CHALMERS

Accurate Delay Measurements and Quality Assurance in IP Networks

Master of Science Thesis in Communication Engineering and Integrated Electronic System Design

Anders Berggren & Lukas Garberg

Department of Signals and Systems
Communication Systems Group
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2011
Master's Thesis 2011:XX

Accurate Delay Measurements and Quality Assurance of IP Networks

ANDERS BERGGREN
LUKAS GARBERG

Examiner: Erik Agrell

Master's Thesis 2011:XX
Chalmers University of Technology
Department of Signals and Systems
Communication Systems Group
SE-412 96 Göteborg
Sweden
Tel. +46-(0)31 772 1000

Department of Signals and Systems
Göteborg, Sweden, 2011

Abstract

In this thesis the implementation and analysis of a highly accurate delay (round trip time) and delay variation measurement system for *Internet Protocol* (IP) networks is described. The system delivers an accuracy better than 1 μs, and typically as low as 10 to 150 ns depending on configuration, using off-the-shelf Linux computers and network adapters with IEEE 1588 support. The evaluation of commercial products, Linux's timestamp *application programming interfaces* (APIs) and generally available Ethernet controllers resulted in a `ping`-like application designed to exploit the Intel 82580 controller to it is fullest, patches for the Linux kernel that among other things enables timestamping of IPv6 packets, and a complete, easy-to-use, self-configuring service level agreement (SLA) system for Tele2 AB.

The way that Linux's `SO_TIMESTAMPING` and the Ethernet controller's IEEE 1588 support is implemented doesn't allow for transmit (TX) timestamps to be embedded in sent packets, and thus existing protocols such as *Two-Way Active Measurement Protocol* (TWAMP) could not be used. Instead, a protocol using *User Datagram Protocol* (UDP) for measurement packets (ping/pong) and *Transmission Control Protocol* (TCP) for transfering timestamps from reflector (server) to initiator (client) was designed. The difficulties of accurate time synchronization resulted in the application only supporting two-way delay measurements for the time being.

Targetting rapid deployment and low maintenance, the measurement application `probed` was built into an appliance based on Debian GNU/Linux. It is running on low-cost Intel Atom servers, booting of a read-only USB stick with an simplified administration console. The measurement node appliances are automatically configured, for example in a full-mesh topology, from a central administration system and aggregates measurement data making it readily available for Tele2's statistics collection systems.

Based on requirements and requests, the system differentiates itself by providing properties in areas where available products were found lacking. Apart from the mentioned accuracy advantage, the system (among other things) verifies DSCP values of returned pongs, allows for high packet rates, performs continuos measurements with no gaps in-between (even during re-configuration) and dynamically adjusts the aggregation resolution in order to store and display more detailed data during transient conditions and network anomalies.

Keywords: IP, measurement, IEEE 1588, SLA, quality, delay, delay variation, jitter, round-trip time, latency.

Contents

Acknowledgements

We would like to express our gratitude to the Transport Network group at Tele2 for the great time we had, and for giving us access to their inter-continental IP network. A special thanks goes to Kristian Larsson who initiated this project and has provided us with the initial requirements, valuable ideas and indispensable feedback.

John Ronciak, Patrick Ohly, Jeff Kirsher and the Ethernet controller team at Intel has been very helpful in understanding the timestamp infrastructure, and let us contribute with patches.

We also like to thank Richard Cochran for being patient with us, trying to get started with Linux's APIs and understanding kernel timestamps.

Zacharias El Banna and others at Juniper Networks were very forthcoming and helpful while getting to know the RPM on EX4200 switches.

Finally, Sui Yutao and Erik Agrell at Chalmers has done a great job organizing and coaching in all academic matters.

Chapter 1

Introduction

This projects overall aim was to *provide Tele2*[1] *with a modern, accurate quality assurance and reporting system*. The following sections will explain what is being measured and motivate the development of a new system.

1.1 Prerequisites

Computer communication is a fairly complex topic, even when considering basic scenarios. The communication model is layered according to the *Open Systems Interconnection* (ISO) model[2], which describes how to encapsulate protocols and technologies into each other. For example, "IP over Ethernet over TP" is the process of sending *Internet Protocol* (IP)[34] packets in several Ethernet frames as electronic signals in physical *twisted pair* (TP) cables. This happens to be the by far most common technology for local networks, which is what most people get in touch with. A good understanding of these three layers (network, data link and transport) is recommended in order to fully comprehend this thesis.

The networks built by carriers and *telecommunication operators* (TELCOs) differs from local networks in many aspects, in spite of both being based on IP. They transmit data over vast distances, and do in fact constitute the main building blocks of the Internet. Consequently their design and technology poses an even greater challenge in order to be understood. Information about such technologies will be explained within the thesis, to a reasonable extent.

There are several code listings in the thesis. Some of them contain programming code, while others list commands executed from a computer's shell (such as Unix's `sh/bash` or Microsoft Windows' `cmd.exe`). Command listings always start with the character "$" for commands that may be executed as any low-privilege user, and "#" for commands that has to be executed as a super-user

[1] Tele2 AB is major European telecommunication operator

[2] Specified by the *International Organization for Standardization* in the 1980's, inspired by ARPANET which came to compose the Internet.

("root" in Unix-like operating systems such as Linux). The command listing for starting a program named `probed` as a super-user would be:

```
# probed
```

1.2 Background

In a *packet switched* (PS) network such as the Internet there are no dedicated circuits between sender and receiver; thus no strict timing for metrics such as delay (latency). Common derivates of delay are jitter (variation in delay) and loss (infinite delay). The traditional method for measuring delay is by sending a probe from an *initiator*[3] to a *reflector*[4], who responds in order for the initiator to calculate the *round trip time* (RTT). Almost every IP-compliant device[5] has *Internet Control Message Protocol* (ICMP) `ping`[33] command built-in, and is thus capable of being initiator or reflector for such a measurement. The RTT is the two-way delay, similar to how distance is measured with light and mirrors. The most striking difference however, is that IP packets are *routed*[6] and thus possibly traversing the network along asymmetrical paths. Consequently, the relation between the one-way delay o and the RTT r is not always $o = \frac{r}{2}$.

Devices supporting one-way delay measurements are scarce, and typically implements more sophisticated protocols such as *One-Way Active Measurement Protocol* (OWAMP). One reason for not being an established protocol in stock devices, is that accurate time synchronization is needed in order to provide accurate results. IP does not provide time synchronization, and although some underlying data link protocols might in fact do so, that information is lost whenever IP traverses diverse data link networks. Time synchronization is instead implemented on top of IP, for example using *Network Time Protocol* (NTP)[24], which degrades its accuracy. Recall that IP is in OSI layer 3, whereas NTP is an application layer protocol, and consequently layer 7. Methods for performing accurate one-way measurements will be presented in chapter 2.

1.2.1 Motivation

The necessity to accurately quantify communication network quality has emerged from many areas, such as verification of *Service License Agreements* (SLAs), system planning, and work related to performance optimizations. Also, Internetwork engineering and management, i.e., routing and transmission network utilization optimization, directly depends on the ability to acquire traffic metrics such as delay, delay variation, loss and bandwidth utilization[14].

Delay-derived measurements have become increasingly important with the

[3]The initiator, also referred to as client, makes the initial request
[4]The reflector, also referred to as server, responds according to a defined protocol
[5]Equipment with IP connectivity, such as a computer, server, router, smart-phone, etc.
[6]Transmitted in several steps, or hops, from router to router on its way to the destination

popularity of modern applications such as *Voice over IP* (VoIP) and carrier's[7] migration from *circuit switched* (CS) networks to *packet switched* (PS) networks such as Ethernet. The real-time requirements of CS technologies such as *Synchronous Digital Hierarchy* (SDH) and *Plesiochronous Digital Hierarchy* (PDH) makes them very sensitive to inconsistencies in network quality if transported over a PS network. Tele2's ongoing work in this area is the major motivation for this project, which aims at producing and documenting a system for highly accurate (μs), continuous delay measurements, suitable for large-scale global deployments, with the results being analyzed at a central location. In practice, the aim is to accurately measure the delay variation of Tele2's core IP network, which will transport CS data using *Circuit Emulation Service over Packet Switched Network* (CESoPSN) equipment[39]. In order for these devices to tolerate delay variation, they have a configurable jitter buffer. The larger buffer, the more delay variation (jitter) they can accept. The buffer does however introduce a fixed delay, which of course should be kept as short as possible. Therefore, knowing the jitter is of crucial importance while operating a CESoPSN network.

1.2.2 Previous works

The topic of Internet/IP network quality assurance has been well studied. Highly notable is the work of the *IP Performance Metrics* (IPPM) working group of the *Internet Engineering Task Force* (IETF), which has written a number of *Request For Comments* (RFC)[8] on the topic. Its early works regards finding metrics for describing the different phenomena which arises in IP networks, but later works include for example *"Two-Way Active Measurement Protocol"*, TWAMP [17], which is a protocol for measurement of two-way metrics between network elements.

The *Surveyor* project carried out in the late 1990s used about 50 measurements probes, standard PCs with attached GPS clock synchronization cards, which performed active measurements of Internet performance [20]. Later, Sprint launched their passive IPMON system, which gathers real network traffic with highly accurate timestamps from backbone links at a number of Sprint's core sites [15]. The data is then transfered to a computing cluster which derives measurement values from the packet dumps.

Later on, similarities between *Surveyor* and the system that this project evolved into will appear.

[7] Also referred to as *Internet Service Providers* (ISPs); of which the company Tele2 AB that sponsors this work is one example

[8] The documents released by IETF containing protocol specifications, research and other things regarding the Internet.

1.3 Objective

The primary objective is to investigate methods for quality assurance of IP networks and develop a system which provides Tele2 with accurate measurement data. Major effort has been put into making the system simple to operate, both when it comes to administration and giving the user access to the data needed. An accuracy better than 1 ms, and if possible in the range of microseconds, was initially requested.

The system will consist of measurement equipment placed at key locations within the network. A central system will at a regular interval collect data from the measurement equipment, store it and make it accessible to the user.

1.4 Scope

The project is limited to providing measurement data for the core network. Although, the theory presented as well as the measurement practice developed is highly applicable in the fringe of the network as well. The emphasis is on active measurements, where measurement probes are injected into the network.

A theoretic background will be provided, but to make sense to the reader a good understanding of IP networking is required.

1.5 Outline

Initially, a theoretic background regarding Internet quality measurements is presented in chapter 2, where metrics such as packet loss, delay and delay variation are introduced (section 2.2). A number of issues with clocks and sampling are then presented. This chapter is largely based on the work of the IETF working group of IPPM.

Then, in chapter 3, the evaluation of measurement systems is described. Requirements are set based on especially demanding applications in section 3.1 and different pieces of measurement equipment are evaluated in section 3.2. Section 4 contains an extensive description of the system developed, which is dubbed SLANG.

Chapter 5 concludes the thesis body, with results, discussion summary and some ideas about what further work on the topic might result in.

Chapter 2

Measuring the Internet

How is the Internet measured? There is a multitude of concepts, metrics and methods available. The general focus, however, is on quality and will thus be emphasized in this thesis. This chapter first covers some measurement concepts and continue with a list of different network metrics. Lastly, practical issues regarding sampling and time, which turn up while performing measurements, are presented.

2.1 Internet measurement concepts

Before digging into metrics and measurement methods, an introduction to a few concepts is in place. First, the distinction between active and passive measurements will be made and then some pros and cons of one-way and two-way measurements will be presented.

2.1.1 Active versus passive measurements

An active measurement is performed by injecting measurement probes into the network being measured. The schoolbook example is the `ping` command available on most computer platforms which transmits an ICMP echo request to a remote system and measures the time required for the remote system response to return.

The injected measurement traffic does naturally affect the network which might impact the validity of the measurement and network itself. However, as the focus is on core networks where the available bandwidth generally is measured in tens of gigabits per second, the impact of a measurement packet stream of (example values, valid to an order of magnitude) ten 512 byte packets per second on a 10 Gbit/s link is negligible. As $\frac{10\times512\times8}{10\times10^9} = 4.1 \times 10^{-6}$, a few ten thousands of a percent of the available capacity, hundreds of concurrent measurement sessions could be run in parallel without affecting the network enough to be worth mentioning.

Instead of inserting traffic into the network being measured the already present, real-world traffic, can be used to gather measurement data. An example of this is the IPMON system deployed by Sprint[15].

2.1.2 One-way versus two-way measurements

One clear distinction that can be made between *active* measurements is one-way and two-way measurements. The primary motivation for one-way measurements is that Internet paths[1] often are asymmetrical, that is the path taken by packets from source to destination is not the same as the path from destination to source. The two paths can have drastically different properties and can perfectly well traverse different Internet service providers' networks. Other things that affect the measurement result differently in either direction is different traffic conditions and differing *Quality of Service* (QoS) configuration. [32]

One-way measurements are complicated by the requirement of a low clock offset between the initiator and the responder. When a packets transmit (at time T_1) and receive (at time T_2) and the timestamps are compared to find the packet delay $D = T_2 - T_1$, it is clear that the accuracy of measurement depends largely on the clock synchronization. In the two-way case, on the other hand, the clock offsets cancel each other out giving a value which does not depend on the offset. The clock's impact on the accuracy of measurement is reduced to the clock skew (see section 2.4) between the two packets and timestamp accuracy. The impact of clock skew during these short time intervals is generally small, unless the clock is adjusted more severely by for example an NTP service.

In favor of the two-way measurements if the ability to performed certain measurements without deploying a specific responder host by using for example ICMP echo or *Transmission Control Protocol* (TCP) as described in [21].

2.2 Metrics for Internet quality

Of utmost importance while performing measurements are the metrics used to express the measured phenomena. The *Internet Engineering Task Force* (IETF) *IP Performance Metrics* (IPPM) working group has defined a set of metrics for Internet performance measurements which are suitable also for quality measurements. Also the *International Telecommunication Union* (ITU) has defined a set of metrics [2], but as the emphasis is on IP networks the bulk of this section is based on the works of IPPM.

2.2.1 General concepts

Before going deeper into specific metrics a few words about general concepts are in place.

[1] A sequence of hosts and links describing one way through the network. A path is defined as being unidirectional. [32]

Where applicable, this part tries to adhere to the foundation laid out in [32] when it comes to for example vocabulary. Although definitions of concepts such as hosts and paths can be found there, a few will be explained in detail below.

An important concept introduced in [32] is the notion of *"Packet of type P"* as the measured value of a metric might vary with what type of packet is being looked upon. For example, a packet marked with a *Differentiated Services Code Point* (DSCP) prioritized higher in the network may have a lower packet loss probability and packet delay than a differently marked packet, a packet with erroneous IP header checksum might be discarded along the way, as would a packet with a too low TTL.

Due to this, the notion of a "packet of type P" is introduced as a generic packet type used to describe measurement traffic. In certain cases the packet type is more strictly defined, and thus gives us the ability to talk about *"generic IP-type-P-connectivity or more specific IP-port-HTTP-connectivity"* [32].

Additionally, *"a separation between three distinct – yet related – notions of metrics"* are introduced in [32], that of the *singleton*, *sample* and *statistical* metric.

The *singleton* metric is the smallest metric of the subject available, for example the two-way packet delay for one single packet. From these atoms the *sample* metric can be derived. Sample metrics are collections of singleton metrics, for example the two-way packet delay for packets transmitted at a rate of 100 *packets per second* (pps) for five minutes. From a sample can then a *statistic* metric be calculated by calculating some statistic from the sample of singleton metrics, for example the mean or median of the 30000 two-way packet delay metrics in the sample from the previous example.

We also make a distinction between *basic* and *derived* metrics, where basic metrics are directly measured (as one-way delay) and derived metrics are calculated from basic metrics (such as one-way delay variation calculated from two one-way delay measurements).

2.2.2 Packet loss

As one of the most basic IP performance metrics, the *packet loss* serves as an important foundation for IP performance measurements. As IP does not guarantee successful delivery of a datagram most software has some robustness to packet loss, but what ever technique utilized the performance will sooner or later start to suffer as the packet loss increases.

IPPM has defined a singleton metric for one-way packet loss as:

> >>The *Type-P-One-way-Packet-Loss* from P to P at T is 0<< means that S sent the first bit of a Type-P packet to Dst at wire-time* T and that Dst received that packet.
>
> >>The *Type-P-One-way-Packet-Loss* from Src to Dst at T is 1<< means that Src sent the first bit of a type-P packet to Dst at wire-time T and that Dst did not receive that packet. [4]

A practical issue is to differentiate between a very long delay and a lost packet. Putting an upper bound to the time waited for a response is necessary for this purpose. If packet loss for a specific application is being measured, the behavior that specific application should be taken into account when determining the actual time out value, as different applications handle high delay differently. In [21] a value of 10 seconds is recommended for general connectivity measurements.

Corrupted packets which still arrive at the destination are regarded as lost with the motivation that it can not safely be determined that the source and destination hosts are correct in the case of a corrupted IP header or that the packet really belongs to a specific test stream in the case of a corrupted datagram payload.

2.2.3 Packet delay

The packet delay is a measure of how long time it takes to transfer a packet from source to destination. A number of factors such as cable length, bit rate and network congestion affects the delay, which therefore is split into distinct parts; transmission delay, propagation delay and routing delay .

- *Transmission delay* is the time needed to output all bits on the wire which makes it dependent on the bit rate used. The transmission delay D_T is given by $D_T = N/R$ where D_T is given in seconds, N is the number of bits to transfer and R is the bit rate in bits per second.

- *Propagation delay* is the time from the first bit leaving the source host until it reaches the destination node, depends on the cable length and propagation speed in the cable medium. $D_P = d/s$ where D_p is the propagation delay in seconds, d the distance in meters and s the propagation speed in meters per second. For CAT 5 cable the propagation speed is about $0.64c$ where c denotes the speed of light in vacuum[41].

- *Routing delay* is the delay induced by routers along the link and is defined as the time between the first bit of a packet reaches the input interface until the first bit leaves the output interface. It is in turn divided into three parts; *queuing delay* during which packets spend in queues waiting to be processed, *processing delay* during which the packet is inspected and the packet action is determined (for example route lookups) and, lastly, *additional delay* which denotes all other types of delay induced by routing equipment.

When regarding packet delay, two distinctions needs to be made: that of the one-way and two-way packet delay.

2.2.3.1 One-way delay

Why measure one-way delay? On the Internet asymmetric paths between two hosts are commonplace, that is the traffic between the hosts does not necessarily utilize the same path in both directions. Even though the paths are symmetric,

the traffic load conditions can differ between the directions, as can the network QoS configuration.

In [3] the singleton *"Type-P-One-way-Delay"* is defined as follows:

> For a real number dT, >>the *Type-P-One-way-Delay* from Src to Dst at T is dT<< means that Src sent the first bit of a Type-P packet to Dst at wire-time* T and that Dst received the last bit of that packet at wire-time T+dT.
>
> >>The *Type-P-One-way-Delay* from Src to Dst at T is undefined (informally, infinite)<< means that Src sent the first bit of a Type-P packet to Dst at wire-time T and that Dst did not receive that packet. [3]

Important to note is that the metric denotes the time needed to *transfer the entire packet*, which means that the measured value will be dependent on the packet size.

To perform a one-way delay measurement the general procedure is to form a test packet of type P at the initiator and transmitting it to the reflector. As this is done, a timestamp will be taken when the packet is transmitted and received, when the packet is as close to the wire as possible. These timestamps can then be compared to obtain the type-p-one-way-delay. Here two problems with one-way measurements arises; the accuracy of the measurement is bound by how closely the clocks at the source and destination host are synchronized. Also their resolution and skew together with how close to the wire the timestamps can be taken impacts the accuracy of measurement.

2.2.3.2 Two-way delay

The two-way delay or *round-trip time* (RTT) is what you obtain from the well-known `ping` program. As stated in the previous section, the two-way delay in comparison to the one-way lacks information needed to determine in what direction the delay is seen.

IPPM specifies a two-way (round-trip) delay metric in [5] where the singleton metric is defined as:

> For a real number dT, >>the *Type-P-Round-trip-Delay* from Src to Dst at T is dT<< means that Src sent the first bit of a Type-P packet to Dst at wire-time* T, that Dst received that packet, then immediately sent a Type-P packet back to Src, and that Src received the last bit of that packet at wire-time T+dT.
>
> >>The *Type-P-Round-trip-Delay* from Src to Dst at T is undefined (informally, infinite)<< means that Src sent the first bit of a Type-P packet to Dst at wire-time T and that (either Dst did not receive the packet, Dst did not send a Type-P packet in response, or) Src did not receive that response packet.

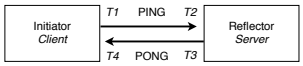

Figure 2.1: Two-way delay measurement with four timestamps

> >>The *Type-P-Round-trip-Delay between Src and Dst at T<< means either the *Type-P-Round-trip-Delay from Src to Dst at T or the *Type-P-Round-trip-Delay from Dst to Src at T. When this notion is used, it is understood to be specifically ambiguous which host acts as Src and which as Dst. [5]

Also in this case, the measured value is dependent on the packet size.

The general procedure for performing a measurement is for the initiator host to craft a type-P packet and transmit it to the reflector host while taking a timestamp when the packet is as close to the wire as possible. The reflector returns the packet as soon as possible and another timestamp is taken as the packet returns to the initiator. By comparing these two timestamps the two-way delay can be estimated.

In contrast to the one-way case, the accuracy of measurement is not dependent on the synchronization between the clocks of the initiator and responder system as the error is cancelled out when the subtraction is performed. Instead, it is the clock resolution and skew that limits the accuracy of measurement and consequently it is easier to get a highly accurate two-way measurement than a one-way.

Another source of error that so far has not been addressed is the time the packet spends at the responder, what in [5] is called the *host time*. In *Network Time Protocol (NTP)* [24], for each delay measurement four timestamps T_1, T_2, T_3, T_4 are taken when the packet exit the initiator host, enter the responder, exit the responder and enters the initiator host once again respectively. See figure 2.1.

By using this scheme the two-way delay can be calculated as in equation 2.1 with less impact of host time.

$$D = (T_4 - T_1) - (T_3 - T_2) \tag{2.1}$$

Instead, the main concern is how close to the wire the timestamp can be taken.

2.2.4 Packet delay variation (jitter)

For streaming applications like VoIP the packet delay variation is of great importance. As the data is presented to a user in a constant stream, problem arises when the arrival of packets is not regular as the receiving application can be missing data to present. To mitigate this problem, a play-out or jitter buffer is usually used, which simply delays the stream for some time to allow for non-regularities. When it comes to quality measurements the delay variation

plays an important role as it can be used as a measure of transient network congestion in comparison to packet loss which may indicate long-term congestion. The main source of variation in delay is variation in queue depth of the routers (part of the *routing delay* mentioned in section 2.2.3) traversed along the path. Packet delay variation is often referred to as jitter, but as this word has other meanings as well the term packet delay variation will be used to an as large extent as possible [9].

In the literature there are a couple of different definitions of packet delay variation. IPPM has defined the term *IP packet delay variation* (ipdv) as

> Type-P-One-way-ipdv is defined for two packets from Src to Dst selected by the selection function F, as the difference between the value of the type-P-One-way-delay from Src to Dst at T2 and the value of the type-P-One-Way-Delay from Src to Dst at T1. T1 is the wire- time at which Scr sent the first bit of the first packet, and T2 is the wire-time at which Src sent the first bit of the second packet. [9]

In appendix A.3.1 of [19] the jitter is defined as *"the absolute value of the difference between the arrival times of two adjacent packets minus their departure times"*, $|(T_{2j} - T_{1j}) - (T_{2i} - T_{1i})|$ where T_2 and T_1 represents the arrival and departure times of packets i and j where $j > i$.

Another approach is given for the jitter estimation used in *Real-time transport protocol* (RTP) [37]. Paired with each RTP data stream a control stream using the *RTP Control Protocol* (RTCP) is set up, which provides feedback on the quality of the transmission. The RTP inter-arrival jitter J is defined to be *"the mean deviation (smoothed absolute value) of the difference D in packet spacing at the receiver compared to the sender for a pair of packets"* [37]. The packet spacing difference between source and destination D is the same as the packet delay variation between two consecutive packets as can be seen in formula 2.2.

$$D(i,j) = (T_{1j} - T_{1i}) - (T_{2j} - T_{2i}) = (T_{1j} - T_{2j}) - (T_{1i} - T_{2j}) \tag{2.2}$$

From the D the inter-arrival jitter J is derived as

$$J(i) = J(i-1) + \frac{(|D(i,i-1)| - J(i-1))}{16} \tag{2.3}$$

This value is calculated continuously for each received packet. As previous packets are take into account, each calculated value provides information not only about the current network state, but the state over a (short) time period.

2.2.5 Packet reordering

IP lacks a mechanism to ensure in-order packet delivery. Due to the nature of packet-switched networks packets belonging to a stream can take different routes through the network, routes which may vary in transmission time. Other reasons

packet reordering occur are lower layer protocols correcting errors or equipment utilizing parallel processing of network traffic [26]. IPPM has defined packet reordering as:

> The value of Type-P-Reordered is defined as TRUE if s < NextExp (the packet is reordered). In this case, the NextExp value does not change.
>
> The value of Type-P-Reordered is defined as FALSE if s >= NextExp (the packet is in-order). In this case, NextExp is set to s+1 for comparison with the next packet to arrive. [26]

To determine whether packets are delivered in-order a strictly increasing sequence number s identifying what number in the order a specific packet has is needed. Then, according to [26] an incoming packet is said to be reordered if its sequence number is lower than what was expected. In all other cases the packet is considered to be in order. This is in [31] described as emphasizing late rather than premature arrivals, which according to the same source gives about 25% higher numbers.

It might also be of interest to quantify the *magnitude* of packet reordering, what in [26] is dubbed the *reordering extent*. Put in words, the reordering extent is distance, measured in packets, between the reordered packet and the earliest received packet with a higher sequence number.

2.2.6 Media delivery index

The *Media Delivery Index* (MDI) is a derived metric provided as an effort to quantify the quality of streaming transmission over IP. It is intended to provide a metric useful when monitoring the impact of IP network dynamics on a large number of simultaneous streams in a production network [40]. As stated in [40], *"It is believed that the MDI provides the necessary information to detect all network-induced impairments for streaming video or voice-over-IP applications."*.

The MDI is expressed as

$$\text{DF} : \text{MLR} \tag{2.4}$$

where the *Delay Factor* (DF) is a measure of the maximum difference between the arrival of media data and the drain of media data over an interval. For a media service using a fixed bit rate this is the drain rate while the arrival rate is the actual rate which data arrives at the destination. For example, a 64 kbit/s *Pulse Code Modulation* (PCM) voice stream has a drain rate of 64 kbit/s. The DF-value is then the maximum number of milliseconds required to fill up the missing (or drain the present surplus) data in a buffer at the receiving side at the drain rate during a specific interval. The interval is typically one second for higher bit rate streams (1 mbit/s and upwards) [40]. The delay factor gives a hint about how large the receiver buffer needs to be to handle the present packet delay variation.

The *Media Loss Rate* (MLR) is on the other hand a measure of lost data during the transmission. The value is defined as the number of lost or reordered *flow* packets during a time interval, where flow packets is not necessarily equal to the number of IP packets (for example, it is common to carry seven 188-byte *Moving Picture Experts Group Transport Stream* (MPEG TS) flow packets in one IP packet [40]).

2.3 Sampling issues

To study variances such as delay variation it is required to take a sample (see section 2.2.1) of singleton values. This can be done in a number of ways with respect to where and when the samples are taken. This will decide what phenomena the measurement system will be able to detect. For example, if sampling packets before reaching a specific host the host's impact will be missing, the same goes for not sampling during a specific time interval.

It is also highly desirable that the sample is unbiased, *"that the process of collecting the measurements in the sample did not skew the sample so that it no longer accurately reflects the metric's variations and consistencies"* [32]. Examples might be when many measurement probes are transmitted at the same time which might congest the network, or sampling at regular intervals which might hide periodic network behavior.

Further, a non-predictable sampling interval makes it more difficult for a network operator to temporarily modify the network to perform more favorable at the time measurements are performed [32].

A common sampling scheme is to sample at a fixed time interval, *periodic sampling*. It is generally simple to perform, but exhibits the problem of potentially showing only part of metrics having a periodic behavior as mentioned above. It is also highly predictable. However, the periodic behavior can be attractive for performing measurements mimicking a specific protocol, such as the 50 Hz packet rate of G.711 u-law-encoded audio over RTP [13].

According to [32] a better way to perform sampling is what they call *"random additive sampling"*, where samples are separated by independently selected random intervals with some distribution $G(t)$. By choosing different $G(t)$ different sampling quality is obtained. With a good selection of distribution, an unbiased sample can be obtained.

By letting $G(t)$ have an exponential distribution with rate λ

$$G(t) = 1 - e^{-\lambda t} \tag{2.5}$$

so called *Poisson sampling* is obtained, a sampling method which has all the desired properties (non-biased, non-predictable, minimal effect on the network under test) [32].

2.4 Clock issues

The accuracy of many of the mentioned measurement types is directly linked to how accurately the current time can be determined. Speaking of clock issues, some definitions are in place.

- The *synchronization* between two clocks specifies how well two clocks agree on what the current time is. The actual difference between the two clocks is the *offset*.
- The *resolution* of a clock tells us the smallest time increase the clock can perform. The resolution does not include how often the clock actually is updated, neither how high *precision* the time is presented with. Also worth mentioning is that some system clocks make sure not to report the same value twice by adding a small value to the time if it is a duplicate of a previous time. Neither this smaller, artificial time increase is part of the clock resolution.
- A clock's *accuracy* is how well the clock agrees with *Coordinated Universal Time* (UTC)[2].
- The clock *skew* is a measure of the frequency difference between the clock and true time, that is the rate of change (first derivative) of offset between the clock and true time. Also the rate of change of the skew is of interest as real clocks do not show a constant drift due to for example temperature changes. This second derivative of the offset is called clock *drift* [23].

Depending on the type of measurement performed the clock accuracy has different impact on the measurement result. As stated in two-way delay measurements are not affected by a large clock offset as the errors are canceled out, compared to the one-way case where the clock offset is a major source to inaccuracy.

There are a number of methods to maintain a level of clock accuracy in computer systems.

2.4.1 GPS

The *Global Positioning System* (GPS), is a widely known system for acquiring the current location. A set of satellites, currently 31 of them [44], continuously transmit time data which receivers can use to deduce the current position by calculating the time the signals from each satellite needs to propagate to the receiver. In order for this to be possible, the receivers clock is accurately synchronized to atomic clocks in the GPS satellites which in turn regularly are synchronized to UTC. Thanks to this the GPS system can be used to synchronize clocks to UTC with a sub-microsecond accuracy. [12].

[2]Coordinated Universal Time is a standard time used worldwide to regulate clock and time.[42]

2.4.2 NTP

The *Network Time Protocol* (NTP), is a protocol used to synchronize clocks over IP networks [23]. NTP is built up by a hierarchy of servers where the hierachy level gives the distance to the reference clock. The hierarchy level is with NTP nomenclature called the stratum level, where the top level servers are assigned stratum zero and the level increases with one for each level you descend. Stratum zero devices are the reference clock themselves such as atomic clocks, GPS receivers or radio clocks. An NTP server with a lower stratum level is generally more accurate than the servers at higher levels.

An NTP clients periodically transmits packets to their servers, which the server respond to with time data. Using packet timestamps the client over time can estimate the latency and jitter over the path to the server and use this information to increase the accuracy. The NTP client also calculates an estimated accuracy of the clock synchronization. [23]

2.4.3 PTP

To achieve higher accuracy the *Precision Time Protocol* (PTP) was developed by IEEE and released as IEEE 1588. Two versions exist: the older IEEE 1588-2002 and the newer non-compatible IEEE 1588-2008 which adds some functionality. The high accuracy mainly stems from the use of specialized hardware which performs highly accurate timestamping when PTP packets are sent and transmitted, enabling highly accurate delay measurements.

The initial objective of PTP was to provide a method for highly accurate (sub-microsecond) clock synchronization in local networks. PTP clocks are organized in a master-slave hierarchy according to information conveyed in multi-cast sync messages. Each slave synchronizes its clock to its master by exchanging a number of packets at a slightly higher rate then NTP; max 1 pps for IEEE 1588-2002 and 10 pps for IEEE 1588-2008. A clock can function as slave on one subnet and master on another, to create chains of clocks. A set of clocks synchronizing to one another form a *domain*, and each domain selects a grand master which is placed on top of the clock hierarchy. In a real-world scenario, a PTP-aware network switch or repeater can act as slave on one subnet where, for example, the grand master clock resides and as master on other subnets. The switch is then functioning as a *boundary clock* which segments the network for PTP and will not forward synchronization packets between the different subnets. The boundary clock also removes the variation in delay itself would induce to the forwarded PTP packets as the packets never pass through the device.[12]

IEEE 1588-2008 adds methods for acquiring even higher accuracy. As an alternative to boundary clocks it also adds *transparent clocks*, devices which instead of being part of the master-slave chain updates a time-interval field in the PTP packets with data which the receiving PTP device can use to compensate for the delay induced by the transparent clock[16]. As the IEEE 1588-2008 requires changes to the packet format, it is not backwards compatible with IEEE 1588-2002.

Chapter 3

Evaluation of measurement products

When this project was planned, it was assumed that commercially available measurement nodes would fulfill the requirements stated. In this section, an evaluation of the preferred systems is detailed. Many more products than mentioned were considered, but out-ruled for different reasons such as cost, support or specifications.

3.1 Requirements

The requirements which have been formed for the measurement system are naturally closely connected to the applications which the network is being used for. Therefore, the most demanding applications has been gathered and the requirements on resolution and accuracy is therefore derived from their respective requirements.

3.1.1 CESoPSN

Circuit Emulation Service over Packet Switched Network (CESoPSN) *"is a method for encapsulating structured (NxDS0) Time Division Multiplexed (TDM) signals as pseudo-wires over packet- switching networks (PSNs)"* [39]. Traditional *public switched telephone networks* (PSTNs), are inherently circuit switched. As these networks were digitalized, the natural choice was to use TDM for multiplexing of multiple analog circuits into a digital one. In Europe, 30 circuits were bundled together with two control channels into a base connection type, the 2048kb/s E1 connection. As capacities and requirements grew, E1 connections were later on bundled into 155Mbit/s STM-1 connections, which were bundled into 622 Mbit/s STM-4 connections and so forth.

Due to the use of TDM, these transmission systems are highly timing-critical. Data for each multiplexed circuit needs to be available when its time slot is

reached, which translates to a steady stream of packets arriving at a constant, non-varying rate when the packet-switched data is decapsulated and transmitted on the TDM network.

To compensate for the high sensitivity in delay variation, CESoPSN equipment includes a jitter buffer [39]. The addition of the jitter buffer delays the transmission of outgoing TDM data a configurable time period to the incoming data from the packet switched network to let the delayed data act as a buffer for the occasion that the next packet arrives slightly late (due to variation in packet delay). If the buffer runs out of data, a buffer under-run occurs, the device is required to send replacement data. This can for example be data generated from a user-configured pattern or a repetition of the last sent frame[22]. No matter what is used as replacement data, this will lead to distortion of the voice signal or data corruption.

To make the system more tolerant for delay variation the jitter buffer can be increased, but as this inevitably leads to increased delay for the TDM data transmission, the buffer should be kept as small as possible.

At Tele2, an older SDH transmission system mostly carrying voice and leased lines runs in parallel with the IP network. However, there is a wish to phase the aging and expensive SDH system out in favor of the IP network. Tests has been made where is has been realized that at some instances however, the IP network does not fulfill the strict delay variation requirements of the CESoPSN system. Tele2 requires the jitter buffer to be only a few milliseconds long.

3.1.2 Voice over IP

With the introduction of *Voice over IP* (VoIP) techniques the network quality requirements increased dramatically as even short-term errors such as packet loss and delay variation can be noticed by the user.

To compensate for delay variation also VoIP systems utilize jitter buffers which as mentioned before introduce a delay. With up to 150 ms end-to-end delay including all delay sources ("mouth-to-ear") regarded as more or less transparent to the user and over 400 ms unacceptable [1], a buffer quite much larger than the one used for CESoPSN can be used given that the mean delay is low.

It is up to each VoIP codec to define how to handle lost packets and the packet rate, which also defines how much voice data, in seconds, each packet contains. The G.729 codec for example transmits around 20 ms of voice data per packet and simply replays the voice data from a previous packet when data is missing [8]. This is however only repeated once, which makes a loss of two or more consecutive packets detectable to the users. Tele2 wishes to detect all these potential errors, which sets a limit for the packet rate the measurement system needs to support.

3.1.3 Results

So what conclusions can be drawn regarding the requirements from the background given above? As the CESoPSN is highly sensitive to variation in delay

with the small jitter buffers requested, this is what will define our requirement in accuracy of measurement.

With a jitter buffer only a few milliseconds long a quite high accuracy is requested, certainly higher then the one millisecond currently provided. From these prerequisites, the requirement on accuracy of measurement is set to at least 100 μs.

As two consecutive packets lost in a VoIP-stream might be detectable to the user, the system needs to be able to detect a loss of connectivity only this long during for example reroutes due to a fiber breakage. As many VoIP streams as previously stated transmit at a rate of 50 Hz, the required measurement resolution is set to 50 packets per second.

3.2 Evaluation procedure

A key component of any measurement system is the equipment performing the actual measurements; in this case sending what in here is referred to as *measurement probes*. A few pieces of equipment has been evaluated according to the requirements specified in 3.1. Since the most important metric according to the requirements is delay variation, the primary method used for evaluation was to have the devices measure the delay variation of a reference connection known to induce low delay variation and comparing the results. This low delay variation connection was created by connecting the equipment under test with a twisted-pair or fiber cable directly.

3.2.1 Measurement setup

In addition to the delay variation accuracy estimation, measurements of other characteristics are verified by comparing them to a reference system from IXIA called 400T as it was available at Tele2. Unlike the measurement nodes that are evaluated, the IXIA system is both initiator and receiver in the same chassis, thus not having any time synchronization issues. Consequently, it provides a verified, high accuracy of 40 ns[1], but can't measure the delay between two distant locations. Therefore a *Multi-protocol Label Switching* (MPLS) tunnel was configured in Tele2's network which provided a real scenario Internet path, which terminates at the same location in both ends. By running the system under test and the IXIA in parallel, also the accuracy of realistic round-trip times could be verified. For the test setup, two of the 400T's SFP slots were fitted with LX fiber modules.

3.2.2 Juniper RPM

Juniper *"Real-time performance monitoring"* (RPM) is a service available in Juniper Networks' JUNOS operating system [27]. For higher accuracy, some de-

[1]Not verified; IXIA's technical documentation is the source for the 40 ns claim

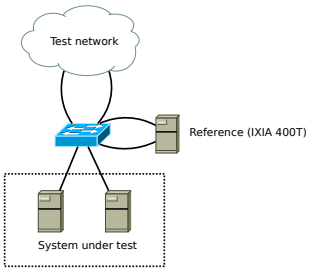

Figure 3.1: Measurement setup for testing real-world delays

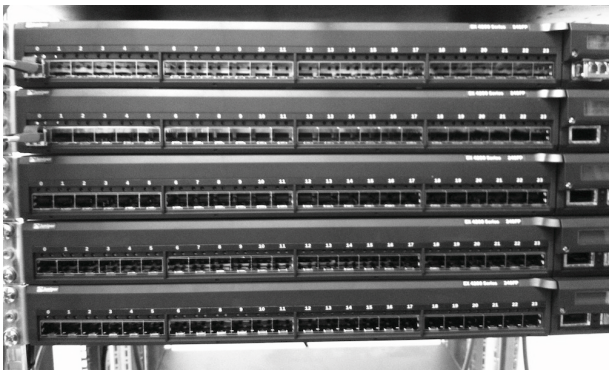

Figure 3.2: Juniper Networks EX4200 mounted in the lab

vices can perform packet timestamping in hardware, very close to the wire, both at the transmitting and receiving device. Thanks to this the (non-deterministic) time spent on handling packets in the operating system or queues on the device should not affect the measurement. Juniper's EX4200 layer 3 switches were the primary choice as measurement nodes, and three of these were acquired. In spite of being affordable, they are capable of performing the RPM's hardware timestamping. They were installed in Tele2's lab using their standard procedures, and equipped with industry-standard gigabit Ethernet fiber SFPs[2]. The standard procedure implies redundant 48V power installation performed by electricians, a choice of tested software images, and configuration templates. The switches are shown in figure 3.2. The RPM's delay variation accuracy was estimated by running the system over a short wire (in this case, 1 meter of single-mode fiber). Several different software versions were used, including Tele2's standard release, and the latest stable release from Juniper Networks.

Although the 95th percentile of the delay variation was within 200 μs, the max delay variation during the 2 hour test period peaked at 2500 μs which is well above what was requested. Another shortcoming is the maximum configurable

[2] *Small form-factor pluggable transceiver* (SFP) is a standardized form factor and interface for hot-pluggable transceivers used extensively in the telecommunication market.

Initiator	Reflector	Mean RTT	Max. delay variation
SLANG	SLANG	$\approx 1\mu s$	$< 1\ \mu s$
SLANG	RPM	$\approx 200\ \mu s$	$\approx 1200\ \mu s$
RPM	RPM	$\approx 400\ \mu s$	$\approx 2800\ \mu s$

Table 3.1: Evaluation of RPM using SLANG as test bench

packets per second (pps) rate of 1, which is slow enough to mask away some of the effects of IP networks, and also far from the 50 pps that is needed to emulate a typical VoIP phone call.

While developing a brand new Linux-based measurement system (dubbed "SLANG") as alternative (should commercial off-the-shelf systems prove unsuitable), RPM support was added to it. First of all, that would make SLANG able to use the Juniper switches as reflectors, which would be practical as such switches are widely deployed within Tele2's network. Secondly, it would make deeper analysis of the Juniper RPM possible. More information about the SLANG system is found in section 4. The RPM protocol was analyzed, using simple packet analysis tools such as `tcpdump` and Wireshark[3] and by *decoding*[4] the data in a Python script, acting as a simple initiator. The RPM code was then ported to `probed`, the daemon[5] of the SLANG system responsible of sending/receiving UDP probes (pings); in other words acting as initiator and reflector. The RPM code was never shipped in the final version of SLANG; but kept as reference. Three delay variation measurement tests were executed (as usual, over a short single-mode fiber) in order to determine the RPM's hardware timestamping characteristics. The results are shown in table 3.1 and suggests that RPM with hardware timestamping on EX4200 has an induced RTT of about 200 μs and a maximum delay variation of more than 1000 μs, irrespective of being used as initiator or reflector.

The tests were performed with JUNOS 10.0R1.8, the release that (out of the ones tested) gave the best results (lowest delay variation). Graphs for the RPM to RPM test is found in figure 3.3. Although the induced RTT does not present a problem in itself (it can be compensated for), the maximum delay variation is troublesome. Juniper Networks were very accommodating and supportive, and discussions with them confirmed and explained these results. However, measurement accuracy will not be improved in the EX4200 within a foreseeable future. As a consequence, the RPM with hardware timestamping on EX4200 switches were not chosen for high-accuracy delay variation measurements.

[3] A graphical network traffic inspection application; http://www.wireshark.org

[4] Splitting a C struct into list elements using the `unpack()` function of Python's struct package

[5] Daemon is the typical Unix/Linux term for an application running as a service in the background

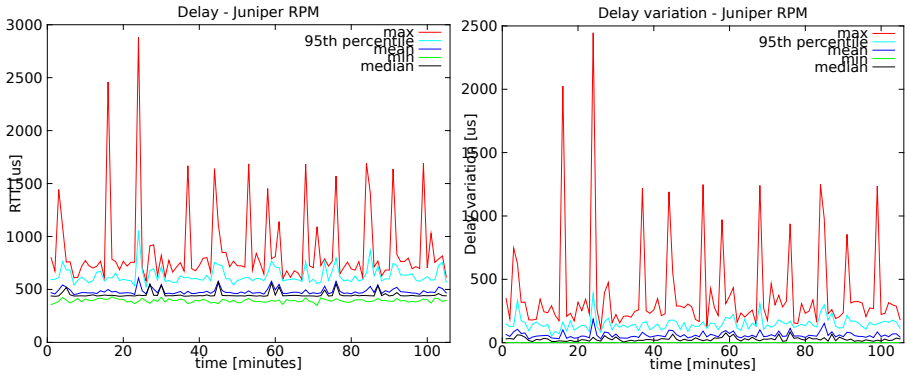

Figure 3.3: Juniper Network's EX4200 results

3.2.3 Cisco IP SLA

The system currently measuring Tele2's core/transport network consists of Cisco 1800-series access routers, configured solely as measurement nodes. The feature was initially named *Service Assurance Agent* (SAA) by Cisco, and later renamed to IP SLA. It's part of their *Internetwork Operating System* (IOS) software, with the primary function of sending probes[6] over various protocols for measuring delay and delay variation. The results are fetched using the *Simple Network Management Protocol* (SNMP) and displayed in line diagram graphs by a system developed at Tele2.

The primary reason for finding a complement to this system was accuracy and resolution requirements described in section 3.1. A typical delay variation graph for a measurement in the Tele2 core network displays a line fixed at either zero, one or two ms, which is explained by the Cisco IP SLA resolution of 1 ms. Secondary reasons for developing a complementary system includes, among the many other feature requests; automatic configuration through integration with Tele2 systems, validation of *Differentiated Services Code Point* (DSCP) flags, and the ability to run continuous tests which doesn't have to be (automatically) "restarted" with short delays in-between.

3.2.4 Prosilient Technologies

Prosilient Technologies develops a complete SLA supervision system used for monitoring core and mobile back-haul networks. They were invited to present their system, which uses a proprietary clock synchronization protocol to maintain a low clock offset between the measurement nodes which permits accurate (50μs [38]) measurements, even in one-way scenarios. In addition to the usual one-way and two-way delay and delay variation measurements their system also support an extensive list of other more specific higher level measurement types.

[6] A probe can for example be an ICMP echo request (ping)

As the Prosilient system is a complete solution, it is built to perform more or less autonomously, something which was shown when it came to integration with other systems. According to what could be derived from the presentation, the system would provide the level of auto-configuration which was requested and provide detailed measurement data from the nodes, but not at the granularity which was requested.

Partly due to these reasons, but mainly due to high cost, the Prosilient system was discarded from the list of alternative systems. No real-world testing of the system was performed.

Chapter 4

Developing SLANG, a Linux-based system

As a result of unsatisfactory performance of existing products, the project that this reports documents evolved into the development of a complete measurement system. The system was dubbed "SLANG" where "NG" initially stood for *Next Generation*, suggesting that it will be an upgrade compared to the current *Service Level Agreement* (SLA) systems currently deployed. "Slang" is also the Swedish word for hose. That however, is no abbreviation or symbol for anything.

4.1 Brief system overview

The project was divided into a few tasks, with defined interfaces. The tasks can be categorized using a layered scheme.

1. **Hardware** (Intel 82580). The lowest layer, is the hardware itself. Apart from a computer (in our case, a low cost Intel Atom server) the hardware most importantly consists of a network adapter with Intel's Ethernet controller chip 82580. The network card's function is, apart from receiving and sending packets, to provide highly accurate timestamps; the input data for delay and delay variation calculations.

2. **Operating system** (Linux). The next layer is the Debian operating system, a software distribution based on the Linux kernel. In order to reliably exploit the network card's timestamp functions, some modifications were made on the Linux kernel. These modifications will be detailed in the upcoming sections; for example section 4.2. To streamline usage and deployment of the system, the operating system was transformed into a small software image[1] that is easily written to a USB stick from which

[1] An image is in computer terms the raw binary data of something; commonly a medium such as a disk

the computer is booted[2], allowing for rapid deployment. The USB stick is read-only during normal operation, implying reliable operation. Following the startup, the administrator is presented with a user-friendly administration console, simplifying the minimal, yet required, configuration of SLANG.

3. **Measurement program** (`probed`). Similar to the commonly used computer program `ping` which sends an echo request (ping) to another computer, which responds with an echo reply (pong) while measuring the delay, a program called `probed` was developed. It acts as both ping client, and pong server, and uses Linux's timestamp API to accurately calculate the delay. The caveats of the timestamp API and 82580 hardware combined with the demanding functional requirements resulted in `probed` being a rather advanced, functionally complete and architecturally sophisticated program.

4. **Data collection and interface program** (`manager`). During normal, continuous monitoring operation, `probed` is configured by, and outputs the measurement data to, the `manager`. The manager stores the measurement information, calculates out-of-order and delay data, continuously aggregates data into statistics such as average and 95th percentile values, and makes everything readily available on an XML-RPC[3] interface. The `manager` accepts complex queries such as dynamic-granularity aggregated statistics, in order for data collection systems to automatically receive more detailed information for periods with transient characteristics. It also accepts configuration requests from a central management server.

5. **Central management** (SLANG control panel). Individual node configuration is a time-consuming and error-prone task. Therefore, a central management system was developed and integrated with Tele2's current node management and IP address management infrastructure (NILS). In that way, the entire cluster of SLANG nodes are automatically reconfigured upon changes such as the addition of new nodes. The control panel also displays a real-time view of the SLANG cluster, along with current measurement information and a 24-hour history view. The control panel does not collect statistics.

6. **Statistics system** (ASM). It's common practice to query network equipment for useful statistics (counters) over protocols such as *Simple Network Management Protocol* (SNMP). In the same way, SLA nodes are usually queried in regular intervals (for example 5 minutes) by a statistics system, that summarizes the data, and stores in some kind of database for

[2]Booting is the process of cold-starting a computer

[3]XML-RPC and its derivate SOAP are industry-standard protocols for *Remote Procedure Calls* (RPC) which has been increasingly popularized with the raise of the Internet. XML-RPC, hence the name, uses XML and encoding, and HTTP (the protocol of the World Wide Web) as transport.

a very long time. Such systems usually contains graphing capabilities; making the information available in a practical fashion. One of the most popular is Cacti[4], based on RRDtool[5]. Tele2 however, has their own statistic system called *Automated/AweSoMe Statistical Machine* (ASM) which was extended to interface with the manager's XML-RPC interface of the SLANG nodes.

During the remainder of this chapter, these components will be referenced and described.

4.2 Background

Since long, there has been an `SO_TIMESTAMP` option for sockets[6], instructing the Linux operating system's kernel (hereby referred to as just "Linux") to add timestamps to received packets (RX), in order for userland applications to more accurately know their receive time. The timestamp is generated in the Linux network code; not in hardware or in the driver. Since then, more sophisticated methods have evolved.

While researching IXIA's high-accuracy measurement platform, a Linux patch adding basic timestamping support for sent packets (TX) was found in the Linux mailing list. This coincidence gave birth to the idea of SLANG; and the development of that measurement system. Below are references to Linux's mailing list[7], giving the chronological background to the hardware timestamping that SLANG depends on.

On 29 Jul 2008, *Octavian Purdila of IXIA* sent "net: support for TX timestamps" and "net: support for hardware timestamping" to the mailing list to the mailing list[29, 28]. Although it was not ready for general usage at that time, it hinted that there might be possible to accurately measure delay with Linux.

On 21 Jan 2009, *Patrick Ohly of Intel* sent "net: new user space API for time stamping of incoming and outgoing packets" to the mailing list. In order to support *Precision Time Protocol* (PTP; IEEE 1588[8]) which depends on accurate timestamping, Intel added a more generic API for RX and TX timestamping to Linux. More specifically, the `SO_TIMESTAMPING` socket option, and `SIOCSHWTSTAMP` ioctl option. It was released in Linux 2.6.30, in June 2009, but was not ready for general usage. No general purpose network adapters able to timestamp arbitrary packets existed at that time, and working for Intel; Ohly used a prototype network adapter[30].

[4] Cacti is an open-source web-based data acquisition, storage and graphing system; see http://www.cacti.net

[5] RRDtool is a widely used, open-source, lightweight, data storage and graphing program suite by Tobias Oetiker; see http://oss.oetiker.ch/rrdtool/

[6] A `socket()`, used to create an endpoint for communication, is the de-facto API for network communication

[7] The mailing list is the primary development forum for Linux

[8] PTP is a highly accurate network time protocol for computers

On 12 Feb 2009, *Patrick Ohly of Intel* sent "igb: infrastructure for hardware time stamping" to the mailing list. It was an initial patch for hardware timestamping for Intel's network adapters.

On 19 Sep 2009, *Christopher Zimmermann* sent "SO_TIMESTAMPING fix and design decisions" to the mailing list. There are a couple of letters from Mr. Zimmermann, making various improvements to the API, which was partly broken at that time.

On March 2010 Intel released the 82580 Ethernet controller, and the first general-purpose network adapter to support RX and TX timestamping of all (arbitrary) packets.

On 4 Jul 2010, *Richard Cochran* sent "phylib: add a way to make PHY time stamps possible" to the mailing list. It was patches improving hardware (PHY, physical devices) timestamping, and also a Linux compilation configuration option `CONFIG_NETWORK_PHY_TIMESTAMPING` which is needed to enable it. More or less ready for general usage, and released 20 October with Linux 2.6.36.

Only weeks after Linux kernel 2.6.36 was released, it became apparent to the members of this project that an alternative solution to commercial measurement systems was needed. The timing couldn't have been better. During a few months, the SLANG system was realized, along with a few patches for Linux.

On 3 Feb 2011, *Anders Berggren and Lukas Garberg of this project* sent "[PATCH] fixing hw timestamping in igb" to the mailing list. Initially the SLANG project's `probed` application used Intel's `igb` driver (with Linux 2.6.36), since Linux's built-in driver had severe timestamp bugs (the time from the card's internal clock didn't reach user-land). Unfortunately, Intel's version had a few quirks as well, such as frequent RX timestamp for packets arriving too close to each other. Being a quite serious bug for this project, the problem was being analyzed and debugged. Linux's version was chosen for development, since it is the one that comes with most Linux distributions, and since Intel's version was not compliant with the latest Linux version (2.6.37) as of February. The patch makes it possible to activate hardware timestamps (solving the clock-source bug), and fortunately Linux's `igb` version didn't inhibit the RX timestamp bug. Consequently, all the code inside `probed` dealing with RX timestamp abnormalities was removed. The removed code was complicated, as it tried to mask not only missing, but rather *incorrect* RX timestamps. The patch was accepted.

On 23 Feb 2011, *Anders Berggren and Lukas Garberg of this project* sent "[PATCH] net: TX timestamps for IPv6 UDP packets" to the mailing list. Marcus D. Leech pointer out in November 2009 that TX timestamps doesn't work with IPv6. IPv6 deployment has accelerated as of the IPv4 address pool now actually running out, and therefore support for the new Internet protocol was a requirement for SLANG. Inspired by Mr. Leech's work, a patch was developed. Unfortunately, it was not implemented in the same fashion as for IPv4; since the required control structures was not present in the IPv6 "append" function. Therefore, the timestamp socket lookup was performed inside the actual "append" function. The patch was accepted.

4.3 Comparing hardware and kernel timestamps

Tests were performed that compared Linux's `ping` (figure 4.1), `probed` with kernel timestamps (figure 4.2), and `probed` with hardware timestamps (figure 4.3). The results of the hardware timestamp test with Intel 82580 controllers are many times better than anything else tested. The jitter of a short wire was reported as only 8 ns, which is also the resolution of the 82580's oscillator. Initial testing suggested that kernel-based timestamps (in the network adapter's driver) would be accurate enough according to the requirements. Therefore, it was implemented throughout in SLANG. It was however not chosen as preferred method in the final system; mainly because of two reasons:

1. The accuracy of the hardware timestamps are many orders of magnitudes higher, motivating the relatively low cost of such a network adapter.

2. The confidence level of the measurements are much better for hardware timestamps. In the same way that kernel-based timestamps are more accurate than user-land timestamps[9], there is a much higher theoretical probability of the kernel not being able to generate the timestamp within a reasonable time, compared to the hardware of the network adapter. Effects suspected to originate from CPU power saving features, the interrupt/poll characteristics of Linux's *New application programming interface* (NAPI) combined with the fact that modern network adapters may generate only one interrupt for several packets[30], and the inaccuracies of the computers clock were observed in the kernel timestamp results. See figure 4.2, where the maximum delay variation is about 250 μs, whereas the 95th percentile is at 20 μs. Although it might be possible to remedy or mask away most of these infrequent but very inaccurate timestamps, it was decided that it was not worth the work.

It's noteworthy that no drivers implement kernel TX timestamps[10] as of 2011, despite of being available for many years[29] and as simple as adding the line `skb_tx_timestamp(skb);` to the transmit function; pointed at by `.ndo_start_xmit`.

4.4 The operating system

Tele2 declared several requirements that would apply if a system was developed in-house rather than purchased. Many of those are listed in section 4.5.1. One that applies directly to the hardware and operating system concerns maintainability. Network equipment such as routers used in the telecommunication sector differs from common computing platforms in terms of providing defined piece of work they are supposed to perform very well; and their administration

[9] User-land processes are generally not guaranteed time slices by the kernel, and the execution path is longer and less predictable

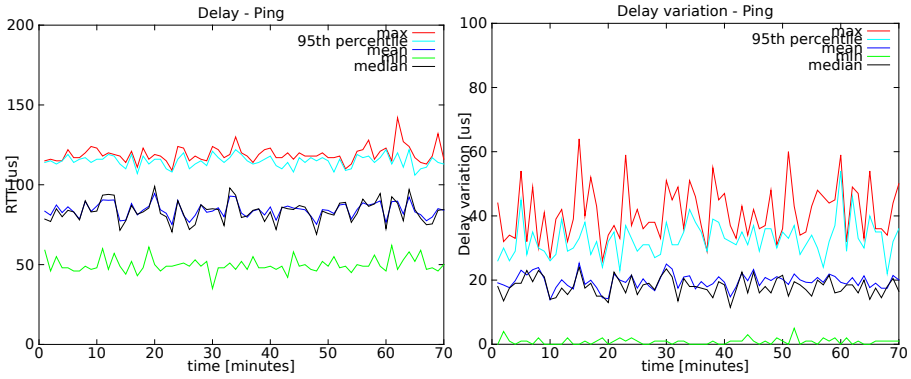

Figure 4.1: Linux's `ping` command results

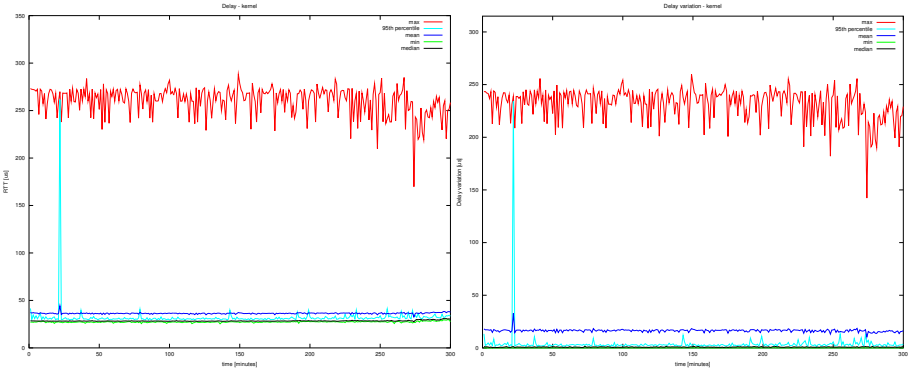

Figure 4.2: Linux's kernel timestamp results

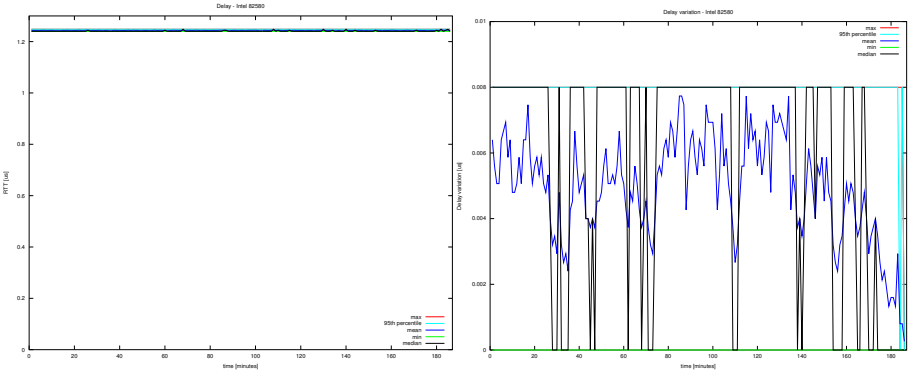

Figure 4.3: Linux's hardware timestamps with 82580 results

and maintenance characteristics are usually adjusted accordingly. Configuration interfaces are stable in terms of syntax, software releases are less frequent, and in cases of underlying operating systems[10] the complexity of those are normally hidden[11]. In the same fashion, Tele2 demanded that the system should not require Linux administration experience from field personnel, nor require any significant maintenance once deployed. First of all, it was decided that the SLANG nodes should contain as few moving parts as possible. When also considering the production of the nodes, it was decided that USB memory sticks should be used instead of hard-drives. USB sticks are easy to duplicate thanks to their USB interface, and simply writing the SLANG software image to the stick should be the only step in the installation procedure. One drawback of USB sticks is that they typically survive far fewer write operations than ordinary hard-drives, thus requiring the operating system to run with a read-only disk. All run-time data is stored on a so-called *Random-access memory* (RAM) file system which, hence the name, never stores the data on a persistent memory. The complexity of an entire Linux operating system is mostly hidden by presenting the administrators with an easy-to-use graphical configuration console only exposing the features that are relevant to an SLANG node.

The most common technique for running Linux on a read-only medium is to load the entire operating system off an image file onto a large RAM file system disk, or providing a so-called union file system that writes changes to RAM. Although having the advantage of the operating system being able to modify files that it expects to be writable, it has the disadvantage of requiring more RAM compared to simply stating that the entire disk is read-only. Since the SLANG nodes requires large amounts of RAM in order to make raw measurement data available for some time, and also in order to aggregate the data, it was decided that the SLANG nodes should use a pure read-only file system. The primary Linux operating system used at Tele2 is Debian[12], which, like most other Linux distributions, does not provide a read-only mode. An Internet guide[13] was used to get Debian onto the USB stick. In short, a partition was created with `fdisk` and `mkfs` which is mounted. Then, `debootstrap` is executed on the mount point. `chroot` to it. `apt-get` is used to get required packets such as kernel and `grub`; the boot loader. An `/etc/fstab` is written, with the disk referred by it is *Universally unique identifier* (UUID) in order to work irrespective of which BIOS drive number is assigned. Also `/boot/grub/grub.cfg` is configured with UUID as kernel "root" argument. In order to operate read-only, `/etc/mtab` is symbolically linked to `/proc/mounts` and important folder such as `/var/run` and `/tmp` is linked to a RAM disk mounted by adding `tmpfs <dir> tmpfs rw 0 0` to `/etc/fstab`. Once ready, unused space is cleared with `/dev/zero`

[10] Such as FreeBSD in the case of Juniper Networks' JUNOS, QNX for Cisco IOS-XR and Linux for VMware ESX (references are scarce)

[11] System administrators dealing with both servers (running for example Linux) and network equipment (such as routers or switches) most likely agrees with this statement

[12] Debian's web page is http://www.debian.org/

[13] Debian USB guide at http://www.func.nl/community/knowledgebase/how-run-debian-usb-stick

`> /zero; sync; sleep 2; sync; rm /zero` in order for the compression of the software image created with `dd if=/dev/disk1 bs=1048576 count=1024 | gzip > debian6.img.gz` to perform better. The system was left running for a few hours with the disk mounted writable, in order to determine which files had been changed, and possibly needed to be linked to the RAM disk.

The simplified user administration was written as a shell[14] script called `ui.sh` and uses the `dialog` command to draw the semi-graphical interface. This design paradigm allows for rapid development, but has the disadvantage of re-drawing the entire screen for each update; which is sub-optimal for slow terminals such as RS-232. The shell script interfaces with Debian's standard configuration files such as `/etc/network/interfaces` and `/etc/resolv.conf` in a non-destructive manner, in order for changes done manually in any of these files to be persistent.

4.5 The measurement program (`probed`)

The function of `probed` is similar to that of the commonly used `ping` program, available in all modern operating systems. Early in the design phase the `probed` program was even called `hwping`; suggesting a similar function to `ping`, but with increased accuracy thanks to hardware timestamps.

In the case of `ping`, a packet of type "echo request" (ping) is sent using the *Internet Control Message Protocol* (ICMP) protocol, recording the round-trip time until (if) a response of type "echo reply" (pong) is received. In order to match pings to pongs, sequence numbers are used. All modern operating systems responds to ping; meaning that they are in fact reflectors; responding with echo replies to echo requests. In a measurement context, the `ping` program is the initiator.

Although the function of `probed` could be roughly described in the same way, there are some notable differences. Most importantly, `probed` was designed to use more accurate timestamps for calculating the *round-trip time* (RTT). That implies that one either have to incorporate accurate timestamps into the operating system's ICMP reflector code, or develop a new (but similar) protocol. Since incorporating code into an operating system is a daunting task, and a ping/pong application is among the more simple one could design, the choice to develop a new protocol was obvious. In other words, an initiator (client) and a reflector (server) was to be developed. As it turned out; they were implemented in the same program; `probed`.

When calculating the RTT based on timestamps, there are four points in time involved. These are usually called T_1, T_2, T_3 and T_4, and depicted in figure 4.4. T_1 is the TX timestamp of the ping, sent from the initiator. T_2 is the RX timestamp of the ping, when received by the reflector. T_3 is the TX timestamp of the pong, sent from the reflector. T_4 is the RX timestamp of the pong, when received by the initiator. Since the $RTT = (T_4 - T_1) - (T_3 - T_2)$ it

[14] A shell is the standard terminal of a Unix-like operating system, and can also be used as an interpreter shell program code

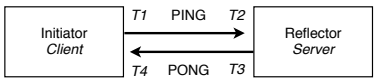

Figure 4.4: Timestamps for in ping/pong measurement

is apparent that the clocks of the initiator and reflector can be unsynchronized without affecting the RTT; what matters is *during how long time* the packet was *away from* the initiator and *spent inside* the reflector, respectively. This simple algorithm is found in the *Network Time Protocol* (NTP)[25] among others, and detailed in section 2.2.3.2.

4.5.1 Reviewing the requirements

Before discussing the design and implementation of `probed`, it is important to understand the requirements of the SLANG project, and the limitations of current hardware timestamp implementations. The requirements relevant to `probed` are:

1. An SLANG node has to be able to operate as both initiator and reflector, in order for flexible full-mesh[15] configurations.
2. An SLANG node has to be able to receive a configuration of the central management system (be reconfigured) without interrupting measurement operations.
3. Given the size of Tele2's network, that a full-mesh configuration has to be possible, and that each measurement should support at least 50 packets per second (pps) in order to simulate a typical voice call, an estimation of the total packet rate a node has to cope with suggests as much as 10,000 pps. While not being an absolute requirement, it is the performance target.
4. It should be possible to run the measurement program (`probed`) stand-alone; like `ping`. Therefore, it has to remember the state of sent packets, contain some kind of presentation mode, and user-friendly command-line arguments.
5. The measurement program should, using a best-effort scheme, try to determine if the ping or pong was lost in case of packet loss. One-way trip times are not a requirement since the routing in Tele2's core network is essentially symmetrical, but would be desirable.
6. The system should support *Internet Protocol version 6* (IPv6); the evolution (and eventually; replacement) of the Internet protocol.
7. It should be possible to configure and verify *Differentiated Services Code Point* (DSCP) for measurements.

[15] Full-mesh is the topology where all nodes have connections between each other

4.5.2 Input for design considerations

The nature of the timestamp API and the Ethernet controllers in question defines a number of properties and constraints that has to be taken into account during design and implementation.

1. Linux's timestamp implementation for both kernel and hardware timestamps (the `SO_TIMESTAMPING` socket option) provides the TX timestamps by making a copy of the sent packet available on the socket's error queue, with the TX timestamp being the RX timestamp of that packet. Therefore, every `sendto()` has to be followed by a `recvmsg()` loop trying to obtain the TX timestamp before sending anything else on the socket.

2. The timestamp support in generally available network adapters is scarce, and in those that support it, it is simply a consequence them being *Precision Time Protocol* (PTP, IEEE 1588) enabled. Some controllers that support PTP (for example Intel 82576) does not support per-packet timestamps; having only a few registers for the most recently received packets matching the structure of PTP packets. Others (for example Intel 82574) supports PTP but appear to be missing the essential timestamp infrastructure in Linux. The controller used in this project is called Intel 82580, and supports per-packet timestamps for any incoming (RX) packets. In practice, not all drivers that were tested implemented that well, resulting in unpredictable results. It does however provide the advantage of not having to generate fake PTP-like packets in order to get them timestamped.

3. The Intel 82580 controller have only one timestamp register for outgoing (TX) packets, and consequently it is essential that all transmission of packets that should be timestamps is done sequentially; otherwise packets might overwrite each other's timestamps.

4. In addition to the previously noted fact that Ethernet controllers usually doesn't support timestamping of arbitrary packets, that also applies to Linux. As noted in section 4.2, patches had to be developed in order to TX timestamp IPv6 packets. The fact that PTP uses *User Datagram Protocol* (UDP) as transport suggests that UDP packets are in fact timestamped. That can be verified by noting that the TX timestamp check `sock_tx_timestamp()` is in fact executed in Linux's `net/ipv4/udp.c`.

5. In order to calculate one-way trip times, the TX timestamp of the initiator (sender) has to be compared with the RX timestamp of the reflector (receiver). Therefore, the accuracy of that calculation will never be greater than the accuracy of their relative clock synchronization. Unfortunately, the accuracy of *Network Time Protocol* (NTP) maintains a time within 10 ms, and as good as tens of microseconds under ideal conditions[24]. That does not meet the requirements. PTP was designed to achieve greater accuracy, within the sub-microsecond range. There are however three

issues. The PTP accuracy greatly depends on the quality of the connection; and that is what is being measured. Secondly, as a consequence of the single TX timestamp register of the Intel 82580, `probed` and some PTP program cannot be running simultaneously. The most real problem however, is that there were no stable, working PTP implementation for Linux at the time. Highly accurate GPS clocks could have been suitable for one-way trip time calculations, if it wasn't for the fact the data centers usually have very bad reception of aerial signals. Another alternative could have been the time synchronization mechanisms in *Wide area network* (WAN) protocols such as *Synchronous Optical Networking* (SONET) and *Synchronous Digital Hierarchy* (SDH). Unfortunately IP does not provide time synchronization, and although these underlying WAN data link protocols in fact do so, that information is lost whenever IP traverses diverse data link networks. Simply put, that clock synchronization is never propagated to LAN equipment such as the SLANG nodes, which are Ethernet devices. Finally, it is noteworthy that the internal clock of the computer was found to introduce inaccuracies that were not negligible (in the range of 1 μs) when comparing measurements using the 82580's oscillator directly (`SOF_TIMESTAMPING_RAW_HARDWARE`) compared to transforming the oscillator-generated timestamps into system (wall) time (`SOF_TIMESTAMPING_SYS_HARDWARE`) which would have been the case when using NTP, PTP or GPS time synchronization. One-way trip times were not implemented in the final version of the system.

4.5.3 Proposing and evaluating designs

When analyzing the requirements and constraints together with other aspects of the project such as the participants programming language preferences, the following outline was formed:

1. Given that one SLANG node should be able to operate as initiator (client) and responder (server) at the same time, and that transmits with `sendto()` has to be performed (system) synchronously in order to get TX timestamps, the client and server were implemented in the same program; `probed`. That is rather unusual; most network applications have one server program, and clients are multiple instances of a client program.

2. As a consequence of UDP being used as transport, and the synchronous `sendto()` limitation, only one socket will be used; for all sessions, both client and server. The typical client/server approach is to use the stateful, connection-orientated *Transmission Control Protocol* (TCP), which enables the main server process to `accept()` a client into a new socket, which can be handled in a separate thread or process `fork()`. Additionally, one typically tries to separate the client and the server. When handling multiple clients within one process of execution, it is important that all operations are non-blocking. Otherwise, one client could stall the server, and thus denying all other clients meanwhile. An example of a blocking

function is `recv()`, which normally blocks until data is received. One way of having multiple threads or `fork()` processes handle individual clients on a UDP socket would be to use the `MSG_PEEK` receive option, and only receive those messages destined for the particular client process. That would however cause many unnecessary `recvmsg()` calls, which doesn't scale well. Also, the `sendto()` still have to be performed synchronously (because of the TX timestamps), which would require something like locking or message passing, adding to the complexity. The final version of `probed` handles the UDP socket in one single process, which is entirely multiplexed with `select()` waiting for pings, pongs and timestamps, and sending pings in defined intervals limited by the `select()` timeout.

3. The IPv6 requirement implies that so-called *dual-stack support* is needed. That is, supporting both IPv4 and IPv6 simultaneously. Because of the "one socket design" that was proposed above, it is fortunate that the "IPv4-mapped IPv6 addresses"[18] exists, and is supported by Linux (controlled by the `IPV6_V6ONLY` socket option). It allows both protocols to be operated on the same socket, with IPv6 native, and IPv4 being represented as `::ffff:X.X.X.X`. The program then has to perform all socket operations for both IPv4 and IPv6, such as enabling both `IP_RECVTOS` and `IPV6_RECVTCLASS` in order to fully support DSCP. When browsing packet headers with `CMSG_NXTHDR()` types in both level `IPPROTO_IP` and `IPPROTO_IPV6` have to be considered.

4. Since TX timestamps are received on the socket's error queue after transmission, the timestamp can't be embedded into the sent packet. This represent a problem for T_3, the TX timestamp of the pong. See figure 4.4. Somehow, the timestamp has to reach the initiator, in order for it to calculate the RTT. Sending an additional UDP packet with the T_3 (and possibly T_2) timestamp would be straightforward to implement, but introducing a very real risk of "timestamp loss". That is, the pong was received, but no RTT could be calculated. In cases of disturbances such as packet loss, the timestamp loss would be even greater, which is very unfortunate. TCP on the other hand, is a transport protocol offering connection-orientated reliable communication with features such as retransmission, and was chosen for timestamp deliver instead of trying to implement reliability onto UDP. Introducing another protocol for timestamp delivery inevitably adds to the complexity. One goal during the design was to maintain the simplicity of the code. The reflector (server) code is entirely stateless; simply responding to pings. The initiator (client) code is somewhat more complex, but yet contained within the main, and only, process of execution. Fortunately, a TCP server can be easily implemented without spawning new processes with `fork()`; only keeping track of the highest client file descriptor as returned by `accept()` and matching client addresses to file descriptors using `getpeername()`. Therefore, it was decided that the reflector part of the program should act as TCP server, remaining stateless. When a UDP ping arrives to the reflector, it is RX timestamp is recorded,

a UDP pong is sent to that client, it is TX timestamp is recorded, and the timestamps are sent to the client address by finding it is file descriptor using `getpeername()`. It would have made sense to have the reflector act as a TCP client instead; as that would have effectively eliminated race conditions between the client's UDP ping and TCP `connect()`. Also, the reflector is the timestamp sender, and would fully appreciate the state of the TCP connection (regarding re-connects, etc.). Now, although a TCP client is possible to implement in a non-blocking fashion as well, implementing that into the reflector would dramatically increase the size of it is state machine; keeping track of the state of all TCP connections for all client sessions. Further, the introduction of *network address translation*[16] (NAT)[11] has consequently led to a best-practice of "not connecting back to the client from the server" for client/server applications. Old protocols not adhering to this best-practice such as the *file transfer protocol* (FTP)[35] have difficulties traversing NAT networks, which has resulted in new protocol revisions[6]. It would not make sense to design a new protocol, incompatible with NAT. Therefore, it was decided that the initiator (client) should contain the TCP client. It was however not compelling to handle the many states of a TCP connection, for all client sessions (`probed` supports multiple initiator sessions), within the non-blocking, single-process initiator code. By moving the TCP clients to their own processes using `fork()`, they can operate in the blocking fashion that TCP's `connect()` was designed for. By doing so, it was possible to avoid multiplexing between the TCP clients (all clients have their own process) nor introduction of blocking function calls into the non-blocking main process. During program start-up, the initiators' TCP clients `connect()` to the reflectors' TCP servers, waiting for timestamps, and when received, delivering those timestamps to the main process using a `pipe()`. In order to avoid ping/`connect()` race conditions (when the ping being sent before a TCP connection is established) the continuous ping transmission is not started until a "hello" has been received from the reflector's TCP server. Otherwise, most client sessions would report timestamp errors for the first pings, whenever the configuration is reloaded.

5. Two other design proposals, opposing the one previously described, were evaluated. The first was to streamline `probed` by moving all complex code such as timestamp delivery and client states (matching pings to pongs, calculating RTT, detecting duplicates and timeouts) into a high-level language process such as the `manager`. Using that design, `probed` would simply send, receive and timestamp UDP ping and pong packets. The pings and pongs would be delivered as raw data to the `manager`. One great motivation for doing so, is that the programming language "C" that `probed` is written in is rather outdated, fiddly and prone to bugs compared to higher-level languages such as "Python". The two main reasons for not realizing that design, were that the requirement of being able to

[16] NAT was invented as a temporary solution to the problem of IP address exhaustion

run `probed` stand-alone like `ping` would be slightly crippled, and that the immense amounts of data that the measurement sessions can possibly produce was difficult to elegantly coop with given the additional layers and levels of abstraction that the design proposed. The layers of abstraction included classes for ping/pong data, holding state in a relational database, and delivering timestamps using XML-RPC. The second design proposal was to use "C++" as programming language for `probed`, perhaps using threads. "C++" contains many constructs for higher-level abstraction, and reduces programming efforts in creating for example lists and binary trees. Threads allows for simplified sharing of variables between different threads of execution and elegant locking mechanisms. The primary reason for not using "C++", was that the finally chosen "C and fork design" was evaluated prior to "C++ and threads design". Since the former was found fitting, the extra work of rewriting portions of the code was deemed unnecessary.

4.5.4 Using Linux's timestamp API

The purpose of `probed` was to enable very high accuracy delay measurements, which requires highly accurate timestamps. However, when running on an off-the-shelf Linux operating system, only the user-land timestamp mode is available, because the necessary API and hardware infrastructure is most likely missing (at least as of 2011). The background to Linux's `SO_TIMESTAMPING` is described in section 4.2, and this section will focus on what is necessary to enable hardware timestamps. These observations are derived from Linux's `Documentation/network/timestamping.txt` and PTP implementation[7].

For both kernel and hardware timestamps, the kernel will most likely have to be rebuilt, since the `CONFIG_NETWORK_PHY_TIMESTAMPING` kernel configuration option is disabled by default. The option is available from version 2.6.36 and is still available in 2.6.38.

One patch developed during this project called "[PATCH] fixing hw timestamping in igb"[17] is required in order for hardware timestamps with Intel 82580 on Linux 2.6.37 to work. It has been applied, and will be shipped with future Linux versions. The patch is included in appendix listing A.1.

Another patch also developed for this project called "[PATCH] net: TX timestamps for IPv6 UDP packets"[18] is required in order to use kernel or hardware timestamps with IPv6 on Linux 2.6.37. It has been applied as well, and is included in listing A.2.

There are, as of version 2.6.38, no drivers supporting kernel TX timestamps. It's however simple to enable it for the desired network controller manually, by adding a line of code to the "start_xmit" (pointed at by `.ndo_start_xmit`) function in the driver. The exact location of the code

```
skb_tx_timestamp(skb);
```

[17] Archived at http://marc.info/?l=e1000-devel&m=129673051106564&w=2
[18] Archived at http://marc.info/?l=linux-netdev&m=129841103607145&w=2

in the function does matter, both in terms of accuracy and functionality. A safe bet for reliable operation at the expense of accuracy is to place it close to the top of the function. As of version 2.6.38, there is however still a trivial "bug" causing a compile error if one does not also add

```
EXPORT_SYMBOL_GPL(skb_clone_tx_timestamp);
```

to `net/core/timestamping.c`. The kernel timestamps are however not used in the final version of the SLANG platform, and the kernel shipped with the SLANG node system image does not support kernel timestamps for any drivers.

All timestamp operations are performed within `probed`'s `tstamp.c` source file, including Linux's `net_tstamp.h`. The three timestamp modes (hardware, kernel and userland) have their respective functions for enabling, and if a more demanding mode (hardware) fail to initiate, `probed` automatic falls back to a less demanding mode (kernel).

To enable hardware timestamps, two system calls are required. The first activates hardware timestamps on the hardware itself by `ioctl()` with `SIOCSHWTSTAMP` on the socket with a `struct ifreq` as request data, with the adapter name (such as eth0) as `ifreq.ifr_name` and a `struct hwtstamp_config` as `ifreq.ifr_data`. For the Intel 82580 controller, the `hwtstamp_config`'s `tx_type` should be set to `HWTSTAMP_TX_ON` for timestamping of arbitrary packets and the `rx_filter` to `HWTSTAMP_FILTER_ALL` for per-packet timestamping. The second system call activates RX and TX timestamps on the socket, in order for Linux's network stack to determine which packets should receive timestamps, and which should not, by `setsockopt()` option `SO_TIMESTAMPING` on protocol level `SOL_SOCKET` with the masked value of `SOF_TIMESTAMPING_TX_HARDWARE`, `SOF_TIMESTAMPING_RX_HARDWARE` and `SOF_TIMESTAMPING_RAW_HARDWARE`.

To read (extract) the timestamp from a packet, it has to be received with `recvmsg()`. By iterating the `struct msghdr` with `CMSG_NXTHDR()` looking for the type `SO_TIMESTAMPING` on level `SOL_SOCKET` and casting that value to

```
struct scm_timestamping {
        struct timespec systime;
        struct timespec hwtimetrans;
        struct timespec hwtimeraw;
};
```

the timestamp as generated by the network controller's oscillator is obtained in `hwtimeraw`. For received (RX) packets, that is basically it. For sent (TX) packet however, the process is somewhat more complicated. Following the `sendto()`, the TX timestamp has to be obtained before sending again. The timestamp is made available by looping the original back to the socket's error queue, with the RX timestamp of that packet being the TX timestamp of the sent one. Instead of having to create a queue for packets awaiting transmission, each `sendto()` is followed by the blocking[19] function `tstamp_fetch_tx()` trying to obtain the timestamp. That is acceptable, since the timestamp is made available "very quickly". The function performs a "read" `select()` with a short timeout on the

[19]Blocking indicates that no other functions in the process takes place during that time; in this case resulting in the operating system having to queue RX packets until the function returns

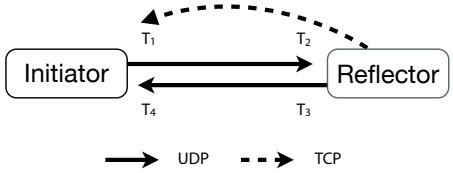

Figure 4.5: Protocol communication

UDP socket; blocking until the timestamp arrives. The reason for not calling `recvmsg()` directly, is that a read on the error queue (flag `MSG_ERRQUEUE`) is non-blocking, and would return `EAGAIN` directly if the timestamp was not immediately available. Running `recvmsg()` in a loop until a timestamp is received would result in busy-waiting [20]. An effect arising from the "read" `select()` is that it is triggered by RX packets and real errors as well, therefore requiring an outer loop with a `gettimeofday()` based timeout. Errors are differentiated from timestamps by checking that `ee_origin` is `SO_EE_ORIGIN_TIMESTAMPING`.

4.5.5 Protocol

The protocol was specified early in the design phase, and was never altered throughout the project duration. It's deliberately simple, because it addresses an isolated, defined, limited problem. Also adding to it is simplicity is the fact that it is not at all extensible or portable, since it was deemed unnecessary. Changes required by future extensions could be added as additional fields at the end of the structure, but padding, alignment and endianess issues has to be solved with an updated `probed` version whenever portability is needed.

As described in section 4.5.3, the initiator sends a ping (over UDP) to the reflector, which responds by sending a pong (over UDP) and the two timestamps T_2 and T_3 (over the TCP session initiated by the initiator).

UDP is being used for the measurement packets (ping/pong) in order to benefit from UDP's level of abstraction, while escaping TCP's reliability and ordering features which would mask effects that are being measured. Figure 4.5 illustrates the principle by which the protocol operates, and the packet payload for both ping, pong and timestamp packets has the following structure:

```
#define DATALEN 48
#define TYPE_PING 'i'
#define TYPE_PONG 'o'
#define TYPE_TIME 't'
#define TYPE_HELO 'h'

struct packet_data {
       char type;
       uint32_t seq;
       uint32_t id;
       struct timespec t2;
```

[20] Busy-waiting and spin-locking are related, normally referring to the process of waiting for a condition by checking repeatedly, wasting resources in a multi-tasked environment

```
        struct timespec t3;
};
```

The first field (8 bits; 1 byte) is the packet type; being one of the `TYPE_` defines. In future revisions it could be migrated to an enumerated type (`enum`). The character type was inherited from the development and debugging phase, as it allows one to operate the protocol using simple tools such as "netcat"[21]. The next field (32 bits; 4 bytes) is the sequence number; used to match pings, pongs and timestamps. The following field (32 bits) is the measurement session identification, which allows for differentiation between several measurements from the same IP address and port. The last two fields (128 bits; 16 bytes each on Linux for x86-64[22]) contains one `time_t` and one `long int`. The `time_t` equals to a `long int` on most operating systems, such as Linux[23]. The size of the struct is declared to 48 bytes by `DATALEN`, which holds true for `probed` compiled on a Linux for x86-64 with the *GNU C Compiler* (GCC) using the flags of the included `configure` script. Simply adding the sizes of the structure members gives 41 bytes; with the extra 7 bytes being compiler and platform dependent padding added in order to speed up memory accesses[36]. Both alignment (normally 4 or 8 bytes) of the members and last member padding with the number of bytes required for the structure size to be a least common multiple of the size of the largest member[43] may affect the size. As noted earlier, the protocol should be improved if any portability is desired. The packet payload fits without one UDP packet with a great margin. No payload size option exists for the time being.

There are two reasons for including the T_2 and T_3 timestamps in all packets. First of all, the same packet structure can be used for all packets. Secondly, it would allow for future controllers with the hypothetical possibility of writing TX timestamps into sent packets to operate without the TCP session.

4.5.6 Client state

In section 4.5.3 the decision to handle client states in `probed` was motivated. The client state is a broad term for all logic required to match ping, pong and timestamp packets to each other in order to detect timeouts, duplicates, DSCP errors, calculate delays, and so on.

Tests indicated that early versions of the `igb` driver failed to match RX timestamps to received packets in situations when two packets arrived very close to each other. These errors presented themselves as the first packet receiving the timestamp of the second packet, and the second packet receiving no timestamp at all. Therefore, early versions of `probed` included additional client state logic in order to mask these errors, by being able to "invalidate" timestamps for previous packets. For example, if `probed` (operating as a reflector) detects a

[21] Netcat, invoked with `nc`, is a simple tool for sending and receiving TCP and UDP packets

[22] x86-64 is the extension name for 64-bit "Intel" (x86) architecture; also referred to as "x64"

[23] Verified by compiling and running `sizeof (time_t);` or by searching the include headers: Linux defines it from `__time_t` in `time.h` defined from `__TIME_T_TYPE` in `bits/types.h` defined from `__SLONGWORD_TYPE` in `bits/typesizes.h` defined from `long int` again in `bits/types.h`

missing T_2 (RX) timestamp on a received ping, it assumes that the previous ping's timestamp was invalid, and sends an additional timestamp packet for the previous ping (to the client that ping originated from, with the previous sequence number). The RX timestamp error detection was scrapped in the final release, but introduced the first revision of the client state machine keeping track of the current pings in transmission. Consequently, other approaches not explicitly holding state for pings were never realized.

The client state machine is found in `client.c` and implements `sys/queue.h`'s linked list as temporary storage, with a bit-mask indicating received pong/timestamp and DSCP error. Whenever a ping has received both ping and timestamp, or the timeout has been reached, one of the following statuses are reported on the terminal or FIFO API, depending on operation mode:

```
#define STATE_OK 'o'       /* Ready, got both PONG and valid timestamp */
#define STATE_DSERROR 'd'  /* Ready, but invalid ToS/traffic class */
#define STATE_TSERROR 'e'  /* Ready, but invalid timestamps */
#define STATE_PONGLOSS 'l' /* Ready, but timeout, got only timestamp, lost PONG */
#define STATE_TIMEOUT 't'  /* Ready, but timeout, got neither PONG or timestamp */
#define STATE_DUP 'u'      /* Got a PONG we didn't recognize, DUP? */
```

The many states are partly explained by the fact that the pong and timestamp are delivered in separate packets, over separate protocols, thus increasing the number of result combinations. There is however no distinction between a lost timestamp (normally caused by the TCP session not delivering it properly) or an invalid timestamp (possibly caused by the network controller or kernel failing to produce timestamps). The DSCP error is triggered if the pong is received with a DSCP value, the six most significant bits of IPv4's *type of service* (ToS) and IPv6's traffic class field, is different from the one in the sent pong. The most common cause of a DSCP error is that a router in the path has reset the value. All client results are handled in the main (UDP) process, rather than in the client-specific `fork()` processes in order to minimize inter-process communication.

The client sessions are maintained in a separate linked list, built from the XML configuration file. When the main processes receives the signal "HUP" a flag is raised requesting a configuration reload as soon as possible. The reason for only raising a flag upon signalling rather than reloading the configuration momentarily, is to reduce the risk of crashes due to functions and calls not being "signal safe"[24]. The configuration reload kills all `fork()` client processes, empties the result and session lists, re-populate the session list based on the XML file, and `fork()` new client processes based on the session list. Therefore, information about pending pings are lost during reload. Since round-trip times are short, normally shorter than the interval between pings, very little (if any) data is lost in reality. It's however unfortunate that the rare case of returning pongs for pings that were deleted will be erroneously reported as duplicates. Future versions may contain a duplicate threshold to deal with that issue.

[24]Since signals interrupt the process execution flow, what can be safely done inside a signal handler is limited

4.5.7 Static code analysis

Because `probed` is never restarted during normal operation, and that the SLANG appliances ideally should not be rebooted either, it is important that `probed` runs stable and without memory leaks. To the highest possible extent, dynamic memory allocation has been avoided; only leaving host name resolving with `getaddrinfo()`, `malloc()` of items in the two linked lists, and data fetched with `xmlNodeGetContent()`. The size of the linked client result list is what's allowed to affect the memory consumption during operation; and is limited by the ping timeout (of 10 seconds in the current version). On the SLANG appliances, `probed` has been using 1032 kB[25] of physical memory constantly during several weeks of continuous operation, and 504 kB per client `fork()`.

With the intention of finding memory and storage bugs, the lint[26], (static code analysis) tool *SPlint*[27] was applied to the code base from the very beginning. Because it requires annotations added as comments in the code in order to properly interpret many constructs, it has assisted in preventing numerous programming mistakes. SPlint did not perform flawless however, and required meta-state annotations as a consequence of not being able to parse some system headers. Different combinations of dependencies and incompatibilities forced separations such as the `unix.c` file, containing wrapper functions for functions and macros that requires SPlint's Unix library; which was incompatible with system headers required by other files. Also adding to the large number of annotations is the fact that it didn't properly interpret library macros and functions such as `libxml2`'s `xmlFree()` and `queue.h`'s linked list.

4.5.8 Program flow at source code level

Whereas reading the generously commented and documented source code provides the most accurate rendition of the workings of the program, the simplified flow chart in figure 4.6 on the following page will be used in this section.

During program startup, some choices are made by the argument parser. The first operations however, is to initiate logging capabilities, and the `bind_or_die()` function that configures two socket (UDP for ping/pong and TCP for timestamps) with all required socket options required for IPv6/IPv4-mapped communication with DSCP abilities. Depending on the chosen timestamping mode, different functions in `tstamp.c` are called which activates timestamping on the UDP socket (and network adapter, in the case of hardware timestamps). There is also so-called operation modes; server (reflector), client (initiator) and daemon (also initiator). Irrespective of operation mode, the server code is always executed, implying that the client mode is in fact both providing an initiator and a reflector. The difference between the client and daemon mode is:

[25] In this case, one kilobyte is 2^{10} bytes

[26] The term "lint" in computer programming refers to the process of finding suspicious usage of computer language

[27] Available at http://www.splint.org/

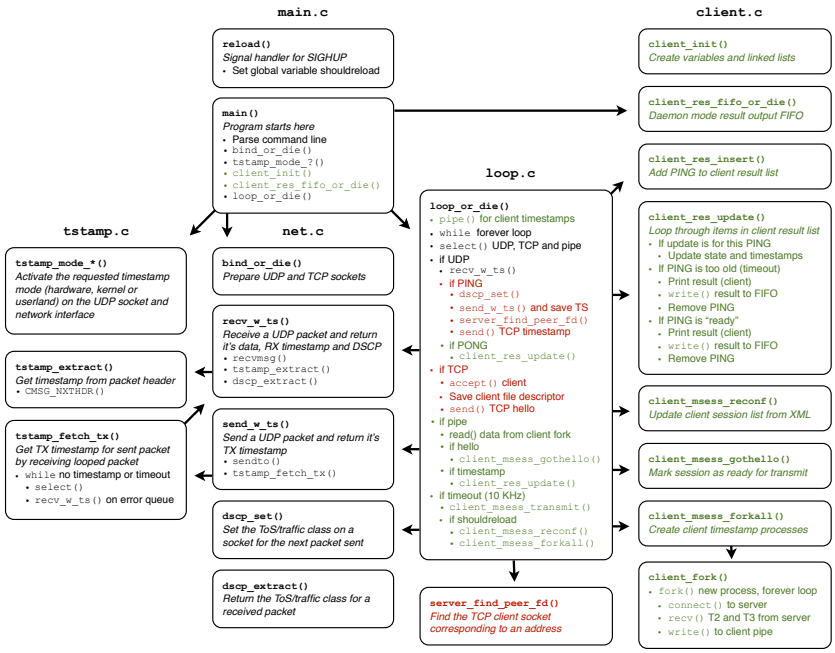

Figure 4.6: Source code chart

- The daemon mode outputs results to a named pipe (FIFO) whereas the client mode outputs to the console (stdout) in a similar fashion to `ping`.
- The client mode initiates a client session against one reflector, as specified by the command-line argument, whereas the daemon mode supports several client sessions configured according to an XML configuration file.

Functions that are exclusive for the daemon mode are `client_res_fifo_or_die()` that provides the FIFO and `client_msess_reconf()` that reads client sessions from the configuration file. In the chart figure, all green text indicates initiator-specific code, and red text reflector-specific.

Then, the `loop_or_die()` function is invoked, beginning with the creation of a `pipe()` for *interprocess communication* (IPC) with the client timestamp processes. An infinite loop is entered, blocking at a `select()` on all input sources: pings (server) and pongs (client) on the UDP socket, client timestamp connect requests (server) on the TCP socket, and finally client timestamps (client) from process `fork()`s on the pipe. Compare with the figure. On the `select()` timeout pings are sent (client) and client session are configured and `fork()`ed upon launch or reload (client). In order to avoid false timestamp errors because of the race condition between early pings and the timestamp TCP connection; ping transmission is not started for a client session (msess) until the client has received a "hello" message from the server's TCP `accept()` routine which is reported with `client_msess_gothello()`.

When starting a ping against itself (`127.0.0.1` or `::1`), the execution flow of the loop would be

1. Idle, waiting for the loop's `select()` to timeout.
2. Timeout is reached; `client_msess_transmit()` sends pings to all measurement sessions that is ready for transmission according to the configured interval, and has received a "hello" (no-one has). Since this is the first iteration, "shouldreload" is true, and thus a client process is started with `client_msess_forkall()`.
3. `client_fork()` creates a new process, which `connect()`s to the server address (itself).
4. The loop's `select()` detects activity on the TCP socket, `accept()`s the connection from the `fork()`ed client process, adds it to the `selects()`s file descriptor set, and `send()`s a "hello" packet.
5. The `fork()`ed client process `recv()`s the "hello", and forwards it to the main loop process by `write()`ing it to the shared pipe IPC.
6. The loop's `select()` detects activity on the pipe, and marks the measurement session as ready with `client_msess_gothello()`.
7. Idle, waiting for the loop's `select()` to timeout.
8. Timeout is reached; `client_msess_transmit()` `sendto()`s a ping to the server (assuming that the session is ready for transmission according to the configured interval) with the configured DSCP and records the ping and it is T_1 (TX) timestamp with `client_res_insert()`.
9. The loop's `select()` detects activity on the UDP socket, `recvmsg()`s the ping and notes its RX timestamp and DSCP value. It then responds by `sendto()`ing a corresponding pong, noting its TX timestamp, and `send()`ing the T_2 (RX) and T_3 (TX) timestamps over TCP to the client's socket as identified by `server_find_peer_fd()`.
10. The loop's `select()` detects activity on the UDP socket, `recvmsg()`s the pong while recording its T_4 (RX) timestamp with `client_res_update()`.
11. The fork()ed client process `recv()`s the timestamp packet, and forwards it to the main loop process by `write()`ing the shared pipe IPC.
12. The loop's `select()` detects activity on the pipe, and records the timestamp packet's T_2 and T_3 timestamps with `client_res_update()` which notes that all timestamps are present, and thus outputs the results.

The process is not, as the enumerated list suggests, sequential. The nature of `select()` allows any of these events, from multiple client and server sessions, to occur "simultaneously" in an asynchronous, interleaved manner.

4.6 Data collection and interface program (`manager`)

The manager is built up by four main parts.

- `Manager`, the core class which keeps track of the different components. It creates instances of component classes, starts threads, handles UNIX signals and also handles incoming XML-RPC requests.
- `Probed` performs all operations related to the `probed` program; starting `probed`, opening and reading the FIFO. The data is parsed into a `Probe` object and passed to the `ProbeStore`. The FIFO reading is run in a separate thread.
- `ProbeStore` is the most extensive part of the `manager` application, containing more than one third of the code. The ProbeStore performs three distinct tasks, namely keeping state of and pre-calculating received measurement data, storing it in an in-memory SQLite database and pre-calculating stored data when it is requested. It also saves pre-calculated aggregated data over 300 seconds.
- `Maintainer` performs maintenance operations at regular intervals such as deleting old data from database, generating aggregates and connecting the central control panel service looking for configuration changes once every hour.

4.7 Central management (SLANG control panel)

A requirement which Tele2 put heavy emphasis on was ease of administration with a vision that as much work as possible should be automated. A major part of the administrative efforts has previously been setting up new measurement nodes and setting up all relevant measurement sessions. Due to this a lot of effort has been put into simplifying these procedures in the SLANG system.

The component which performs the central administrative tasks is the SLANG control panel, essentially a web application whose primary duty is to supply the nodes with an up-to-date configuration, a list of all measurement sessions to set up. As the development progressed, other features was added along the way which gives an overview of the current nodes, measurement groups and measurements.

The control panel itself does not store any data about what nodes there are or anything else for that matter; instead it it tightly coupled with Tele2's *Node and Information Listing System* (NILS) via a remote API which was added to the current NILS system together with a few smaller modifications to accommodate SLANG data. NILS is basically a database containing all nodes in Tele2's network their function, location and other data regarding them. It was for SLANG extended slightly. One extension is the addition of SLA groups, groups to which nodes can belong. Between the nodes belonging to the same SLA group a mesh of measurements can then be generated by the control panel, a

mesh which can look different depending on what type of SLA group is regarded and what sessions the group is configured to set up. Currently, only one SLA group type is defined; the *full-mesh* type. As can be imagined, a SLA group of type full-mesh will perform full mesh measurements, that is there will be measurements defined between all nodes in the group.

When the `manager` (see section 4.6) of a SLANG node starts up, it fetches its configuration from the control panel. When the control panel receives a configuration request a quite extensive process is initiated.

1. First, information regarding the connection node is fetched from NILS together with its SLA group assignment, the SLA group configuration and a list of all nodes belonging to the same group. The node requesting configuration will hereafter be addressed as the *current* node. Note that a SLA node can belong to multiple groups and information regarding all groups are fetched. Also a list of current measurements the current node is involved in is fetched.

2. From the SLA group configuration a set of all measurements involving the current node is generated. To eliminate double measurements the set elements is defined so a measurement from node A to node B is considered equal to a measurement from node B to node A. The procedure is repeated for all SLA groups the node belongs to, with the measurements for all groups added to the same set, also this to eliminate duplicates.

3. The set of all calculated measurement sessions, the *required* sessions are compared to the set of *current* sessions fetched from NILS. If there are differences (due to new sessions added to the group template or nodes added/removed), take note of all other nodes affected by the differences and make sure that the NILS database is updated.

4. If there were differences which affected other nodes, start over from step 1 for each of the other affected nodes and push out their new configuration to them via the `manager` XML-RPC API.

5. Generate configuration and return to the current node.

As previously stated the SLANG control panel is also able to present data to the end user. It gives an overview of the current state of the system by listing nodes, groups and measurements as well as displaying current measurement statistics and node internal memory statistics. Neither for this any state is kept in the control panel application, except from some short-time (one minute) caching of the data. Instead, everything is fetched from NILS, ASM and directly from the SLANG nodes.

To speed up the development (and avoiding to reinvent the wheel) the control panel was build using the Pylons Python web development framework together with model classes built to work with the XML-RPC interfaces provided by NILS and the SLANG nodes.

4.8 Node and Information Listing System (NILS)

NILS is a system developed at Tele2 to keep track of their network nodes. As new equipment is deployed or removed the NILS database is updated as a part of the work flow to mirror the current state.

The functionality of NILS was slightly extended due to SLANG requirements.

- Each node can now be member of an SLA group. An SLA group groups together nodes between which a specific set of measurements is supposed to be set up. Each group has a name, a description and a type, for example "slang" for SLANG measurements.
- The SLA groups have a list of measurement templates defining what types of measurements should be performed, for example one IPv4 session sending 100 packets per second with DSCP `ef` and one IPv6 session, 20 packets per second in the best-effort traffic class.
- A list of the current running measurement sessions. This is mainly kept to make sure that each measurement session has a unique static ID which can be used when measurement data is fetched from the nodes.
- Then, an external XML-RPC API to many of the NILS functions was added to give external systems (primarily the SLANG control panel) access to the NILS database.

4.9 Using SLANG

Users will be able to monitor the network, add or remove nodes, and configure the measurement sessions by simply browsing to the user-friendly SLANG control panel or using NILS's command-line interface. Administrators on the other hand are required to understand the many components of SLANG, and their interfaces, in order to fully comprehend the working of the system and service it in the unfortunate cases of failures. In the upcoming sections, the compilation of the code into deployable nodes will be detailed, followed by administration tasks, and concluding with end-user tasks.

4.9.1 Creating an operating system USB-stick

The operating system image, based on Debian 6.0, is simply a base operating system. Apart from Debian itself, it contains a timestamp-enabled kernel, basic packages, configured LDAP authentication with a configuration console as shell, and many changes relating to its ability to run on a read-only root file system. It weights in at a mere 62 MB, and is copied to a USB-stick by typing

```
$ gunzip -c -d sla-ng.img.gz | dd of=/dev/disk1 bs=1048576
```

where "/dev/disk1" is the path to the USB stick. When plugged into a computer supporting USB booting, it should simply start, outputting to the VGA and RS-232 (serial) consoles. In order to populate it with the SLANG programs, log in as the super-user "root" and execute the Debian package installation commands described in the next section.

4.9.2 Compiling from source

Although many components are developed in high-level scripting languages, some of the code has to be compiled. While in the repository's source code root folder, type

```
$ ./deploy install
```

to create the necessary build scripts using `autoconf`. The project has been tested with GNU Autoconf 2.67. Then, or if one is building from a release with build scripts in place, type

```
$ ./configure
```

to generate the `make` files. Finally, build the `probe` program binary with

```
$ make
# make install
```

with the latter command also installing the program into the default program folder if executed as super-user "root". Since the Linux-based operating system Debian was used as foundation for the project, one can type

```
$ ./deploy debian
```

to to create a Debian package that is suitable for deployment. The package contains `probed`, `sla-ng-manager` that collects the data and makes it available over XML-RPC, `sla-ng-view` that can be used to monitor `probed` (when in daemon mode) and the `sla-ng-manager`, a start-up script for `init.d` and finally the configuration user interface `ui.sh`. To install the Debian package, type

```
# dpkg -i sla-ng.deb
# apt-get install -f
```

as "root". The second command's purpose is to download and install any dependencies required by the package. Verify that `probed` is installed by typing

```
$ probed
```

and compare the output with

```
SLANG probed 0.1
usage: probed [-kqsu] [-c addr] [-d path] [-i iface] [-p port] [-f path]

                MODES OF OPERATION
        -c addr   Client: PING 'addr', print to standard output
        -s        Server: respond to PINGs
        -d path   Daemon: server and (many) clients, print to FIFO 'path'

                OPTIONS
        -f path   Daemon only, path to config file [default: probed.conf]
        -w time   Client only, wait time between PINGs [default 500] (ms)
        -i iface  Network interface for hardware timestamps [default: eth0]
        -p port   UDP port, both source and destination [default: 60666]
```

```
-k        Create timestamps in kernel driver instead of hardware
-u        Create timestamps in userland instead of hardware
-q        Be quiet, log to syslog only
```

which happens to be the brief help message for the program.

4.9.3 Running `probed` stand-alone

For testing purposes or short-time measurements `probed` can be invoked manually. In order to verify that the program is working, start it with the simplest possible configuration by running

```
$ probed -u -c localhost
```

where "u" instructs it to timestamp in user-land (kernel or hardware timestamps requires patched kernels and/or certain hardware) and "-c localhost" to start in client (initiator) mode, pinging the loopback interface of the local computer. In reality, also the server (reflector) mode is started; it is implicit. Press CTRL+c to interrupt the ping. The output will be

```
SLANG probed 0.1
probed: Binding port 60666
probed: Using userland timestamps
probed: client: ::1: Connecting to port 60666
probed: server: ::1: 8: Connected
probed: client: ::1: Connected
Response    1 from 0 in 11776 ns
Response    2 from 0 in 15257 ns
^C
2 ok, 0 dscp errors, 0 ts errors, 0 unknown/dups
0 lost pongs, 0 timeouts, 0.000000% loss
max: 15257 ns, avg: 13516 ns, min: 11776 ns
```

if the program was correctly installed, the port 60666 was available, the loopback interface is up, and the computer supports IPv6. The program also supports IPv4; but using IPv4-mapped IPv6 sockets. The address `::1` the actually the IPv6 loopback address; similar to `127.0.0.1` for IPv4. The first line reports the software version. The second and third line is shared by both client and server, and confirms that the required settings were successfully used. The fourth line is the client's (initiator's) TCP client `fork()`, doing a `connect()` to the server's (reflector's) TCP server. The fifth line is the main loop (shared by both client and server) doing an `accept()` of the connection into socket file descriptor "8". On the sixth line, the client announces that it has received a "hello" packet from the server, and will therefore start sending UDP pings. Not surprisingly, the local computer responds to itself, and the results are printed when CTRL+c (interrupt) is pressed.

For a real-world measurement using hardware timestamps, log into a SLANG node with a supported network adapters such as Intel 82580, or prepare a server according to section 4.5.4. One way of confirming that the hardware timestamps are working, is to directly connect to nodes using only a cable/fiber, and execute a measurement. As the super-user "root", start a server (reflector) on one node by typing

```
# probed -s -i eth2
```

where “eth2” is a supported network adapter. Compare the output with

```
SLANG probed 0.1
probed: Binding port 60666
probed: Using hardware timestamps
probed: Server mode: waiting for PINGs
```

and verify that no errors occurred. Typical errors include the port being used (use “-p 1234” to choose another) or the hardware not being supported (`SIOCSHWTSTAMP` not being supported; verify that the correct network adapter was chosen with “-i”). Start a client on the other computer by typing

```
# probed -c 10.0.0.2 -i eth2
```

comparing the output with

```
SLANG probed 0.1
probed: Binding port 60666
probed: Using userland timestamps
probed: client: ::ffff:10.0.0.2: Connecting to port 60666
probed: client: ::ffff:10.0.0.2: Connected
Response    1 from 0 in 1240 ns
Response    2 from 0 in 1240 ns
^C
2 ok, 0 dscp errors, 0 ts errors, 0 unknown/dups
0 lost pongs, 0 timeouts, 0.000000% loss
max: 1240 ns, avg: 1240 ns, min: 1240 ns
```

which shows the typical delay (1240 ns; about 1 μs) and delay variation (close to 0 μs) for a working hardware timestamp configuration with Intel 82580 controller.

There are a couple of common misconfigurations, causing confusing errors. To start with, the “TX timestamp error” will occur if hardware or kernel timestamp is chosen, and for example pinging localhost. That is because no timestamp will be generated, since the packets never leave the computer via a supported network adapter. Another scenario causing the same problem is in computers with multiple network adapters, and packets leave through the wrong interface. Let’s say that hardware timestamps were requested on `eth2`, but the packet leaves on another interface called `eth1` (because the routing table instructed it to do so) with kernel timestamp support only; TX timestamp errors will occur.

Beware that since timestamps are delivered to the initiator from the reflector via TCP, there is a possibility of UDP pings being successful, but missing timestamps and thus RTT; or the other way around.

4.9.4 Running `probed` in daemon mode

In Linux and other Unix-like operating systems, a daemon is a program running in the background, usually providing a service. When running `probed` in daemon mode, it starts a server and a number of clients based on a configuration file with syntax

```
<config>
  <probe id="1">
    <address>130.244.97.213</address>
    <dscp>0</dscp>
```

```
    <interval>5000</interval>
    <type>slang</type>
  </probe>
</config>
```

The client results are outputted to the named `pipe()` specified by the FIFO argument with a structure similar to the SLANG ping/pong packets, and identical to the client result structure

```
#define STATE_OK 'o'       /* Ready, got both PONG and valid TS */
#define STATE_DSERROR 'd'  /* Ready, but invalid TOS/traffic class */
#define STATE_TSERROR 'e'  /* Ready, but invalid timestamps */
#define STATE_PONGLOSS 'l' /* Ready, but timeout, got only TS, lost PONG */
#define STATE_TIMEOUT 't'  /* Ready, but timeout, got neither PONG or TS */
#define STATE_DUP 'u'      /* Got a PONG we didn't recognize, DUP? */

struct res {
     struct timespec created;
     char state;
     struct in6_addr addr;
     uint32_t id;
     uint32_t seq;
     struct timespec ts[4];
     LIST_ENTRY(res) list;
};
```

The field "created" usually equals to T_1, but since T_1 is empty in the case of timestamp errors and not necessarily synchronized with the wall time for hardware timestamps, a separate field always populated with the computer clock's wall time is needed in order to determine timeouts, etc. The "state" is a `char` (not an `enum` of historical reasons, having to do with the structure fitting within 128 bytes) which equals to one of the defines above. The address "addr" is the server (reflector) address that was being pinged. The "id" and "seq" are used together with the address to match packets. The array "ts" contains the four timestamps, T_1 to T_4. The "list" is an internal variable used by `probed` to maintain a linked list. The program `sla-ng-view` can be used to read from the FIFO, and view the results on the terminal. Note that `probed` are rarely started in daemon mode manually, but rather launched by the `sla-ng-manager` that is started on boot on SLANG nodes.

4.9.5 Using the `sla-ng-manager`

As this program's purpose is to collect and aggregate information from `probed` and make it available on an XML-RPC interface, it is rarely invoked manually, but rather started by the SLANG nodes' `/etc/init.d/sla-ng` start-up script on boot. To restart it, type

```
# /etc/init.d/sla-ng restart
```

but beware that the data collection will be restarted, creating a "gap" (statistics outage) of a few minutes in the graphs. Normally, the SLANG control panel interfaces automatically with the `sla-ng-manager`, but it is possible to perform many tasks manually as well. A few examples using Python as programming language will be presented hereafter. Starting the interpreter on your computer with

Figure 4.7: The SLANG's configuration console; `ui.sh`

```
$ python
```

and prepare a connection by typing

```
>>> import xmlrpclib
>>> s = xmlrpclib.ServerProxy('http://lab-slang-1.tele2.net:8000')
```

where "lab-slang-1.tele2.net" is the host name or IP address of the SLANG node. Functions are then executed on the object "s", like

```
>>> s.reload()
```

that forces the program to download the `probed` XML configuration from the SLANG control panel (as defined by the configuration file `/etc/sla-ng/manager.conf`) and instructs `probed` to reload the configuration by sending the `HUP` signal. The most commonly used function is probably `get_last_dyn_aggregate(id, num)` that returns the last `num` number of aggregated 5-minute intervals from measurement session `id` with dynamic resolution, depending on packet loss.

4.9.6 Deploying a SLANG node

A complete SLANG node is a computer with an Intel 82580 network card, bootable using the USB-stick, and populated with the SLANG Debian package. Such a node is easily deployed thanks to the configuration console, that is automatically started for LDAP users. For other users, such as "root", simply type

```
# ui.sh
```

to launch it. The main screen is depicted in figure 4.7.

From within the configuration console, important settings such as IP addresses and host name is set. It is also possible to perform debugging using for example the ping viewer. It reads directly from `probed`'s output FIFO, and when started with "-1" it shows all results, for all measurement sessions. If one of the sessions is found particularly interesting, the viewer for aggregated pings can be used, asking for a specific measurement session ID. It will query the `manager`'s XML-RPC interface for aggregated pings during a user-defined period of time.

As noted in section 4.4 the system may be administered directly using Linux's shell. It is accessed by choosing "Start Shell (bash)" from the menu, and allows for more in-depth access to the measurement node.

Chapter 5

Conclusions

The project objective as stated in the introduction was *to provide Tele2 with a modern, accurate quality assurance and reporting system.* The initial assumption was that affordable, commercially available products would provide sufficiently accurate measurements. As the tested products proved to be lacking the project scope grew in size, as measurement nodes had to be developed.

Despite the constrained schedule, the final system did provide all required functions on time. As noted in section 5.2 there is however room for improvement in several areas; such as further validation of the accuracy, adjusting the protocol and programs to adhere to best practices, et cetera. The accuracy requirement that disqualified most commercial products were met with flying colors; better than 1 μs. In some configurations the reported jitter of a short wire was 8 ns; suggesting exceptionally good accuracy.

5.1 Discussion

Many of the thesis' topics can be further elaborated. A large number of design choices has been made, some which are mentioned, other not. This section discusses some of these topics.

The estimation of measurement accuracy which was described in section 3.2 has room for improvements. The method of simply performing a large number of measurement between two devices connected directly without any interfering equipment and looking for the maximum delay variation between two consecutive packets might not be very scientific, but provides an idea about the stability of the measurement. To simply look at the difference between maximum and minimum delay measured over a time period might give a better result, but as the variation seem is so much lower than the accuracy requested by Tele2 (100 μs) any deeper studies is not required. The maximum delay variation ever seen during the tests has been around 150 ns, and the decision has been made to take a conservative approach and state an accuracy of less than 1 μs; the time it takes for an electric signal to travel $\approx$192 m in a twisted-pair cable.

As actual timestamps are collected when packets are sent and received, it seems highly unlikely that a bug in the measurement application `probed` or the timestamping API skews the delay measurements to the low variation seen. The same application and API has also been used with kernel-space timestamping. These tests have demonstrated other characteristics, which can be expected from kernel-space timestamps, and thus adds to `probed`'s credibility.

In next section it is proposed that the protocol itself is improved before being exposed to a wider audience. Its imperfections were noted in section 4.5.5, and can only be justified by the fact that architecture portability nor extendibility were priorities during the development process that was focused on producing a "proof of concept".

One of `probed`'s weaknesses, especially when it comes to attracting a wide audience, is the fact it requires re-compilation and even patching of the Linux kernel. Although kernel timestamps could be implemented in all network drivers without any side effects, that has not happened during the relatively long time that the infrastructure has been in place. Even with the patches developed by this project, and other driver patches, Linux still has the timestamp API disabled by default, and has to be re-compiled for it to work. Finally, typical Linux distributions ship with Linux kernel versions much older than what is available. For example, Debian 6.0 that is used for this project, was released in February 2011, with kernel 2.6.32 which was released in December, 2009. On the bright side, the participants of this project didn't experience the notoriously difficult process of getting patches into the official Linux kernel, although none of us are kernel developers nor had committed to Linux before.

Early tests has shown that the system can handle about 10,000 pps. With some optimization this can probably be increased to support higher rates making higher measurement packet rates and more concurrent measurements possible. The fact that `probed` sends pings during the `select()` timeout both inherently limits the number of *packets per second* (pps) per measurement session, and also increases the idle CPU load because of unnecessary timeouts. There is also currently not possible to exactly time the interval between pings, because sessions with identically configured interval (pps) will be sent sequentially. All these issues could be resolved by replacing the timeout with precision timers triggering the `select()`.

Finally, there is a trade-off between low maintenance and being up-to-date with security patches. One can ask for how long a Linux computer can be securely operated without updates. The kernel itself has a good track record with very few remote exploits. The system is aggressively streamlined, with very few installed software packages. Finally, since Debian is a commonly used operating system, a remote exploit in a core component would probably not go under the radar. Still, in the case of serious security vulnerability, someone would have to log into each and every node, and run Debian's update procedure. Anyhow, it is probable that all nodes deployed will enjoy packet filtering from a fronting router, limiting access to Tele2 networks, or that the SLANG nodes are upgraded with a packet filter (firewall) themselves.

5.2 Further work

The work hereby presented can be extended in a number of ways. For starters, the term "quality" can be further elaborated as the entire concept of using packet loss, delay and delay variation as quality measures has not really been assessed.

Additional effort may be put into the entire SLANG system. Regarding measurements, one-way measurements could be interesting, as would adding support for the standardized *One-Way Active Measurement Protocol* (OWAMP) and *Two-way Active Measurement Protocol* (TWAMP) protocols. Also, Tele2 has requested support for the Juniper RPM as well as Cisco IP SLA measurement protocols to be able to perform measurements against equipment already deployed. The accuracy of measurement will of course be worse, but so will the quality requirements of the topical parts of the network.

Further, it would be interesting to add one-way measurement support. For delay measurement as stated earlier highly synchronized clocks are required, something that could be done with GPS receivers or maybe using the clock synchronization network used by Tele2's SDH network. One-way delay variation measurements can however be derived with high accuracy from the data collected by the current system without any complications.

The current code base for the `probed` program is not very portable; it is not endian-safe and does not take differences in memory alignment of 32 versus 64 bit systems into account. Neither will it compile or run on any other operating system then Linux. If SLANG is to be opened to a wider audience, this would be highly desirable, if not necessary.

5.3 Summary

Surprisingly, none of the commercially available products evaluated proved suitable according to Tele2's requirements for a IP network delay measurement system. It was with a great portion of luck that Linux's timestamp API and Intel's 82580 Ethernet controller were released in time for this project, as they enabled the development of such an accurate delay measurement system.

The system was decomposed into several independent layers, most of them having XML-RPC interfaces in-between. Each of these layers provides a core functionality; such as being an appliance (Intel 82580, Debian USB and `ui.sh`), performing accurate measurements (`probed`), calculating on-node aggregated data (`manager`), automatically computing configurations for all nodes (control panel), integrating the system into Tele2's node and address planning system (NILS) and finally collecting and presenting the statistics (ASM).

Bibliography

[1] One-way transmission time. Recommendation G.114, Telecommunication Standardization Sector of ITU, May 2003.

[2] Internet protocol data communication service - IP packet transfer and availability performance parameters. Recommendation Y.1540, Telecommunication Standardization Sector of ITU, 2008.

[3] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Delay Metric for IPPM. RFC 2679 (Proposed Standard), September 1999.

[4] G. Almes, S. Kalidindi, and M. Zekauskas. A One-way Packet Loss Metric for IPPM. RFC 2680 (Proposed Standard), September 1999.

[5] G. Almes, S. Kalidindi, and M. Zekauskas. A Round-trip Delay Metric for IPPM. RFC 2681 (Proposed Standard), September 1999.

[6] S. Bellovin. Firewall-Friendly FTP. RFC 1579 (Informational), February 1994.

[7] R. Cochran and C. Marinescu. Design and implementation of a PTP clock infrastructure for the linux kernel. In *ISPCS 2010, International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 116 – 121, Portsmouth, October 2010. OMICRON electronics GmbH, IEEE. DOI: 10.1109/ISPCS.2010.5609786.

[8] J. Davidson, J Peters, M. Bhatia, S. Kalidindi, and S Mukherjee. *Voice over IP Fundamentals*. Cisco Press, 2 edition, July 2006.

[9] C. Demichelis and P. Chimento. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). RFC 3393 (Proposed Standard), November 2002.

[10] Eric Dumazet. Re: Problems obtaining software TX timestamps. Linux "netdev" developer mailing list majordomo@vger.kernel.org. 8 Sep 2010, `http://marc.info/?l=linux-netdev&m=128395879031578&w=2`, 2010. Accessed 24 March 2011.

[11] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), May 1994. Obsoleted by RFC 3022.

[12] J. Eidson. IEEE-1588 standard for a precision clock synchronization protocol for networked measurement and control systems - a tutorial. In *Proceedings of 2005 Conference on IEEE-1588 Standard for Clock Synchronization*, Winterhur, October 2005. Agilent Technologies, Inc, `http://www.nist.gov/el/isd/ieee/upload/tutorial-basic.pdf`. Accessed 22 March 2011.

[13] D. Endler and M. Collier. *Hacking exposed VoIP: voice over IP security secrets & solutions*. McGraw-Hill Professional, 2006.

[14] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. Netscope: Traffic engineering for IP networks. *IEEE Networks Magazine*, 14, March 2000.

[15] C. Fraleigh, C. Diot, B. Lyles, S. M. P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *IWDC '01 Proceedings of the Thyrrhenian International Workshop on Digital Communications: Evolutionary Trends of the Internet*, London, 2001. Springer-Verlag.

[16] C. Gordon. Introduction to IEEE 1588 & transparent clocks. Tekron International, `http://www.tekroninternational.com/userfiles/file/transparent_clock_whitepaper.pdf`, 2009. Accessed 22 March 2011.

[17] K. Hedayat, R. Krzanowski, A. Morton, K. Yum, and J. Babiarz. A Two-Way Active Measurement Protocol (TWAMP). RFC 5357 (Proposed Standard), October 2008. Updated by RFCs 5618, 5938, 6038.

[18] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052.

[19] V. Jacobson, K. Nichols, and K. Poduri. An Expedited Forwarding PHB. RFC 2598 (Proposed Standard), June 1999. Obsoleted by RFC 3246.

[20] S. Kalidindi and M. J. Zekauskas. Surveyor: An infrastructure for internet performance measurements. `http://www.isoc.org/inet99/proceedings/4h/4h_2.htm`, 1999.

[21] J. Mahdavi and V. Paxson. IPPM Metrics for Measuring Connectivity. RFC 2678 (Proposed Standard), September 1999.

[22] Inc. Maxim Integrated Products. Application note 4115: How to use jitter buffers on TDMoP products to compensate for packet-delay variation (PDV). `http://pdfserv.maxim-ic.com/en/an/AN4115.pdf`, September 2007.

[23] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305 (Draft Standard), March 1992. Obsoleted by RFC 5905.

[24] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[25] D.L. Mills. Network Time Protocol (NTP). RFC 958, September 1985. Obsoleted by RFCs 1059, 1119, 1305.

[26] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser. Packet Reordering Metrics. RFC 4737 (Proposed Standard), November 2006.

[27] Juniper Networks. Real-time performance monitoring on juniper networks devices. `http://www.juniper.net/us/en/local/pdf/app-notes/3500145-en.pdf`, 2010.

[28] IXIA Octavian Purdila. [RFC][PATCH 1/1] net: support for hardware timestamping. Linux "netdev" developer mailing list majordomo@vger.kernel.org. 29 July 2008, `http://marc.info/?l=linux-netdev&m=121729025219054&w=2`, 2008. Accessed 24 March 2011.

[29] IXIA Octavian Purdila. [RFC][PATCH 1/2] net: support for tx timestamps. Linux "netdev" developer mailing list majordomo@vger.kernel.org. 29 Jul 2008, `http://marc.info/?l=linux-netdev&m=121729194521281&w=2`, 2008. Accessed 24 March 2011.

[30] P. Ohly, D. N. Lombard, and K. B. Stanton. Hardware assisted precision time protocol design and case study. Bruehl, 2008. Intel GmbH, Linux Clusters Institute Conferences. `http://www.linuxclustersinstitute.org/conferences/archive/2008/PDF/Ohly_92221.pdf`.

[31] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[32] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. RFC 2330 (Informational), May 1998.

[33] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), September 1981. Updated by RFCs 950, 4884.

[34] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[35] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

[36] T. J. Rothwell. The gnu c reference manual. `http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf`, 2009. Free Software Foundation, Inc.

[37] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051.

[38] B. Söderberg. *Prosilient Technologies*, 2010. [Obtained from e-mail conversation].

[39] A. Vainshtein, I. Sasson, E. Metz, T. Frost, and P. Pate. Structure-Aware Time Division Multiplexed (TDM) Circuit Emulation Service over Packet Switched Network (CESoPSN). RFC 5086 (Informational), December 2007.

[40] J. Welch and J. Clark. A Proposed Media Delivery Index (MDI). RFC 4445 (Informational), April 2006.

[41] Wikipedia. Category 5 cable. `http://en.wikipedia.org/w/index.php?title=Category_5_cable&oldid=419895462`, 2011. Accessed 23 March 2011.

[42] Wikipedia. Coordinated universal time. `http://en.wikipedia.org/w/index.php?title=Coordinated_Universal_Time&oldid=419842446`, 2011. Accessed 21 March 2011.

[43] Wikipedia. Data structure alignment. `http://en.wikipedia.org/w/index.php?title=Data_structure_alignment&oldid=418803837`, 2011. Accessed 28 March 2011.

[44] Wikipedia. Global positioning system. `http://en.wikipedia.org/w/index.php?title=Global_Positioning_System&oldid=419080861`, 2011. Accessed 22 March 2011.

Appendix A

Code

Listing A.1: Linux patch "fixing hw timestamping in igb"

```
diff -u linux-2.6.37/drivers/net/igb/igb_main.c linux/drivers/net/igb/igb_main.c
--- linux-2.6.37/drivers/net/igb/igb_main.c     2011-02-03 10:02:53.000000000 +0100
+++ linux/drivers/net/igb/igb_main.c    2011-02-03 10:12:40.000000000 +0100
@@ -98,6 +98,7 @@
 static void igb_setup_mrqc(struct igb_adapter *);
 static int igb_probe(struct pci_dev *, const struct pci_device_id *);
 static void __devexit igb_remove(struct pci_dev *pdev);
+static void igb_init_hw_timer(struct igb_adapter *adapter);
 static int igb_sw_init(struct igb_adapter *);
 static int igb_open(struct net_device *);
 static int igb_close(struct net_device *);
@@ -1987,6 +1988,10 @@
        }    #endif
+
+       /* do hw tstamp init after resetting */
+       igb_init_hw_timer(adapter);
+
        dev_info(&pdev->dev, "Intel(R) Gigabit Ethernet Network Connection\n"); /* print bus type/speed/widt
        dev_info(&pdev->dev, "%s: (PCIe:%s:%s) %pM\n",
@@ -2301,7 +2306,6 @@
                return -ENOMEM;
        }
-       igb_init_hw_timer(adapter);
        igb_probe_vfs(adapter);
        /* Explicitly disable IRQ since the NIC can be in any state. */
```

Listing A.2: Linux patch "net: TX timestamps for IPv6 UDP packets"

```
diff --git a/net/ipv6/ip6_output.c b/net/ipv6/ip6_output.c
index 94b5bf1..74d9343 100644
--- a/net/ipv6/ip6_output.c
+++ b/net/ipv6/ip6_output.c
@@ -1115,6 +1115,7 @@ int ip6_append_data(struct sock *sk, int getfrag(void *from, char *to,
        int err;
        int offset = 0;
        int csummode = CHECKSUM_NONE;
+       __u8 tx_flags = 0;
        if (flags&MSG_PROBE)
                return 0;
@@ -1199,6 +1200,13 @@ int ip6_append_data(struct sock *sk, int getfrag(void *from, char *to,
                }
        }
```

```
+       /* For UDP, check if TX timestamp is enabled */
+       if (sk->sk_type == SOCK_DGRAM) {
+               err = sock_tx_timestamp(sk, &tx_flags);
+               if (err)
+                       goto error;
+       }
+       /*
         * Let's try using as much space as possible.
         * Use MTU if total length of the message fits into the MTU.
@@ -1303,6 +1311,10 @@ alloc_new_skb:
                                                                sk->sk_allocation);
                                if (unlikely(skb == NULL))
                                        err = -ENOBUFS;
+                               else
+                                       /* only the initial fragment is
+                                          time stamped */
+                                       tx_flags = 0;
                        }
                        if (skb == NULL)
                                goto error;
@@ -1314,6 +1326,9 @@ alloc_new_skb:
                        /* reserve for fragmentation */
                        skb_reserve(skb, hh_len+sizeof(struct frag_hdr));
+                       if (sk->sk_type == SOCK_DGRAM)
+                               skb_shinfo(skb)->tx_flags = tx_flags;
+
                        /*
                         *      Find where to start putting bytes
                         */
```