expected to obtain delays, reordering and packet loss. When the packet arrives to the receiver side, the parameters like departing packet's timestamp and sequence number are parsed. These parsed departure parameters are compared with the arrival sequence numbers and timestamps, so is calculated the one way delay jitter and packet drop rate. When the packet arrives, the difference between the departure timestamp and the arrival timestamp current packet is calculated. The comparison is done by a simple subtraction. And the final values after subtraction are stored either in an array in wait state or by updating the first and second moments with the new metrics when each packet is received.

One other drawback is the unsynchronized form of the source and destination. This can be solved by setting time to 0, for both the departure and reception time, with the acceptance of the initial packet, and treating the rest of the packets with reference to zero initial time, $T_0$.

The delay jitter calculation is made after a specific amount of packets are reached, e.g., time interval or packet interval.

6.2.2 Packet Interval Approach (30 packets/Interval)

In order to test whether the passive measurement module is calculating the delay jitter as expected, the tool called 'Netem' is used. It provides the user to set the delay jitter to a desired value. After the departed packets are timestamped and the sequence number is appended at the GRE header at the source, the packets can be manually queued in the source buffer in a desired way. If it is desired to set the delay jitter to 10ms, the following command should be used [22].

```
# tc qdisc change dev eth0 root netem delay <delay> 10ms
```

After setting the delay jitter with the above line, the results of the measurement module on the other end of the tunnel is observed. The data in the graphs on the next page are obtained by the 6th equation stated in Chapter 2.7.1 (Standard Deviation). The obtained results are as illustrated on the next page.
The delay jitter is set to 5 ms, and the graph is almost constant at 5 ms. The measurement module has results more close to expected when low delay jitters are introduced.

The delay jitter is set to 30 ms by NETEM. It is observed that the measurement module shows delay jitter values with a mean of 23 ms.
• 

```bash
# tc qdisc change dev eth0 root netem delay 100ms 50ms
```

The delay jitter is set to 50 ms by NETEM. It is observed that the measurement module shows almost constant trend deviating around 50 ms.

•

```bash
# tc qdisc change dev eth0 root netem delay 150ms 70ms
```

The delay jitter is set to 70 ms by NETEM. It is observed that the measurement module is introducing more oscillations around 58 ms.
The delay jitter is set to 100 ms by NETEM. It is observed that the measurement module is almost fixed at 80 ms.

The delay jitter is set to 250 ms by NETEM. It is observed that the measurement module is almost fixed at 180 ms.
The delay jitter is set to 1000 ms by NETEM. It is observed that the measurement module is almost fixed at 580 ms.

As the set value of the delay jitter from NETEM increases, the measurement module is not increased in the same way, moving away from linearity. This is illustrated in Illustration 16.
6.2.3 Time Interval Approach (one second interval)

The results showing on the graphs below are the delay jitter outputs obtained when the delay jitter is set on the sender side of the tunnel with two tools where the first one is called NETEM while the latter is called a tool called FPING [23]. It's not possible to use ping since ping sends the icmp packets one packet per each second. The experiments are desired to test within one second time intervals with various delay jitters within one second. Therefore, by using the 'fping' it is possible to send as many packets within per second. The ring buffer method, with only concentration on the arrival times to give a brief idea how the ring buffer method is implemented, is included on the following code sniplet. Departure times were not included in this code sniplet, in order to keep the code short for now. However, the following results and the results in Appendix A: Chapter 8.1.11 are obtained by using equation 6 from Chapter 2.6.1, and when the departure times are included to the ring buffer algorithm.

Illustration 15: Illustration of DJ results from NETEM and measurement module
The adjusted delay jitter by the Netem tool is expected to be measured as the same magnitude in the measurement module. The last parameter that has been set in the below command shows the delay variation applied to the network system. The corresponding graph below the command illustrate the actual delay jitter values measured with the measurement module. It is observed that the delay jitter values are close to the set parameter values which we can comfortably say that the measurement module gives more reliable values as the delay jitter decreases.

```c
current_interval=(current_synched_rcv_time/1000); //current interval consists of 1 second. It prevents buffer overflow. No need to store all the packet parameters.

if(current_interval!=oldinterval){
if(count_packets>1){
variance_new=m2/count_packets-1;
delay_jitter=sqrt(variance_new);
printk("DJ: %u",delay_jitter);
mean_new=0;
m2=0;
count_packets=0;
oldinterval=current_interval;
}
if(current_interval==oldinterval){
count_packets++;
if(count_packets!=1)
if(differences_of_recv_times>mean_new) {delta=differences_of_recv_times-mean_new;}
if(differences_of_recv_times<mean_new) {delta=mean_new-differences_of_recv_times;}
if(count_packets>1)
{mean_new=mean_new+(delta/count_packets-1);
if(differences_of_recv_times>mean_new)
{m2=m2+(delta*(differences_of_recv_times-mean_new));} //uses the new value of the mean.
if(differences_of_recv_times<mean_new)
{m2=m2+(delta*(mean_new-differences_of_recv_times));} //uses the new value of the mean.}
old_synched_rcv_time=current_synched_rcv_time;
old_synched_dept_time=current_synched_dept_time;
```

Implementation Of Measurement Module For Seamless Vertical Handover
15 seconds of observation shows that the values are oscillating in between 20-25 ms when the delay jitter is set to 20 ms.

22 seconds of observation shows that the values are oscillating in between 55-70 ms when the delay jitter is set to 50 ms.
The delay jitter is set to 80 ms, the measurement module shows data with a mean of 80 ms that is varying between 60 ms and 120 ms.

When the delay jitter is set to 100 ms, the oscillations in the graph increases. The data set is concentrated on 125 ms.

When the delay jitter is set to 100 ms, the oscillations in the graph increases. The data set is concentrated on 125 ms.
When the delay jitter is increased to 120 ms, the data deviates between 80 ms and 145 ms with a mean of 118 ms.

The dataset is deviating around 150 ms as has been set with NETEM. However, there are oscillations observed.
With 30 seconds of observation of the network traffic with the measurement module, it can be concluded that the delay jitter is stable with a mean of 200 ms, even though oscillations are experienced.

```
# tc qdisc change dev eth0 root netem delay 500ms 20ms
```

```
# tc qdisc change dev eth0 root netem delay 600ms 30ms
```

```
Delay Jitter(ms) vs Time Intervals
Each Time Interval is 1 second
```

```
Delay Jitter vs Time Interval
Each Time Interval is 1 second
```

Implementation Of Measurement Module For Seamless Vertical Handover
As we plot the data from NETEM and measurement module as in Illustration 17, the corresponding linear trend is observed as described:

Illustration 16: Illustration of DJ results from NETEM and measurement module

Illustration 17: Input from Netem and output from MM is depicted. There exists a correlation with a linear equation stated on the graph and with $R^2 = 0.99$
6.2.4 TCPDUMP versus Measurement Module

TCPDUMP is known to be a reliable tool amongst researchers that is believed to be showing the truth in network monitoring. Therefore, to verify the measurement module results, the outputs of the module are compared with the tcpdump as shown in Table 4.

Let’s consider the received timestamps. The delta of the timestamps between the adjacent packets (see column two) are approximately equal to the delta of the timestamps of the same packets according to the measurement module (MM) results (see column four). However, it is seen that the case is not correct for the departure times. The reason is that the departure times are timestamped before it is entered into the Netem tool which is used to shape the traffic according to the desired delay variation values. That’s the reason why the departure MM delta/ms values are identical in the eighth column in contrast to the delta values at the receiver side in column four.

The tests have been made continuously for 10 times by using the bash script shown on the next page. The script loads the relevant modules to both ends of the tunnel, to two Linux boxes in our case. The script prompts the user to enter the desired delay and delay variation to be set on the NETEM tool (see Appendix A: Chapter 8.1.5) as well as the the IP address of the local and remote ends of the tunnel. The number of tests to be done is also asked to the user, so that the sender Linux box will set the GRE tunnel. Once these required arguments are entered, the testing starts. The connection to the other Linux box that holds the end of the tunnel is reached by ssh. Public/Private key encryption is used for the ssh authentication. The bash not only runs the required commands to establish the tunnel in the departing side, but also connects through ssh to the other Linux box and runs the appropriate script that is stored on the other Linux box to configure the receiving side tunnel end. The ssh connection is terminated after each test is complete. Following is the script that is used to automate the experiments.

<table>
<thead>
<tr>
<th>Tcpdump RX</th>
<th>Tcpdump Delta/s</th>
<th>MM RX</th>
<th>MM delta/ms</th>
<th>Tcpdump TX</th>
<th>Tcpdump Delta/s</th>
<th>MM TX</th>
<th>MM Delta/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.000000</td>
<td>.207947</td>
<td>207</td>
<td>207</td>
<td>27.207953</td>
<td>.207613</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>27.418219</td>
<td>.210266</td>
<td>418</td>
<td>211</td>
<td>27.527837</td>
<td>.210257</td>
<td>403</td>
<td>202</td>
</tr>
<tr>
<td>27.614785</td>
<td>.196566</td>
<td>614</td>
<td>196</td>
<td>27.724420</td>
<td>.196553</td>
<td>604</td>
<td>201</td>
</tr>
<tr>
<td>27.823190</td>
<td>.208405</td>
<td>823</td>
<td>209</td>
<td>27.932797</td>
<td>.208377</td>
<td>805</td>
<td>201</td>
</tr>
<tr>
<td>28.007246</td>
<td>.184056</td>
<td>1,007</td>
<td>164</td>
<td>28.118835</td>
<td>.184038</td>
<td>1007</td>
<td>202</td>
</tr>
</tbody>
</table>

*The above results are in milliseconds level.

*Table 4: Measurement module vs tcpdump

Let’s consider the received timestamps. The delta of the timestamps between the adjacent packets (see column two) are approximately equal to the delta of the timestamps of the same packets according to the measurement module (MM) results (see column four). However, it is seen that the case is not correct for the departure times. The reason is that the departure times are timestamped before it is entered into the Netem tool which is used to shape the traffic according to the desired delay variation values. That’s the reason why the departure MM delta/ms values are identical in the eighth column in contrast to the delta values at the receiver side in column four.

The tests have been made continuously for 10 times by using the bash script shown on the next page. The script loads the relevant modules to both ends of the tunnel, to two Linux boxes in our case. The script prompts the user to enter the desired delay and delay variation to be set on the NETEM tool (see Appendix A: Chapter 8.1.5) as well as the the IP address of the local and remote ends of the tunnel. The number of tests to be done is also asked to the user, so that the sender Linux box will set the GRE tunnel. Once these required arguments are entered, the testing starts. The connection to the other Linux box that holds the end of the tunnel is reached by ssh. Public/Private key encryption is used for the ssh authentication. The bash not only runs the required commands to establish the tunnel in the departing side, but also connects through ssh to the other Linux box and runs the appropriate script that is stored on the other Linux box to configure the receiving side tunnel end. The ssh connection is terminated after each test is complete. Following is the script that is used to automate the experiments.

<table>
<thead>
<tr>
<th>Tcpdump RX</th>
<th>Tcpdump Delta/s</th>
<th>MM RX</th>
<th>MM delta/ms</th>
<th>Tcpdump TX</th>
<th>Tcpdump Delta/s</th>
<th>MM TX</th>
<th>MM Delta/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.000000</td>
<td>.207947</td>
<td>207</td>
<td>207</td>
<td>27.207953</td>
<td>.207613</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>27.418219</td>
<td>.210266</td>
<td>418</td>
<td>211</td>
<td>27.527837</td>
<td>.210257</td>
<td>403</td>
<td>202</td>
</tr>
<tr>
<td>27.614785</td>
<td>.196566</td>
<td>614</td>
<td>196</td>
<td>27.724420</td>
<td>.196553</td>
<td>604</td>
<td>201</td>
</tr>
<tr>
<td>27.823190</td>
<td>.208405</td>
<td>823</td>
<td>209</td>
<td>27.932797</td>
<td>.208377</td>
<td>805</td>
<td>201</td>
</tr>
<tr>
<td>28.007246</td>
<td>.184056</td>
<td>1,007</td>
<td>164</td>
<td>28.118835</td>
<td>.184038</td>
<td>1007</td>
<td>202</td>
</tr>
</tbody>
</table>

1 Note that the measurement module delta is converted form to of the tcpdump delta to milliseconds resolution as explained in Chapter 5.
#!/bin/sh
echo "Enter delay: ">
read delay
echo "Delay is set to $delay."
read jitter
echo "Jitter is set to $jitter."
read packetno
echo "Number of Packets is set to $packetno."
read lip
echo "Local IP is set to $lip."
read rip
echo "Remote IP is set to $rip."
echo "Enter # of Loops: ">
read loop
echo "Remote IP is set to $loop."
for i in `seq 1 $loop` ;
do
ssh -v $rip "ls -l; date; cd /home/selim/Perimeter/; ./grercv15dec.sh; exit;"
l s
rmmod ip_gre_dec14_send.ko
insmod ip_gre_dec14_send.ko
ip tunnel add perimeter mode gre local $lip remote $rip ttl 255 seq key 666
ip link set perimeter up
ip addr add 10.2.2.1 dev perimeter
ip route add 10.1.1.0/24 dev perimeter
tc qdisc change dev eth0 root netem delay "$delay"ms "$jitter"ms
fping 10.1.1.1 -c $packetno -p $delay 5000
    echo $i
    sleep 4
done
rmmod ip_gre_dec14_send.ko
6.2.5 Reordering Scenario Results

<table>
<thead>
<tr>
<th>Exp. Seq</th>
<th>Rcv’d Seq</th>
<th>Dept. Timestamp</th>
<th>Rcvd. Timestamp</th>
<th>Delta</th>
<th>STDEV</th>
<th>MM Delay Jitter (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>202</td>
<td>241</td>
<td>39</td>
<td>42.96</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>101</td>
<td>241</td>
<td>140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>302</td>
<td>332</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>403</td>
<td>504</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>502</td>
<td>509</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>604</td>
<td>724</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>805</td>
<td>785</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>705</td>
<td>785</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>906</td>
<td>928</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1007</td>
<td>1089</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>1107</td>
<td>1175</td>
<td>68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>1208</td>
<td>1192</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>1309</td>
<td>1408</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>1409</td>
<td>1535</td>
<td>126</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>1510</td>
<td>1611</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>1610</td>
<td>1715</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>1711</td>
<td>1796</td>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>1812</td>
<td>1872</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>1912</td>
<td>1946</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>2013</td>
<td>2118</td>
<td>105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>2114</td>
<td>2179</td>
<td>65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>2214</td>
<td>2209</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>2315</td>
<td>2369</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>2416</td>
<td>2388</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>2516</td>
<td>2495</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>2718</td>
<td>2782</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>27</td>
<td>2617</td>
<td>2782</td>
<td>165</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The output above are in milliseconds level when reordering is introduced to the network. It is observed from the outputs that packets two and three are reordered as well as the packets 28 and 27. Event though, they are reordered they reach the destination at the same time. The standard deviation (delay jitter) calculated by the measurement module is same as the standard deviation value calculated by Excel tool according to the results obtained in column 5 (in milliseconds). This data is presented to verify the standard deviation algorithm used in the implementation of delay jitter calculation in the measurement module.
6.2.6 Method Validation Test

The validity of the results calculated by the measurement module is important from the analysis point of view. Since the delay jitter is calculated as the standard deviation of the transmission delays, Graph 1 shows how well the implementation of standard deviation fits into the standard deviation calculations of these delays calculated by Excel. The graphs show the results after eight consecutive tests are applied on the measurement module. The delays are recorded and calculated in Excel and through the measurement module independently. The data is attached to Appendix A: Chapter 8.1.1.

<table>
<thead>
<tr>
<th>Excel DJ (ms)</th>
<th>MM DJ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.45</td>
<td>17</td>
</tr>
<tr>
<td>14.55</td>
<td>14</td>
</tr>
<tr>
<td>13.62</td>
<td>13</td>
</tr>
<tr>
<td>8.39</td>
<td>9</td>
</tr>
<tr>
<td>18.84</td>
<td>18</td>
</tr>
<tr>
<td>9.1</td>
<td>9</td>
</tr>
<tr>
<td>10.3</td>
<td>10</td>
</tr>
<tr>
<td>16.9</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5: Comparison of measurement module and Excel. Data in both columns are shown in milliseconds.

Reorder Density

The reorder density results are as the data on the right. The reordered packets that are detected are in bold. The output with respect to the data is given on the next page. The results are according to the reordering density implementation as explained in Chapter 5. The reordering distance for each packet is calculated and printed.

\[
\text{tc qdisc change dev eth0 root netem delay 100ms 100ms}
\]

The above command describes that the delay in between the packets are set to 100 ms together with the 100ms delay variation. Thus, the delay varies in between 0 ms and 200 ms.
The 0 distance is 27. The 1 distance is 8. The 2 distance is 0. The 3 distance is 0. Lost Packets: 0

The output above show that there are 27 packets arrived as expected. eight packets arrived with one packet distance in between the expected packet number. There are 35 packets in total.

6.2.7 Reorder Check

A more comprehensive method is implemented to keep track of the reordered packets to treat them as lost after waiting for a limited threshold. In this case, it is set to four packets. The expected value is checked against the actual received packet. The reordering distance is taken but also the track of the lost packets are being made. Line 3 and line 4 are highlighted with bold in the below output show a sample of the output in reordered packet scenario is introduced to the network. As can be seen, the 9th packet arrived before 7th and 8th packets. Thus, a notification 'Reordering Occurred' is appeared on the output. Later it is observed that the 7th packet is arrived, thus resulting that packet to be deleted from the lost packet array and notifies the user as 'The previously lost 7th packet has arrived'. The packet loss is determined with respect to the distance of the packet to the expected packet sequence number and is also determined when the expected packet is not received within the after next 4th packet interval. PacketLostCounter at line 4 starts to count the number of packets after a missing packet. If this variable reaches 4, then the missing packet will be treated as lost and will increment the variable packetloss by one. See Chapter 5.2 for the Implementation. Finally, after

<table>
<thead>
<tr>
<th>Expected</th>
<th>Received</th>
<th>Expected</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
every 20 packets, the reordering density is calculated as can be seen in line 12. There are 12 packets that arrived as expected, on the other hand there exists 8 packets that arrived with one packet distance in between the expected and the received packet.

```
```

The 0 distance is 12. The 1 distance is 8.
### 6.2.8 Packet Loss Results

**e:** expected sequence number of the packet.

**s:** actual received packet's sequence number.

**Packet loss:** packet loss values are updated after every packet reception.

<table>
<thead>
<tr>
<th>e</th>
<th>s</th>
<th>Packet Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>46</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 6: Another Packet Loss Calculation (Fast Simple but Not Reliable)*

When a packet is not received when expected, the number for packet loss is incremented by one, and when it is received at later times within a time interval, the packet loss is again decremented by one.
Chapter 7

Conclusion

Measurements have significant importance over the heterogeneous seamless vertical handover mechanisms where critical decisions are made. Kernel space measurement module implementation is presented in this thesis, and is aimed to be used as a support for vertical handover mechanisms. There are numerous tools developed for network measurements. This implemented measurement module makes a difference while including the departing parameters of the IP packets into the measurement process. With the lack of departing parameters, in cases when the sender does not generate packets at constant rate, no transmitting packet latency difference (delay jitter during transfer) can be observed, resulting in incorrect delay jitter measurements. This thesis provides a solution to this problem by involving the departure parameters, e.g., the timestamp and the packet sequence number, to the calculations.

The measurement module is implemented in two different approaches. The former is as described as in Chapter 5.1 (Payload Dependent approach) and the latter is as described as in Chapter 5.2 (Payload Independent approach). Even though both of the approaches are implemented on kernel space, the challenging advantage of the second one is that it provides a measurement independent of the packet contents transferred, and can be considered to be working as a passive measurement module completely within kernel space. The first approach is used to test various delay jitter calculation algorithms together with reordering and packet loss detection methods.

The first approach was tested with the parameters sent from a complementary module in user space which is called as the tester module. This is done by adding the departure timestamp together with the departure sequence number to the data field of the UDP packet. The parameters sent from the user space would be then parsed and appropriate calculations are made in the measurement module that is loaded to the Linux kernel. During the tests of the first module, sender box was located in Malmö and the receiving measurement module was located in Karlskrona, Sweden. This is considered to be a significant drawback since there are many obstacles like the number of routers and varying switching times that, can effect the measurement results through a single packet's journey. Despite all of the above mentioned facts, this module has helped for the improvement of the methods and tests.

The second approach consists of a measurement module that functions completely in kernel space. Once the tunnel is established among the source and destination, accurate results can be obtained in
milliseconds level. Tunneling module is loaded to the source, in a way that, when the packet enters into the tunnel, it is encapsulated with GRE tunneling header. Inside the header, the timestamp and packet sequence number parameters are injected. The same module is also loaded to the other end of the tunnel, and decapsulates the GRE header when it is arrived to the outlet. The parsed departure timestamp and the sequence number helps measurement module to calculate the DJ and the PDR. The test is done by using a tool called NETEM. It helps the sender to adjust the packets with desired delay jitter, packet loss and reordered forms before the packets leave the Ethernet interface. Relying on the set values on NETEM tool is not enough. When the delay jitter is calculated with the help of TCPDUMP on the receiver side, it is observed that the resulting DJ values are slightly less than the set values via NETEM tool. It is also observed that the TCPDUMP results at the receiver are the same with the results obtained from the measurement module. The tests are applied for 10 times and the results can be found in Chapter 6.2.2, Chapter 6.2.3 and Appendix A: Chapter 8,1,11. With a slight tuning the code, the results appear as expected. The experiments are repeated for various reordering scenario cases and also for the scenarios where there exits packet loss.

7.1 Experienced Problems

Problems experienced while finding answers to the research questions are as follows:

- Kernel programming is not widely preferred, and not very common since it is sophisticated. It has intrinsic complexity and consists of a long debugging cycle.
- Debugging an operating system may require installing a new kernel and rebooting the machine in every debugging cycle.
- Need a lot of care to do Linux kernel network programming.
- Clock synchronization between the linux boxes are problematic during implementation.
- Various methods that has been discussed in Chapter 2 and in [6] have advantages and drawbacks for different scenarios and applications used. Choosing the best fit for all scenarios is sometimes problematic.

Along with the complexity of the kernel level programming, most of the time it was problematic to restart the operating system each time the kernel is disturbed by the code applied. Most of the time, the reason for the kernel to malfunction were pointer issues, indeed was actually happening while reading the data through a specific pointer in the code. To avoid the restarts of the computer physically, vmware software is used. This provides restarting only the corresponding operating
system without restarting the computer physically. “It provides a completely virtualized set of hardware to the guest operating system” [19]. Ubuntu 8.04 linux machine has been used as the guest operating system with a kernel 2.6.29.

The software was developed first as a payload dependent property. Then, it is tested as a payload independent measurement tool. This has been accomplished by getting benefit from the already developed features of GRE tunneling module. It provides the GRE header to be manipulated by the transmitting side of the tunnel, in which the timestamp and the sequence number is appended before transmission. The GRE module brings flexibility in the usage of the tunnel as it extends the GRE header.

Testing the code needed to be also done by establishing the GRE tunnel on two separate linux boxes with a kernel 2.6.29. Remotely connecting to the linux boxes via ssh were becoming sometimes problematic in terms of lack of status of the physical interfaces of the remote Linux boxes. Establishing a tunnel among devices where they are behind a NAT is hard, therefore testing was done among two linux boxes located at the same subnetwork. The configurations of the Ethernet adaptor of Vmware tools were fixed in order to provide the public address for the linux boxes that are running as guest operating systems.

One other problem was the calculation of timings among unsynchronized machines since the delays would not be accurate. This problem is solved by setting the initial values for both the departure and receiver times to time zero, and carry out the measurements depending on the synchronized time values.

### 7.2 Future Work

In the second payload independent measurement tool, the accuracy decreases as the delay jitter value that is set on the NETEM tool increases. The values for the dropped packets can not be put into the analysis and thus results in obtaining non accurate results.

The thesis can also be extended and featured into a two way delay jitter measurement module. Now with this implementation, the measurement is being done only on the receiving module. This is suitable in streaming services that sends the information mainly in one direction on a regular basis, i.e., periodically [2]. However in cases like sending information on both ways like in interactive services, the sender might also need to know about the status of the traffic on the network;in other words inside the GRE tunnel. For this case, the two-way delay jitter measurement is needed. This
will be possible by using the transmitting and receiving methods on both ends of the tunnel as shown below.

The departure and the receiving times of the modules will be examined at the same module and will be used in the calculations. This will of course increase the utilization inside the GRE tunneling by creating an extra traffic on the reverse direction. The modifications and routing should be configured on the ipgre_tunnel_xmit module to forward the selected data, i.e., timestamps and headers, of received packets back to the beginning of the tunnel which is denoted by the green color T on the left side of the above picture.
Chapter 8
APPENDICES

8.1 Appendix A.

8.1.1 Validation of the delay jitter measurements (Payload independent approach)

The measurements that are taken from measurement module are compared to the measurements taken from the TCPDUMP and calculated by Excel. On all of the tables below, the 4th column has the Excel result while the 5th column consists of the measurement module output. All the results are in milliseconds level. Time intervals are set to 3 seconds.

<table>
<thead>
<tr>
<th>Time Interval = 3 seconds</th>
<th>DT</th>
<th>RT</th>
<th>DT-RT</th>
<th>STDEV</th>
<th>MM STDEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>202</td>
<td>217</td>
<td>15</td>
<td>17.45</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>403</td>
<td>413</td>
<td>10</td>
<td>5.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>660</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>847</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>1024</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1208</td>
<td>1245</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1409</td>
<td>1428</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1610</td>
<td>1666</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1812</td>
<td>1855</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>2035</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2214</td>
<td>2251</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2415</td>
<td>2438</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2616</td>
<td>2616</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2818</td>
<td>2829</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time Interval = 3 seconds</th>
<th>DT</th>
<th>RT</th>
<th>DT-RT</th>
<th>STDEV</th>
<th>MM STDEV</th>
</tr>
</thead>
<tbody>
<tr>
<td>205</td>
<td>236</td>
<td>31</td>
<td>13.62</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>406</td>
<td>451</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>608</td>
<td>627</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>827</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>1006</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1211</td>
<td>1219</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1413</td>
<td>1454</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1614</td>
<td>1599</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1815</td>
<td>1823</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2016</td>
<td>2030</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2218</td>
<td>2248</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2419</td>
<td>2453</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2620</td>
<td>2640</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2821</td>
<td>2823</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Validation of measurements (TCPDUMP+Excel vs Measurement module)
Table 8: Validation of measurements (TCPDUMP+Excel vs measurement module)

<table>
<thead>
<tr>
<th>Time 1</th>
<th>Time 2</th>
<th>Time 3</th>
<th>Time 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>202</td>
<td>258</td>
<td>18.84</td>
<td>18</td>
</tr>
<tr>
<td>403</td>
<td>420</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>605</td>
<td>638</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>806</td>
<td>855</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>1054</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>1209</td>
<td>1246</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>1410</td>
<td>1454</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>1611</td>
<td>1612</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1813</td>
<td>1862</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>2056</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>2215</td>
<td>2264</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>2416</td>
<td>2439</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>2618</td>
<td>2635</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>2824</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>242</td>
<td>41</td>
<td>13.77</td>
</tr>
<tr>
<td>403</td>
<td>391</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>599</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>806</td>
<td>828</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>1007</td>
<td>1045</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>1208</td>
<td>1232</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1410</td>
<td>1416</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1611</td>
<td>1603</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1812</td>
<td>1799</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>2015</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2215</td>
<td>2212</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2416</td>
<td>2433</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>2618</td>
<td>2634</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td>2828</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>191</td>
<td>10</td>
<td>10.3</td>
</tr>
<tr>
<td>403</td>
<td>438</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>592</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>799</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>1015</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1208</td>
<td>1216</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1409</td>
<td>1422</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>1610</td>
<td>1623</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>1811</td>
<td>1822</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>2040</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>2214</td>
<td>2251</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>2415</td>
<td>2401</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2616</td>
<td>2616</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td>2804</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time 1</th>
<th>Time 2</th>
<th>Time 3</th>
<th>Time 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>194</td>
<td>7</td>
<td>9.1</td>
</tr>
<tr>
<td>403</td>
<td>385</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>575</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>761</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1007</td>
<td>1023</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>1208</td>
<td>1201</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1409</td>
<td>1417</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1610</td>
<td>1629</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>1812</td>
<td>1788</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>2017</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2214</td>
<td>2225</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2415</td>
<td>2392</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>2616</td>
<td>2563</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>2818</td>
<td>2796</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>3019</td>
<td>2997</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>197</td>
<td>4</td>
<td>8.59</td>
</tr>
<tr>
<td>402</td>
<td>428</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>599</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>763</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>1011</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1207</td>
<td>1232</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>1409</td>
<td>1432</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>1610</td>
<td>1624</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1811</td>
<td>1785</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>2006</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2213</td>
<td>2222</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2415</td>
<td>2433</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>2616</td>
<td>2626</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2817</td>
<td>2823</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>214</td>
<td>13</td>
<td>16.9</td>
</tr>
<tr>
<td>403</td>
<td>413</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td>637</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>851</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>1024</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>1208</td>
<td>1212</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1409</td>
<td>1426</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>1610</td>
<td>1655</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>1811</td>
<td>1801</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>2021</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2214</td>
<td>2248</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>2415</td>
<td>2414</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2617</td>
<td>2619</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2818</td>
<td>2864</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>

Implementation Of Measurement Module For Seamless Vertical Handover

95
8.1.2 GRE Setup

Bash Script : GRE.sh

Outlet of the GRE Tunnel

Setting Up the GRE Tunnel
insmod measurement.ko
ip tunnel add measurement mode gre local 194.47.148.199 remote 194.47.148.95 ttl 255 seq key 666
ip link set measurement up
ip addr add 10.1.1.1 dev measurement
ip route add 10.2.2.0/24 dev measurement

Turning Down the GRE Tunnel
ip link set measurement down
ip tunnel del measurement
rmmod measurement .ko

Inlet of the GRE Tunnel:

Setting Up the GRE Tunnel
insmod measurement.ko
ip tunnel add measurement mode gre local 194.47.148.95 remote 194.47.148.199 ttl 255 seq key 666
ip link set measurement up
ip addr add 10.2.2.1 dev measurement
ip route add 10.1.1.0/24 dev measurement

Turning Down the GRE Tunnel
ip link set measurement down
ip tunnel del measurement
rmmod measurement .ko

8.1.3 Makefile

ifeq ($(KERNELRELEASE),)
    obj-m := measurement.o ip_gre.o
else
    KERNELDIR $?= /lib/modules/2.6.29-generic/build
    PWD := $(shell pwd)
endif

default:
    $(MAKE) -C $(KERNELDIR) -I/usr/include/ M=$(PWD) modules

clean:
tar:
    n=`basename `pwd``; cd ..; tar cvf -$n | gzip > $n.tar.gz
8.1.4 Used Linux Kernel Commands

dmesg (display message): is a command that shows the kernel messages in kernel buffer.

insmod: is used to insert the kernel module to the kernel

rmmod: is used to remove the kernel module from kernel

ssh: means secure shell. It provides the connection with the local and remote device in a cryptographic way.

scp: means secure copy. It is used to securely transfer a file between two hosts.

J*/.exe: this command is used to run an application in exe format in linux shell.

Make: Is a tool that controls the generation of executables from the program's source files.

Makefile: Is a special file that provides instructions of how to build the executable file to the make command. It provides to use to make command to build or install the program.

Gcc: It's a packet of the GNU project that consists of C language compiler. It's used in the following format to create the exe files from a source file tester.c. (gcc tester.c -o tester.exe)

tcpdump: is a packet analyzer commandline-based tool that allows the user to display TCP/IP and other packets being transmitted or received over a network to which the computer is attached.

ip -s tunnel: is used to observe the status of the GRE tunnel created between two ends of the tunnel. The received and transmitted number of the packets can be tracked by the relevant number changes. It's also possible to detect the errors and loops through the tunnel

route -n : is used to observe the routes of all the routes attached to each interface that exist on the computer.

8.1.5 Used Netem Commands

# tc qdisc change dev eth0 root netem delay 100ms 10ms
# tc qdisc change dev eth0 root netem delay 100ms 20ms distribution normal
# tc qdisc change dev eth0 root netem loss 0.1%

8.1.6 FPING Commands

fping 10.1.1.1 -c 100 -p 100 5000

Implementation Of Measurement Module For Seamless Vertical Handover

97
Sends 100 packets with 100ms inter packet delay to destination IP, 10.1.1.1.

8.1.7 Code Snippet Regarding the Calculation of Delay Jitter

```c
static void calculate_jitter_new(unsigned long thissstime,unsigned long thatime,unsigned long ul_seqnooo,
unsigned long exseqno,unsigned long thissecond, unsigned long thatseconds)
{
    unsigned long timeee;
    //printk("INITIAL CONTROL SEQNO: %u.",ul_seqnooo);
    if(ul_seqnooo==0)
    {
        //printk("TIMES RESET");
        initial_time_dept=thatime;
        initial_time_recv=thissstime;
    }
    printk("the seq no: %u.",ul_seqnooo);
    //printk("thistime: %u.",thissstime);
    //printk("thattime: %u.",thatime);
    //printk("thatsecond: %u.",thatseconds);printk("thissecond: %u.",thissecond);
    //printk("initial_time_dept %u: ",initial_time_dept);
    //printk("initial_time_recv %u: " ,initial_time_recv);

    if(thatime>initial_time_dept&&thissstime>initial_time_recv)
    {
        synhed_recvtime=thissstime-initial_time_recv;
        synhed_deptime=thatime-initial_time_dept;
    }
    printk("synhed_recvtime: %u.",synhed_recvtime);printk("synhed_deptime: %u.",synhed_deptime);
    //printk("Expected Sequence Number: %u. ",exp_seqno);
    if(ul_seqnooo>1)
    {
        if(exp_seqno==ul_seqnooo&&reorder_flag==true&&after_four==false)
        {
            printk("Expected Sequence Number: %u. ",exp_seqno);
            differences_received_times = synhed_recvtime-recv_times_buffer;
            differences_dept_times = synhed_deptime-dept_times_buffer;
            if(differences_received_times>differences_dept_times)
```

Implementation Of Measurement Module For Seamless Vertical Handover
(timeee=differences_received_times-differences_dept_times;)
if(differences_received_times<differences_dept_times)
(timeee=differences_dept_times-differences_received_times;)
timeee=differences_received_times; //added to test METHOD 2
printk("synhed_recvtime: \%u.",synhed_recvtime);
printk("synhed_deptime: \%u.",synhed_deptime);
printk("differences: \%u.",timeee);
stdeviation(timeee,ul_seqnooo);
timeee=synhed_recvtime_old-recev_times_buffer;
printk("differences: \%u.",timeee);
stdeviation(timeee,ul_seqnooo);
after_four=true;
exp_seqno+=1;
reorder_flag=false;
synhed_recvtime_old=synhed_recvtime;
}
if(exp_seqno>ul_seqnooo&&reorder_flag==true&&after_four==false)
{
printk("Expected Sequence Number: \%u. ",exp_seqno);
differences_received_times=synhed_recvtime-synhed_recvtime_old;
differences_dept_times=synhed_deptime-synhed_deptime_old;
printk("synhed_recvtime: \%u.",synhed_recvtime);
printk("synhed_deptime: \%u.",synhed_deptime);
if(differences_received_times>differences_dept_times)
(timeee=differences_received_times- differences_dept_times;)
if(differences_received_times<differences_dept_times)
(timeee=differences_dept_times-differences_received_times;)
reorder_flag==true;
timeee=differences_received_times;
printk("differences: \%u.",timeee);
stdeviation(timeee,ul_seqnooo);
synhed_recvtimes_old=synhed_recvtime;
exper_seqno+=1;
}
if(exp_seqno<ul_seqnooo&&reorder_flag==false)
{
printk("Expected Sequence Number: \%u. ",exp_seqno);
reorder_flag=true;

Implementation Of Measurement Module For Seamless Vertical Handover
dept_times_buffer = synhed_deptime;
recv_times_buffer = synhed_recvtime;
printk("synhed_recvtime: \%u.",synhed_recvtime);
printk("synhed_deptime: \%u.",synhed_deptime);
exp_seqno++=1;
}
if(exp_seqno==ul_seqno000&&reorder_flag==false)
{
differences_received_times=synhed_recvtime-synhed_recvtime_old;
differences_dept_times=synhed_deptime-synhed_deptime_old;
printk("differences_received_times: \%u.",differences_received_times);
//printk("differences_dept_times: \%u.",differences_dept_times);
printk("Expected Sequence Number: \%u. ",exp_seqno);
if(differences_received_times>differences_dept_times)
{timeee=differences_received_times- differences_dept_times;}
if(differences_received_times<differences_dept_times)
{timeee=differences_dept_times-differences_received_times;}
timeee=differences_received_times ; //added to test METHOD 2 // comment it out //08 OCT

printk("differences: \%u.",timeee);
stadeviation(timeee,ul_seqno000);

synhed_recvtime_old=synhed_recvtime;
synhed_deptime_old=synhed_deptime;
exp_seqno=ul_seqno000+1;
reorder_flag=false;}}

Implementation Of Measurement Module For Seamless Vertical Handover

100
8.1.8 GRE Header Structure:

<table>
<thead>
<tr>
<th>Bits 0–4</th>
<th>5–7</th>
<th>8–12</th>
<th>13–15</th>
<th>16–31</th>
</tr>
</thead>
<tbody>
<tr>
<td>C R K S</td>
<td>Recur</td>
<td>Flags</td>
<td>Version</td>
<td>Protocol Type</td>
</tr>
<tr>
<td>Checksum (optional)</td>
<td>Offset (optional)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key (optional)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number (optional)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routing (optional)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The detailed description of some of the above GRE packet fields are as follows:

**Checksum Present (C), 1-bit**

This is the Checksum field. The Checksum and Offset fields are present if either the Checksum Present bit or the Routing Present bit are set.

**Routing Present (R), 1-bit**

If R is set then the Offset field is present, also the Checksum and Offset fields are present if either the Checksum Present bit or the Routing Present bit are set.

**Key Present (K), 1-bit**

If K is set then the Key field is present.

**Sequence Number present (capital S), 1-bit**

If S is set then the Sequence Number field is present.

**Protocol, 16 bits**

Contains the protocol type of the payload packet. In general, the value will be the Ethernet protocol type field for the packet. Additional values may be defined in other documents.

**Checksum, 16 bits**

Contains the IP (one's complement) checksum of the GRE header and the payload packet.

**Key, 32 bits**

The encapsulator inserts a number to the Key field and this number is used for identification of a traffic inside the tunnel.

**Sequence Number, 32 bits**

The encapsulator inserts a sequence number to this field of packet's header. It is used to track the
ordered and reordered packets through the tunnel by examining these numbers at the decapsulator (other end) of the GRE tunnel at the receiver side.

8.1.9 GRE Tunneling Configuration Steps Under Linux

- **Kernel Support**
  The compiled kernel module for GRE tunneling should be loaded to kernel on the Linux machine with the following command
  
  ◦ `insmod ip_gre.ko`

- **Establishing the tunnel device**
  The tunnel device establishment is done with the below command line instruction.
  The ip tunnel tool is used during the setup of the GRE tunnel among the peers. During the testings, loopback configuration is also configured by changing the remote IP address to the loopback ip address:127.0.0.1
  
  ◦ `ip tunnel add measurement mode gre remote 194.47.148.169 local 194.47.148.99 ttl 255`

- **Designation of an IP address to the tunnel device**
  A virtual ip address for a remote device to be connected is given as 10.2.2.1 with a peer IP address where measurement is the name of the tunnel device.
  
  ◦ `ip address add dev measurement 10.1.1.1 peer 10.2.2.1`

- **Designation of the routing definitions**
  Turn the links up.
  
  ◦ `ip link set measurement up`

- **It is necessary to include the 10.2.2.0/24 network to the tunnel device with the following command line.**
  
  ◦ `ip addr add 10.2.2.0/24 dev measurement`.

- **When it is desired to remove the tunneling on the interface, the following command lines are used.**
  
  ◦ `ip link set measurement down`

  ◦ `ip tunnel del measurement`
8.1.10 More Payload Dependent Approach Results

Some data and graphs regarding the results of the payload dependent part is shown as the following.

1) **Verification of The Ring Buffer Method** to test whether it gives stationary results in payload dependent measurement module.

```
Packet Delay vs Packet Sequence Number
```

```
Packet Delay vs Packet Sequence Number
```

*Illustration 18: Ring Buffer stability test*
2) Verification of storing the packet parameters in a long array buffer in payload dependent measurement module to test its stability.

Packet Delay vs Packet Sequence Number

<table>
<thead>
<tr>
<th>Packet Seq. No.</th>
<th>Variation of Delay among packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>72</td>
</tr>
<tr>
<td>2</td>
<td>76</td>
</tr>
<tr>
<td>3</td>
<td>174</td>
</tr>
<tr>
<td>4</td>
<td>79</td>
</tr>
<tr>
<td>5</td>
<td>178</td>
</tr>
<tr>
<td>6</td>
<td>141</td>
</tr>
<tr>
<td>7</td>
<td>78</td>
</tr>
<tr>
<td>8</td>
<td>118</td>
</tr>
<tr>
<td>9</td>
<td>79</td>
</tr>
<tr>
<td>10</td>
<td>125</td>
</tr>
<tr>
<td>11</td>
<td>77</td>
</tr>
</tbody>
</table>
Packet Delay vs Packet Sequence Number

Calculated Mean: 184 microseconds
Calculated Standard Deviation: 289 microseconds

Implemented Mean: 194 microseconds
Implemented Standard Deviation: 287 microseconds
Packet Delay vs Packet Sequence Number

Calculated Mean: 98 microseconds
Calculated Standard Deviation: 28 microseconds

Implemented Mean: 99 microseconds
Implemented Standard Deviation: 29 microseconds
3) Verification of the same method to test DJ calculation accuracy for reordered packet sequence on payload dependent measurement module

Calculated Mean: 875 microseconds  
Implemented Mean: 874 microseconds

Implemented Standard Deviation: 2264 microseconds  
Calculated Standard Deviation: 2386 microseconds
8.1.11 More Payload Independent Approach Results

Below five graphs are the delay jitter results observed from measurement module in series of tests by using the 6th formula in Chapter 2.7.1

![NETEM is set to 20 ms](image)

![NETEM is set to 40 ms](image)
NETEM is set to 50 ms

NETEM is set to 125 ms

NETEM is set to 175 ms

Implementation Of Measurement Module For Seamless Vertical Handover
8.2 APPENDIX. B

References


