technische universität
dortmund

Lehrstuhl für Kommunikationsnetze
Prof.Dr.-Ing. Christian Wietfeld

Diplomarbeit

Development of a Scalable and Distributed System for Precise Performance Analysis of Communication Networks

cand.-Ing. Fiona Klute

Betreuer:	Prof. Dr.-Ing. Christian Wietfeld
Eingereicht am:	8. Oktober 2013

Abstract

This thesis proposes a packet generation and measurement tool for communication networks, called the Lightweight Universal Network Analyzer (LUNA). LUNA is designed to allow per packet analysis, and has been optimized for use with PREEMPT_RT Linux to enhance realtime performance. A timing analysis shows the advantages this provides for precise packet generation. LUNA also works on standard Linux kernels, albeit with reduced precision.

Mimicing realistic network loads for analysis may require the creation of packets with various interval and size patterns. LUNA supports this by offering a flexible packet parameter generation API, allowing the implementation of random distributions or other parameter sources as needed. LUNA features a multi-component design, which includes the core traffic generator, analysis tools, and a remote control system. Separating traffic generation and recording from analysis improves flexibility, and even allows re-evaluating results retrospectively. The remote control system supports distributed network analysis using a single comprehensive control file containing transmission definitions.

Performance analysis shows that it is possible to reach single digit microsecond timing precision in packet generation with LUNA, but it also shows that the degree to which software precision translates into on-the-wire transmissions strongly varies depending on the networking hardware used, even within one type of network. These results highlight the importance of choosing measurement equipment carefully, even when only off-the-shelf hardware is available.

Finally, two example applications of LUNA are presented. One is a throughput analysis of a software defined networking (SDN) testbed for traffic with certain characteristics, the other a round trip time analysis of a real world LTE (4G) mobile network.

Kurzfassung

In dieser Diplomarbeit wird ein System zur Paketerzeugung und Auswertung für Kommunikationsnetze namens „Lightweight Universal Network Analyzer“ (LUNA) entworfen. LUNA kann die erzeugten Datenströme bis auf die Ebene einzelner Pakete untersuchen. Das Echtzeitverhalten von LUNA ist für den Einsatz unter PREEMPT_RT Linux optimiert. Ein Einsatz unter normalen Linux-Kerneln ist ebenfalls möglich, kann aber die Präzision reduzieren.

Die Darstellung realistischer Netzauslastung kann es erfordern, Pakete mit unterschiedlichen Größen und Sendeintervallen zu erzeugen. Zu diesem Zweck stellt LUNA eine flexible Programmierschnittstelle (API) zur Erzeugung von Paketparametern bereit, sodaß benötigte Zufallsverteilungen oder andere Muster nach Bedarf implementiert werden können. LUNA ist in mehrere Komponenten aufgeteilt: den eigentlichen Paketgenerator und -empfänger, Analysewerkzeuge und ein Fernsteuerungssystem. Die Trennung zwischen Paketerzeugung und -messung auf der einen und Analyse auf der anderen Seite erhöht die Flexibilität, und ermöglicht es auch, Auswertungen zu wiederholen. Das Fernsteuerungssystem unterstützt die Untersuchung verteilter System, da auch komplexe Experimente mit mehreren Übertragungen so in einer einzigen Datei konfiguriert und zentral gesteuert werden können.

Eine Leistungsanalyse zeigt, daß LUNA eine Genauigkeit im einstelligen Mikrosekundenbereich erreichen kann, aber auch, daß es stark von der verwendeten Netzwerkhardware abhängt, ob sich diese Genauigkeit auch auf der physikalischen Übertragungsschicht niederschlägt. Die Ergebnisse zeigen, daß der Auswahl von Meßausrüstung große Bedeutung zukommt, auch wenn nur handelsübliche Geräte eingesetzt werden.

Abschließend werden zwei beispielhafte Anwendungen von LUNA vorgestellt: Eine Durchsatzanalyse einer experimentellen SDN („Software Defined Network“, virtualisiertes Netzwerk) Anlage unter Netzlast mit vorgegebenen Parametern, und eine RTT („Round trip time“, Paketumlaufzeit) Untersuchung über ein LTE (4G) Mobilfunknetz.

Release Notes and Copyright License

This is my graduation thesis as it was handed in for grading in October 2013, plus a few editorial corrections. You can find a detailed listing of these changes in Appendix A. The LUNA source code is available on GitHub under the GNU GPL version 3 or later at the following address:

https://github.com/airtower-luna/luna

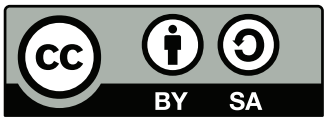
You can also clone the repository directly using the Git command below.

```
git clone https://github.com/airtower-luna/luna.git
```

If you would like to contact me with regard to this thesis, please send your mail to:

fiona2.klute@uni-dortmund.de

Fiona Klute, 2014-05-16



DOI: 10.5281/zenodo.9907

Contents

List of Figures

List of Tables

1 Introduction

In communication networks research, software tools to analyze network behavior play a very important role. However, many popular tools like Iperf[1] lack precision in their results and the ability to create specific traffic profiles. For tools that offer this capability, it is usually unclear to which degree they can actually follow the configuration. However, in mobile networks and other communication systems which use time slots the exact time when a data packet is sent as well as packet intervals can have a significant impact on network behavior. Similarly, packet sizes can influence network performance.

A traffic generator and analyzer that is supposed to offer good performance as well as precision must be designed as a lightweight system. While the generation of complex traffic patterns might require complex code, the core system should be as simple as possible to be able to handle high amounts of traffic.

When analyzing larger networks, configuring each involved host separately becomes impractical, and a remote control system is desirable. At the same time, such a system should be strictly separate from the core traffic generation and analysis code to avoid compromising the *lightweight* design principle.

Although convenient, the common practice of tying traffic generation and analysis together into one program has two disadvantages: It *(i)* limits analysis to calculations that are fast enough to be done in parallel with the actual measurement and *(ii)* makes later analysis of the raw data impossible by discarding it in favor of processed results. Storing the raw data and doing analysis separately could improve performance and make it possible to apply various analysis methods later, including methods which were not available when the measurement was taken.

A traffic generation and analysis system based on the considerations above is proposed in this thesis. This system is called Lightweight Universal Network Analyzer, or *LUNA* for short.

1.1 Structure of this Thesis

Current traffic generation technologies and their problems, as well as background knowledge which may be useful to understand the following chapters are described in Chapter 2.

Chapter 3 presents the core structure, design principles, algorithms, and nomenclature used for LUNA, followed by the implementation of the core generator. All code is designed to be fully IPv6 compatible.

[1] http://iperf.sourceforge.net/

As shown in Section 2.1, using a realtime operating system is likely required to achieve good precision in packet generation. In Chapter 4, LUNA is adapted for optimal performance on the realtime-friendly PREEMPT_RT Linux. The chapter includes an analysis of the effects this has on the precision of the generator.

A traffic generator should be able to create various traffic patterns as required for experiments. In Chapter 5, LUNA is extended to include a generic framework and API which supports the implementation of different methods to generate packet parameters.

Another expansion, modifying the protocol as well, is described in Chapter 6. This makes it possible to create round trip instead of one way transmissions if desired, and measure round trip times.

Chapter 7 describes the design and implementation of a remote control system for LUNA. This makes using LUNA in distributed scenarios convenient by enabling the user to define the measurement setup in only one configuration file and then run the experiment with one command, including collection of results from all involved hosts.

The precision of the generated traffic patterns is a recurring topic throughout this whole thesis. Chapter 8 provides a performance analysis and example applications, which once again include a closer look at LUNA's precision, too.

Finally, Chapter 9 presents potential improvements for future development of LUNA, as well as questions which may become topics of future work.

2 Context and Technical Background

This chapter provides information that is necessary or helpful for understanding the research described in the later chapters. The first section contains an overview of the state of related research and development, the second one defines important terms, and the remainder introduces technical concepts which are important for this thesis.

2.1 State of the Art

Kolahi et al. [1] found that the results of measurements taken using different traffic generators may vary considerably, and that the character of these differences changes with packet size. The same paper also describes the problem that most widely used traffic generators only provide an average load, not precise, configurable packet timings and sizes.

However, Botta et al. [2] say that, strictly speaking, Iperf and similar tools that do not allow such a precise configuration should not be called traffic generators (emphasis added):

> "Examples of the second category are Iperf and Netperf: such tools usually work by sending as much traffic as possible to measure network performance, but they are not strictly considered traffic generators because *they cannot generate specific traffic profiles requested by the operator*."

Their analysis of more precisely configurable generators and how they are commonly used reveals certain problems. Software traffic generators generally suffer from imprecisions caused by the underlying hardware and operating system. Lack of precision can impact repeatability and comparability of experiments, leading to wrong assumptions and conclusions. Thus, it is important to verify the characteristics of the *actual output* a traffic generator creates.

When Botta et al. [2] configured the tested traffic generators to send 1950 packets per second, which should lead to inter departure times around 513 µs, they found that some of them showed peaks of IDTs at around 4 ms (!) and below 10 µs. Exponential IDT distributions showed significant deviations as well. Results improved with a polling system for timeouts, however, Botta et al. [2] used Linux 2.6.15, which was already severely outdated when they performed their study. Modern Linux systems using High Resolution Timers should show significantly better performance (see Section 2.3.2 below).

On the plus side, their analysis does reveal some software design decisions (especially polling for timeouts) in the traffic generators considered that seem rather poor in the light of these somewhat newer developments, although they may have been the best available methods when the decisions were made.

Surprisingly, Botta et al. [2] did not consider realtime operating systems, although Paredes-Farrera et al. [3] already mentioned their importance for precise traffic generation and the resulting measurements back in 2006. Current tools should make the best possible use of the available timer APIs and also be developed with use on realtime operating systems in mind.

Flowgrind is a performance measurement tool for TCP [4]. While the author does not agree with some design decisions made by the Flowgrind developers, the goals are very similar to the tool presented in this thesis in that they, too, aim to precisely evaluate the transmission characteristics of data flows. However, Flowgrind uses TCP, which is by definition bi-directional (although often with asymmetric data rates) with some unavoidable overhead, while LUNA uses UDP.

2.2 Definitions of Terms

2.2.1 Precision

One goal of this thesis is to understand what measurement precision is possible while working on the software level. In principle, properties of specific networking hardware cannot be observed directly from software, although comparisons with other hardware or pure software networking (loopback, virtual networks) may allow for some conclusions.

Botta et al. [2] define "accuracy" of traffic generation as

> "[...] the measure of the difference between the requested traffic profiles (i.e., *imposed rates* and *statistical distributions*) and those actually generated, [...]"

and correctly point out that this accuracy must be evaluated to reach meaningful conclusions from measurements. Throughout the following chapters, experiments to evaluate the precision of the system under development will be a recurring topic. *Absolute* precision is impossible to reach with a software traffic generator, but such experiments should help users judge the precision of results obtained with the traffic generator in question.

2.2.2 Inter Send Times and Inter Arrival Times

Inter Send Time (IST), also called Inter Departure Time (IDT), is the time that passes after a sender in a packet based network has sent one packet until the next one is sent. If packet n is sent at $t_{send(n)}$, and packet $n+1$ at $t_{send(n+1)}$, the inter send time is defined as:

$$IST = t_{send(n+1)} - t_{send(n)} \tag{2.1}$$

The definition of the Inter Arrival Time (IAT) on the receiving side is similar:

$$IAT = t_{arrive(n+1)} - t_{arrive(n)} \tag{2.2}$$

Statistical network models frequently assume that ISTs follow certain statistical distributions, for example [5, 6, 7], so the ability to precisely mimic the distributions assumed by such models should be useful in experiments aimed at verifying them.

It should be obvious that any experiment with the goal of precisely measuring IATs requires a packet generator that can create packets with precise ISTs [2]. Any imprecision in the packet generation necessarily increases the overall measurement imprecision, possibly leading to wrong assumptions concerning traffic properties.

In practice, both sending and receiving a packet does not happen at a singular moment in time, but takes some amount of time. However, measuring these times in a meaningful way is hardly possible without examinations at the physical network layer, which would require attaching specialized measurement equipment to the appropriate signal cables for Ethernet, and similarly complex setups for other types of networks. The best a software traffic generator can do is to acquire arrival times from the underlying operating system's network stack.

Inter Arrival Times can be characteristic for certain network types, as protocol properties and the behavior of certain network components may influence them (e.g. transmission time slots or packet aggregation). In [8], statistical values based on IATs are used to classify networks.

2.2.3 Round Trip Time

The Round Trip Time (RTT) in a communication system is the time between the departure of a message until an answer arrives. Ignoring potential measurement imprecision on the client, the round trip time consists of three components, which can be seen in Figure 2.1:

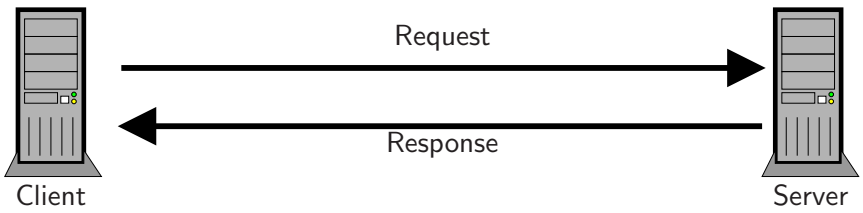


Figure 2.1: Components of a round trip time measurement

1. The transmission time of the request,

2. the time it takes for the server to process the request and send a response,

3. and the transmission time of the response.

None of these are individually observable on the client, and it would be wrong to assume that the one-way transmission time is simply half the round trip time. Even if the transmission times in either direction are equal — which is not necessarily the case, depending on the type of network [9] — that assumption would ignore the processing time on the server and thus lead to estimates above the actual transmission times.

When analyzing a specific *application* the processing time in the remote system may be relevant, however, when the *network* itself is the target of the analysis the influence of data processing must be minimized to optimize measurement precision. Thus, a network analysis tool providing round trip time measurements must be designed to minimize the processing time. It may be possible to observe the processing time on the server using local profiling tools, but since processing times will vary based on hardware performance and possibly system load, such measurements would need to be done individually for each measurement setup.

The RTT is a good measure for the "overall responsiveness" of a communication network, but is not necessarily correlated with data throughput.

2.3 Realtime in Operating Systems

As described by Paredes-Farrera et al. [3], high precision in packet generation requires a realtime operating system. However, there are different definitions of realtime in computing. Precise time measurement is a closely related topic and required as well.

2.3.1 Hard and Soft Realtime

> "A time-constraint is called hard if not meeting that constraint could result in a catastrophe." (Kopetz [10])

Realtime constraints come in two flavors: *soft* and *hard* realtime. Hard realtime constrains as described by Kopetz [10] can only be met by special realtime operating systems with predictable timing behavior [11, p.182]. Soft realtime, on the other hand, is essentially a best effort based timing with high quality, and can be achieved with appropriate extensions to normal operating systems, as shown in Chapter 4. Soft realtime should be sufficient for most measurement systems, as long as the user can estimate the limits of precision.

2.3.2 High Resolution Timers

Traditionally, timings inside the Linux kernel and therefore user space timers as well were measured based on "jiffies". Jiffies are events (ticks) from a regularly firing timer interrupt. At each tick, the timer subsystem would check if a timer had expired, which means timer precision was limited to tick resolution [12, sec.1].

Timing based on jiffies is insufficient for applications which require precise timing. To change this, the *hrtimers* framework [12, sec.4]., which allows the use of High Resolution Timers (HRT) instead of tick based timers, was added to the main line kernel in Linux 2.6.16 (published March 2006). HRTs can create timer interrupts whenever needed [13], limited only by the capabilities of the hardware used. Dynamic ticks, which make it possible to avoid jiffies altogether and use HRT interrupts only, were introduced in Linux 2.6.21 (published April 2007) [13]. With HRT and dynamic ticks, the hardware capabilities and unavoidable software overhead are the only limitations to timer precision.

Using high resolution timers does not require changes in user space per se, but user space software should make sure to use current C library functions. It is unknown to the author why Botta et al. [2] were still using Linux 2.6.15 in their paper published in 2010, while newer Linux kernel versions that might have allowed more precise measurements were available for years prior to their study.

For judging the results of Chapter 4, it may be relevant that Gleixner and Niehaus [12] have compared a Linux 2.6.16-hrt[1] kernel with 2.6.16-rt[2], and found that the difference in precision is small at low load, while PREEMPT_RT can shine at higher loads.

2.4 Memory Management

When writing code for realtime systems as described in Chapter 4, proper memory management is even more important than in generic programming. Instead of slowing the process or the whole system down, bad memory management may completely break realtime behavior. The following basics are required to understand the descriptions in Section 4.2.

The most important sections of memory associated with a process are the *heap* and the *stack*. Other memory segments containing things like global variables and the actual executable code exist, but are of no concern for runtime memory management because of their fixed size. Classically, stack and heap were growing towards each other, starting at the highest stack address and lowest heap address. The operating system must prevent them from growing into each other, and modern operating systems often employ more advanced addressing schemes. This is shown in Figure 2.2.

Robbins [14] explains the stack as follows, with a reference to C programming:

[1] Vanilla kernel with added patches for high resolution timers
[2] Linux PREEMPT_RT kernel, includes HRT code, for more on PREEMPT_RT see Chapter 4.

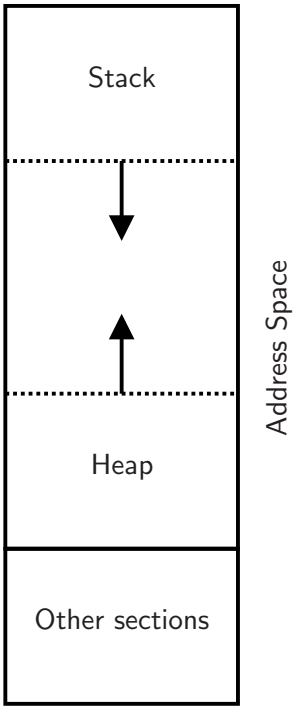


Figure 2.2: Process Data Memory in Linux (simplified)

> "The stack segment is where local variables are allocated. Local variables are all variables declared inside the opening left brace of a function body (or other left brace) that aren't defined as `static`."

The stack grows whenever a function is called by the amount needed for that functions local data, and shrinks accordingly when the function returns. Note that each thread in a process has its own stack, while the heap is shared. A fixed amount of memory to be used for the stack is allocated at thread start, Linux on x86_64 uses 8 KB stack memory [15].

On the other hand, the heap does not have a fixed size, although the operating system may impose limits. Robbins [14] writes:

> "The heap is where dynamic memory (obtained by `malloc()` and friends) comes from. As memory is allocated on the heap, the process's address space grows, as you can see by watching a running program with the `ps` command."

In C, heap memory is dynamically allocated by the appropriate functions, and must be released by a call to `free()` as well. This means the programmer is fully responsible for proper heap memory management.

3 A Lightweight and Fast Traffic Generator

This chapter describes the development of the core software necessary for the work described in the later chapters. The most fundamental requirement for LUNA is the ability to take precise measurements. In turn, that requires fast and precise packet sending.

The traffic generator and receiver code developed at this stage therefore aims to be as simple as possible, details are described below. It was written with later use in a realtime system in mind, but will operate on any Linux system, with whatever precision the kernel allows. Dynamic packet sizes and intervals were not implemented at this stage, but the constant values are configurable at program start. Similarly, the system will be designed with distributed use in mind, but not yet optimized for it. The most important issue in that regard is keeping the command structure or configuration simple. LUNA is implemented in C.

All components have been designed to be fully IPv4 and IPv6 capable[1]. By default, both protocols will be active with preference in name resolution depending on the underlying system's settings, but the user can restrict a client or server to use either IP version exclusively by specifying −4 or −6 on the command line.

In this chapter, I will discuss only the most basic scenario with one data stream (a transmission) from one client to one server as shown in Figure 3.1. More advanced and distributed use will be discussed in later chapters.

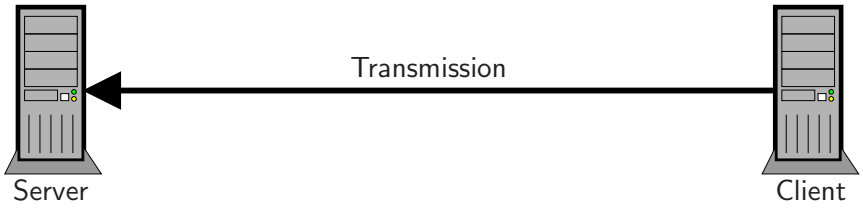


Figure 3.1: Use of client, server and transmission in the LUNA context

3.1 LUNA Protocol

The protocol is based on UDP [17], mainly because the connection setup and acknowledge packets of a connection-based protocol like TCP would create additional load on the network and thus might

[1] Since IPv9 has reached "the end of its useful life", as described in RFC 1606 [16], adding support for it has been deemed unnecessary.

influence the results. Another requirement is for the protocol to take as few bytes as possible, so the user can vary packet sizes in a broad range. In the current version, the only required data is a four byte sequence number, stored right after the UDP header, which can be used to detect lost packets and reordering. Sequence numbers start at zero. If the user requested a packet size larger than four bytes, padding is added as necessary to reach that size. The resulting packet structure is shown in Figure 3.2.

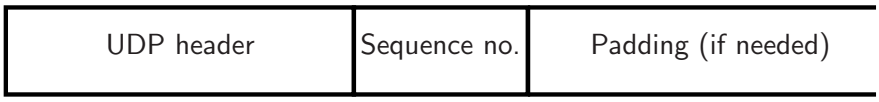


Figure 3.2: LUNA protocol

Users should note that UDP packets may be much longer than Ethernet frames commonly are. Requesting a packet size above the path maximum transmission unit (MTU) is theoretically possible, although not advisable. Linux will return the error `EMSGSIZE` when a UDP packet is too large to be transmitted in one IP packet, unless explicitly configured otherwise [18], however, packet fragmentation may silently occur on the IP layer if a valid IP packet is too large for the underlying transport protocol (e.g. Ethernet). For network performance measurements, it is usually sensible to avoid packet fragmentation whenever possible. If only one of the fragments were to be lost, the whole packet would be lost, so the risk of packet loss would increase exponentially with the number of fragments. Additionally, successfully transmitted fragments would be rendered worthless and their transmission time lost, reducing the effective data throughput as measured on the UDP layer, where LUNA operates, leading to wrong conclusions concerning the network's performance. Thus, the only case in which selecting UDP packet sizes above the path MTU of the network might be desirable is an experiment directly aimed at studying the effects of fragmentation. In the current implementation as described below, the receive buffer of the LUNA server limits the maximum UDP payload to 1500 bytes, although it could easily be enlarged if desired.

It is important to note that the protocol does not provide any way for the client to verify if the packets actually arrived at their destination. The server can use the sequence numbers to check for lost packets, but it cannot detect completely failed transmissions or missing packets at the end of a transmission.

3.2 LUNA Client

The client is responsible for generating packets and sending them towards the server. The lower limit for possible inter send times (IST) should be determined only be the capabilities of the underlying system, so using the smallest possible amount of CPU time per packet is critical. At the same time, the best available clocks have to be used for packet timing. The client's process flow is shown in Figure 3.3 and described in the paragraphs below.

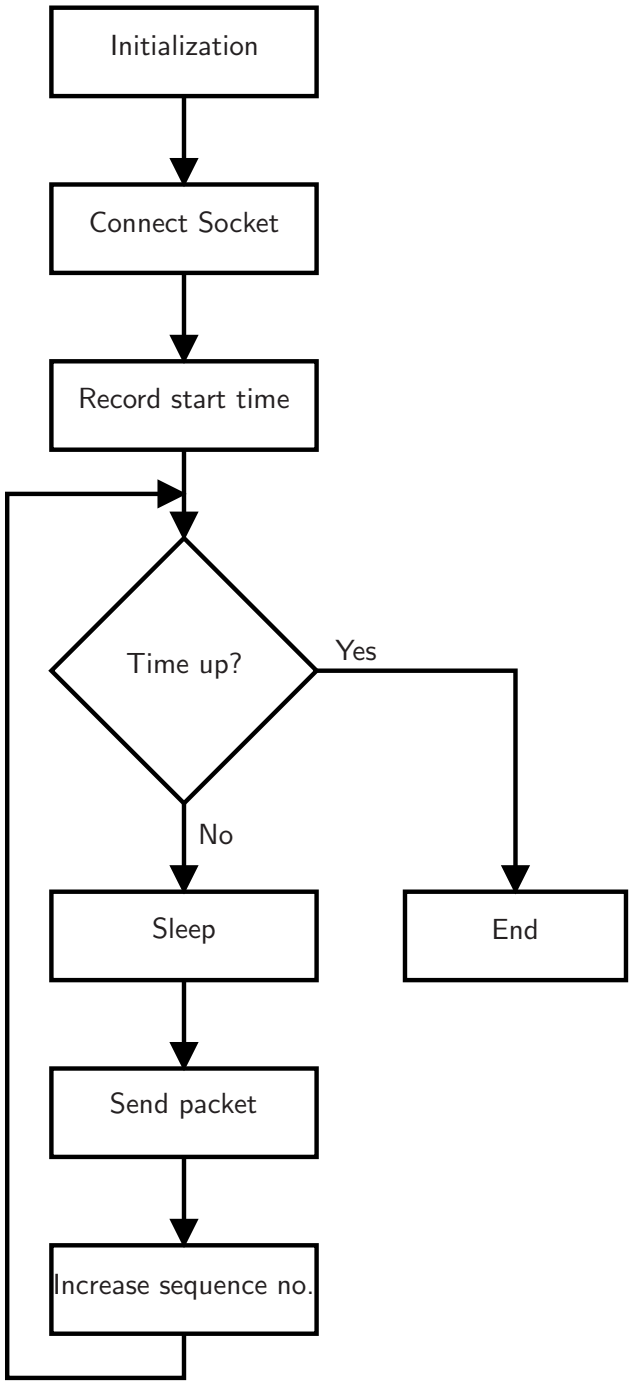


Figure 3.3: LUNA Client process flow diagram

Initialization refers to various tasks that must be done before any network activity can take place. It includes parsing command line options, resolving the server host name if necessary, and turning the server's IP address into the format required for socket creation. If the server is specified as an IP address, the address format will dictate the address family to use. Client and server share the source code for initialization. The dedicated client code receives the destination address, the requested interval between packets (IST), the packet size and the time it should keep operating.

To be precise, the function `getaddrinfo()` [19] is used to acquire the set of binary address data necessary for socket creation from a textual IP address (IPv4 or IPv6) or a hostname (which requires address resolution) and the port number provided by the user. `getaddrinfo()` does not return a single set of data, but instead a list structure, containing multiple data sets if available, for example in case a hostname can be resolved to multiple IP addresses. The client tries to use each element of the list in order until the socket has been set up successfully, which is exactly the way the list is meant to be used.

Creating and configuring the socket for sending is the next step towards starting the transmission. The POSIX `connect()` function is used to set the destination address. Explicitly connecting the socket is not technically required for UDP, but it saves passing the destination address along with every packet and thus simplifies the code. A buffer of the requested payload size is allocated as well.

3.2.1 Overall operating time

In the first prototype code, the client sent a certain number of packets rather than running for a certain amount of time. At this point, the wait between sending two packets was the only place where precise timing was needed (see Section 3.2.2).

Once absolute timings were used for packet intervals, the overall time could easily be controlled based on the same absolute clock. For this purpose, the end time is calculated before starting the send loop. It is important to note that the `struct timespec`, which is used to store times at nanosecond precision, contains separate values for seconds and nanoseconds, both of which must be taken into consideration when doing any operation on it.

In a first step, the end time was compared to the next packet's timing as the loops exit condition. However, this will lead to trouble when the client cannot send packets as fast as requested, because planned packet timings are based on the scheduled time of the previous packet, and not the time when it was actually sent. In particular, a requested IST of zero will lead to an endless loop. The solution is to get the current time from the same monotonic clock after sending a packet, and then using that time as the current time when checking the loop condition.

3.2.2 Precise timing for packet intervals

The basic function for making a process wait for a sub-second time is `nanosleep()` [20]. The problem with `nanosleep()` is that it is relative. When the body of the core loop contains the statements to prepare and send a packet followed by a `nanosleep()` call, the actual duration of one loop will not be the time passed to `nanosleep()`, but rather the sleep time plus the time preparing and sending a packet takes. The second part will vary depending on the hardware used, and may also vary between loop iterations depending on other activity (including the OS) on the underlying system, so it is not practical to reduce the sleep time to account for this.

A better approach is to use the Linux kernel's high resolution timers[2] [12] in combination with absolute timers on a monotonic clock. The monotonic clock is different from the realtime clock in that it cannot be modified by other applications (like synchronization via NTP), which eliminates a potential source of timing errors. A sleep call using an absolute timer causes the process to sleep until the clock reaches the given time, rather than a fixed duration.

The resulting timing algorithm, shown in Figure 3.4, is as follows: Before starting the send loop, the current time from the monotonic clock is loaded into a variable called `nexttick`. In each iteration of the send loop, the sequence number is incremented and written to the send buffer in network byte order, and the requested interval is added to `nexttick`. Then a call to `clock_nanosleep()` [21] using `nexttick` as an absolute target time lets the process sleep until it is time to send the next packet. The core points (the gray boxes in Figure 3.4) are that the sleep ends at an absolute time, so the sleep *duration* is automatically adjusted according to the time needed to prepare the next packet, and that the packet is sent right after the sleep call returns.

Once the loop ends, dynamically allocated memory is freed before the program exits.

3.3 LUNA Server

On the server side, the most important issue is getting precise measurements of packet arrival times. In the initial planning stage, multiple ways to minimize the delay between receiving a packet and creating a timestamp were considered. The problem is that receiving the packet in user space takes time and must happen after the kernel has received it. Additionally, under a high network load the process might not be able to read packets from the socket as fast as they arrive, especially considering that some processing is required after receiving. All these delays could significantly skew the result.

One idea to minimize such inaccuracies was to use a multithreaded approach: Either by having one thread read from the socket as fast as possible and then distribute the packets to handler threads, or have multiple threads compete for a mutex that would allow read access to the socket and have them read from there, effectively creating a load based circulation. In both cases, however, the time the packet needs from kernel to user space would be included in the measurement, and context switches between threads take time as well, which would offset the result even more.

A better way would be to acquire an arrival timestamp directly from the kernel, and indeed the Linux kernel offers an ioctl for this purpose: `SIOCGSTAMP`. `ioctl()` is a function that can perform various tasks on "special files" [22], in this case the network socket. Ioctl calls on a socket require three parameters: the socket, the ioctl identifier, and a pointer to memory for the result [23]. With `SIOCGSTAMP` as the ioctl identifier, the arrival time of the last packet passed to user space will be written to the specified memory as a `struct timeval`. Because the timestamp is created and stored along with the packet by the kernel, it will not change depending on when user space sends the request, as long as user space

[2] Supported by any modern kernel version as of the time of this writing, activated at build time with `CONFIG_HIGH_RES_TIMERS`. See Section 2.3.2 for details.

has not read any other packet from the same socket in the meantime. This can easily be guaranteed if only one thread is allowed to read from the socket. Using both the kernel arrival time and measuring the arrival time in user space makes it possible to calculate the delay between them, this is discussed in detail in Section 3.3.2.

3.3.1 Server process flow

The server's process flow is shown in Figure 3.5. As already described in the client section, initialization refers to various tasks that must be done before any network activity can take place. The first server specific task is binding the listening socket. Unless IPv4 only mode is selected, the server socket will use IPv6, which by default includes support for mapped IPv4 addresses. If IPv6 only mode is selected, the socket option `IPV6_V6ONLY` [24] is set to prevent receiving IPv4 packets. Before actually reading from the socket, various buffers for receiving and processing packets must be allocated.

The main receive loop is implemented as a `while` loop with a loop variable that is initialized to true. This way the server runs for an unlimited time, but can be cleanly shut down using a handler for `SIGTERM` that sets the loop variable to 0 when called. The main steps inside the loop are:

1. Receiving the packet from the socket,
2. calling `gettimeofday()` [25] to get the user space arrival time,
3. requesting the kernel arrival time using the `SIOCGSTAMP` ioctl as described above,
4. and finally writing the result to the standard output.

Recording the user space arrival time is required for the evaluation in Section 3.3.2 only and has been removed afterwards for better performance, although it is available as a compile time option if needed.

The output is *per packet*, without further processing. The transmission as a whole can be analyzed later using Octave as described in Section 3.4, or other tools if the user chooses to do so. Because of this, the server should be able to write the data in a machine readable format. A command line option (`-T`) has been introduced, which leads to output in a `TAB`-separated table format.

For example, Listing 3.1 shows the first few lines of the data behind Figure 3.7. Times (the first two columns) are in Unix time, extended to microsecond precision.

Listing 3.1: Example of `TAB`-separated output from the LUNA server

```
1 # ktime utime source port sequence size
2 1368543912939397 1368543912939463 ::1 33606 0 4
3 1368543912940373 1368543912940436 ::1 33606 1 4
4 1368543912941372 1368543912941399 ::1 33606 2 4
```

Metric	Value
Average	27 µs
Median	24 µs
Upper limit	3765 µs
Lower limit	12 µs
Standard deviation	49 µs

Table 3.1: Metrics of the recorded differences between kernel and user space arrival times

When redirecting the output from standard output into a file, the file was frequently empty when stopping the server process. This was due to buffering and unclean exit (`Ctrl+C` on the command line). There are two possible solutions: The one actually used is ensuring that buffers get flushed on proper exit (`SIGTERM` as described above), alternatively it would have been possible to force the output stream into line buffered mode, but that might incur a performance penalty.

3.3.2 Delay between kernel and user space

To evaluate the timing difference between kernel and user space, a test using the timing measurements described in the previous section was done. The test covers one transmission with 30000 packets (packet interval of 1 ms, 30 s duration), using IPv6 over the loopback interface of a Laptop with a dual core Athlon processor, running Linux kernel 3.8. No packets were lost.

Transmitting via loopback (`lo`) means that the packet never leaves the host. As shown in Figure 3.6, the client sends the data to the kernel network stack, addressed to the loopback interface. As soon as the interface receives the packet (left arrow, t_0), it becomes available on the server's listening socket. This marks the kernel arrival time. The server then reads the packet from the socket (right arrow), the result of the `gettimeofday()` call directly after that marks the user space arrival time (t_1). Taking this measurement via loopback eliminates any possible influence from network hardware or hardware drivers.

The results can be seen in Figure 3.7 as a histogram showing the distribution of the measured differences. Note that the x-axis is shown with an upper boundary at median plus double standard deviation, with the rightmost bar including all values beyond that. The standard deviation and other metrics can be seen in Table 3.1.

While the delay is in many cases between 20 and 30 µs, the peak around 40 µs and the large standard deviation show that an arrival time measurement in user space would not be able to provide the desired precision. An even bigger problem, however, is that far longer delays are possible. The recorded maximum was 3765 µs, which is almost four times the intended time between the arrival of two different packets. These results show that it is imperative to use the arrival times measured by the kernel for precise analysis.

3.4 Post-processing in Octave

As described in the server section (Section 3.3), the server performs only packet-wise processing and prints the results for later analysis. In principle, a user can do such an analysis using any tool they wish, the approach used by the author was to perform calculations and create figures in GNU Octave[3]. The Octave scripts written for this purpose are part of the LUNA source code and together form LUNA's analysis component.

The tasks implemented in this stage are:

1. Checking sequence numbers for lost and reordered packets
2. Analyzing the delay between kernel and user space arrival times (see Section 3.3.2)
3. Analyzing the packet inter arrival times (IAT)

Combined with packet sizes, IATs can be used to calculate the overall throughput of a network. The analysis script calculates IATs based on kernel arrival times only to provide the best possible accuracy as shown in Section 3.3.2.

A way to distinguish multiple connections was not implemented at this stage.

3.4.1 Sequence Numbers

In a first step, the sequence numbers are only checked for completeness: According to the protocol definition (Section 3.1), sequence numbers start at zero, so the total number of packets in a connection should be the highest sequence number plus one. This approach cannot catch lost packets at the end of a transmission: If 1000 packets were sent and the last two get lost, the highest sequence number recorded by the server will be 997, from which the script will calculate an expected number of 998 packets. The assumed overall number of packets and the number of lost packets are written to standard output.

Reordered packets are detected by checking if any sequence number is lower than the previous one, and if yes a message with the sequence number is written, but no further handling takes place. At the same time the system checks for duplicate sequence numbers, which should not occur unless there is an error in the traffic generator.

[3] http://www.gnu.org/software/octave/

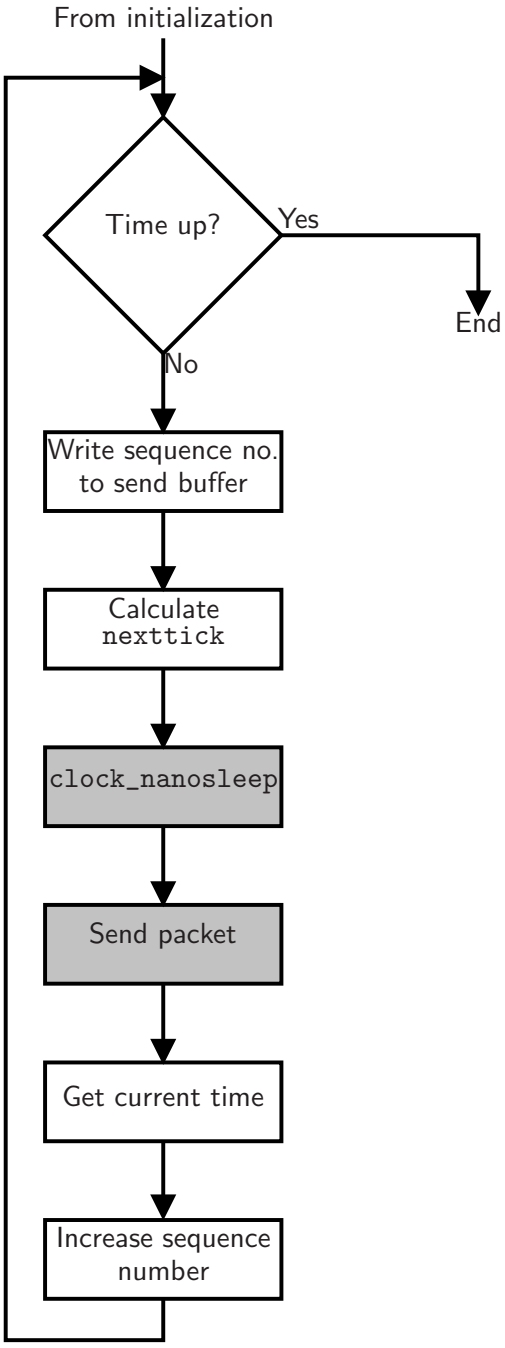


Figure 3.4: Flow diagram of the send loop in the client, a more detailed version of the loop part in Figure 3.3

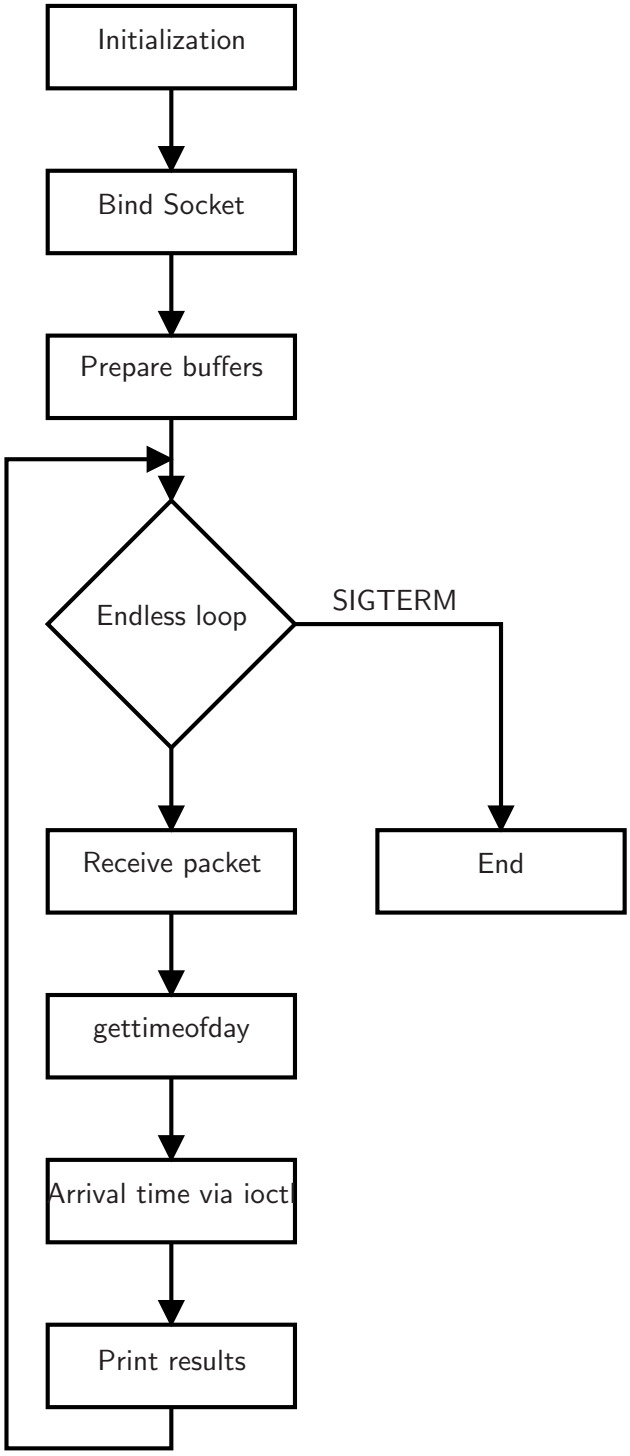


Figure 3.5: LUNA Server process flow diagram

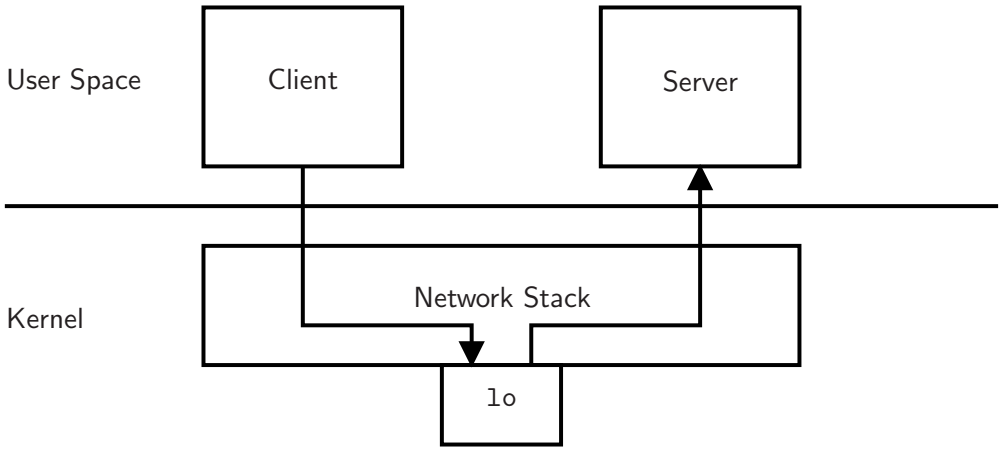


Figure 3.6: LUNA transmission via loopback interface

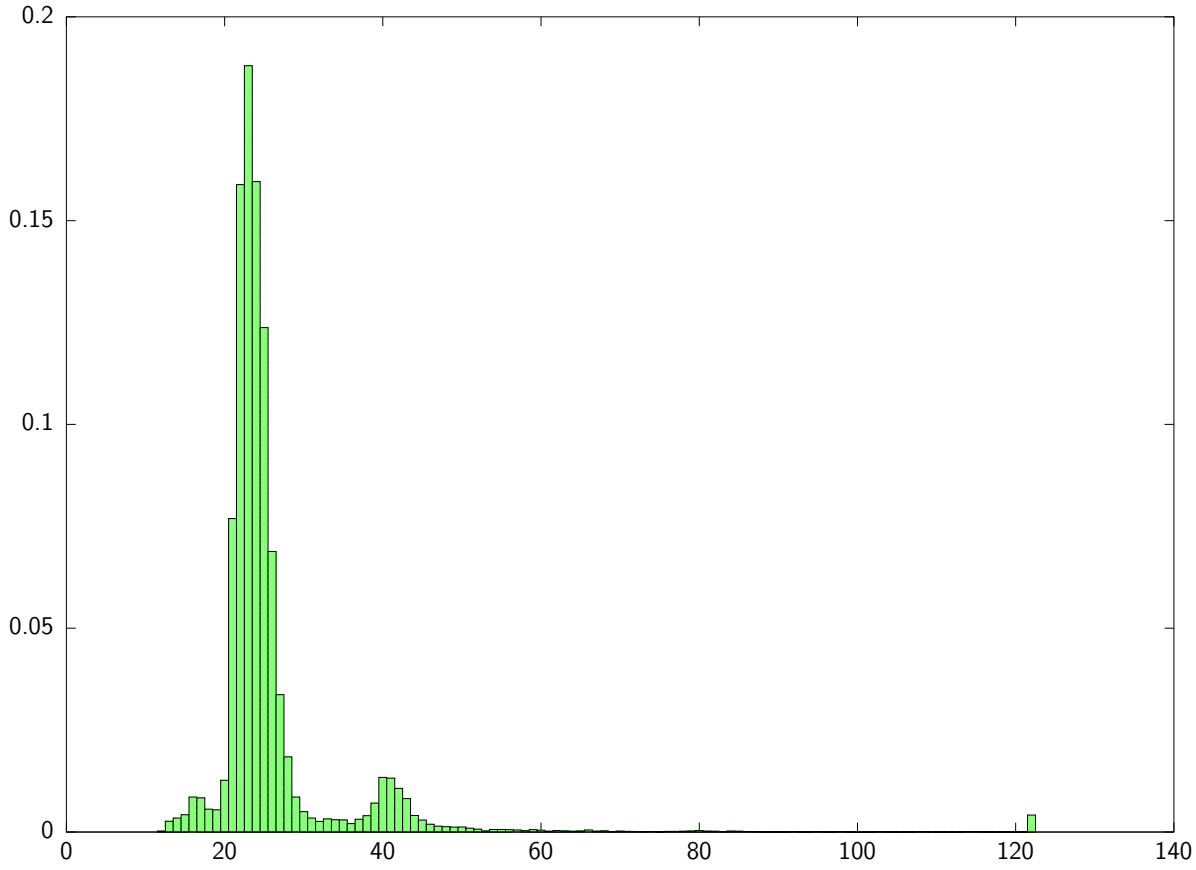


Figure 3.7: Distribution of differences between kernel and user space arrival times in µs

4 LUNA on PREEMPT_RT Realtime Linux

As shown by Paredes-Farrera et al. [3], a properly used realtime operating system can improve the accuracy of packet generators. In this chapter, LUNA as described in Chapter 3 will be optimized for realtime operation and the influence of using a realtime operating system on the traffic generator will be evaluated.

There are two approaches to achieving realtime performance on Linux systems, both described in [26]: One is to run the Linux kernel as a preemptible process under a microkernel or hypervisor, which can provide the required realtime scheduling. The problem with this approach is that realtime processes need to run directly under the realtime kernel instead of Linux, and thus need to be written and compiled for that realtime kernel, limiting portability (Figure 4.1). Essentially this provides an environment where realtime processes can be controlled using Linux, but are not really running under it. Marwedel [11] calls this type of realtime OS a "hybrid system".

The other possible approach is to improve the realtime behavior of the Linux kernel itself, which would allow programs written and compiled for the normal Linux interfaces and libraries to take advantage of realtime features (Figure 4.2). This is what PREEMPT_RT does.

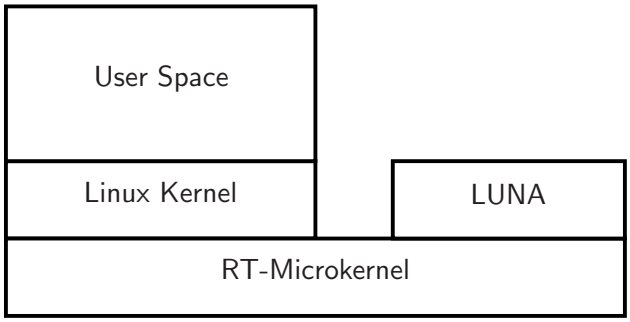


Figure 4.1: Linux System with RT-Microkernel

4.1 Realtime Linux with the PREEMPT_RT patchset

McKenney [27] describes the fundamental concept of PREEMPT_RT Linux as follows:

> "The key point of the PREEMPT_RT patch is to minimize the amount of kernel code that is non-preemptible, while also minimizing the amount of code that must be changed in

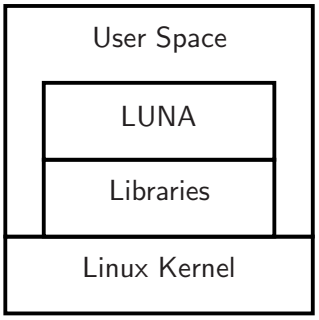


Figure 4.2: Standard Linux System

> order to provide this added preemptibility. In particular, critical sections, interrupt handlers, and interrupt-disable code sequences are normally preemptible."

This should explain the name PREEMPT_RT Linux: It aims to provide realtime behavior by enabling preemption of almost all kernel code. Userland code does not need to be changed to work with a PREEMPT_RT kernel. Software needs to be written carefully to avoid creating latencies of its own as described in Section 4.2, but the same source code and even binaries can be used on a standard Linux kernel, albeit without the benefits of realtime scheduling. However, while these measures serve to avoid scheduling latencies as much as possible, PREEMPT_RT cannot guarantee maximum delays, so it is classified as a soft realtime operating system (see Section 2.3.1).

While based on the mainline Linux kernel, PREEMPT_RT is not (yet) fully included in it and distributed as a patchset[1] instead, however some Linux distributions provide precompiled kernels with PREEMPT_RT. In Debian, which was used in my realtime test bed, the RT kernel package for the AMD64 architecture is called `linux-image-rt-amd64`.

This flexibility and ease of use provide the motivation to use PREEMPT_RT instead of a hard realtime system. Relying on a special kernel would make it impossible to integrate LUNA into normal Linux systems, forcing users to employ special measurement equipment instead. Users who do not require microsecond precision can use LUNA on their standard Linux systems, while setting up a system with LUNA on PREEMPT_RT Linux can be done quickly if needed.

4.2 Adapting LUNA for Real-Time Operation

The PREEMPT_RT documentation describes the considerations that should be made when writing an application that needs good realtime behavior [28, 29]. The following sections describe how they were applied to LUNA.

[1] A patchset is a file or set of files that describes changes to one or more source code files in a machine readable format, so they can be applied automatically.

4.2.1 Memory Usage

Page faults of any kind will cause unexpected delays with unknown durations. Avoiding them is therefore necessary to provide good realtime behavior. There are two kinds of page faults: major page faults and minor page faults. This section requires some knowledge of memory management fundamentals, as described in Section 2.4.

Major Page Faults

Major page faults occur when the process tries to access a memory page that has been swapped out of RAM. In this case the kernel must reload the page from swap. Swap space is typically located on a hard disk, which means the access will cause a huge delay. The only reliable way to prevent this short of disabling swap system-wide is to lock the process memory in RAM, disabling swap for this process only. This can be done using the `mlockall()` system call with `MCL_CURRENT` and `MCL_FUTURE` flags set [30]. Locking the memory requires the `CAP_IPC_LOCK` capability (see Section 4.2.3).

Minor Page Faults

Minor page faults occur when virtual memory has been allocated by the process, but the kernel has not assigned physical memory by the time the memory access happens. The kernel then has to allocate physical memory before the access can complete. The delay will likely be shorter than that caused by a major page fault, but might still disrupt realtime behavior. After the stack has been forced to memory by `mlockall()`, this should only be of concern for dynamically allocated memory.

The easiest way to avoid minor page faults is writing to every memory page before using it in a realtime context. More advanced methods, like allocating a pool of heap memory in advance that can be used for dynamic allocation later do exist as well, but are not needed in LUNA. Note that one memory area allocated on the heap may span multiple pages, so multiple writes with different offsets to the same heap variable may be required. The page size can be determined by calling `sysconf(_SC_PAGESIZE)` [31].

All of these measures must take effect before entering the realtime sections of the program to be useful. For example, writing to a memory page that has not yet been assigned to physical memory will always cause a minor page fault. The point is to intentionally cause these faults during process initialization, rather than having them occur during realtime operation.

Error Checks for the Memory Management

The success can be verified by accessing process resource usage information collected by the kernel. The data structure `struct rusage` as returned by the `getrusage()` function contains the number of major and minor page faults caused by the process in its fields `ru_majflt` and `ru_minflt`, respectively

[32]. Storing this data right before entering a realtime section and right after leaving it makes it possible to check if any page faults occurred during the realtime section, and warn the user if there were any. This serves as a safety measure to ensure the memory management measures have been implemented properly.

The Localtime Problem

Indeed, this error check revealed one problem on the server side, where the receive loop always caused one minor page fault total, irrespective of the number of loop passes. Additional measurements taken during various parts of the loop indicated that the very first call to `localtime()` was the culprit. `localtime()` turns time structures as used by kernel and C library into printable ASCII strings with a configurable format [33]. An unusual property of `localtime()` is that it does not require memory for the string to be passed to the function. Instead, it writes the string to an internal buffer and returns a pointer to that buffer. The observed page fault was likely caused by the allocation of memory for the buffer.

The obvious solution seemed to be using `localtime_r()` instead, which does write to a caller-supplied buffer. However, this did not prevent the page fault. Presumably, `localtime_r()` works similar to `localtime()` internally and only copies its result to the caller-supplied buffer in the end. Thus, the final solution was to add a call to `localtime()` before the realtime section, forcing the buffer allocation before it could cause problems.

4.2.2 Real-Time Scheduling

The Linux kernel offers multiple scheduling policies, as described in the `sched_setscheduler()` man-page [34]. The important distinction here is the one between realtime and non-realtime scheduling, the latter being the default. Processes which need good realtime behavior should acquire a realtime scheduling policy and priority. Realtime priorities are distinct from the normal "nice values" (see [35]). Non-realtime processes always have a realtime priority of zero, and a runable processes with a higher realtime priority will always preempt one with a lower priority.

Two scheduling policies are available for processes with realtime priority: SCHED_FIFO and SCHED_RR [34]. The FIFO scheduler guarantees that a realtime process that has been preempted by a process with higher RT priority can resume work before other processes with the same priority, while a process which blocks (e.g. because it is waiting for data to arrive on a socket) on its own is moved to the end of the process queue. The RR (round-robin) scheduler behaves similarly, but ensures that all processes with the same RT priority get their fair share of processing time by automatically preempting a process if it has been running longer than a certain time quantum. The mechanism to determine this time quantum depends on the Linux kernel version in use [36]. On a dedicated measurement system conflicts with other applications should be rare, but if multiple instances of LUNA are running on the same system,

it is highly desirable that processing time is split equally in case of insufficient CPU resources. Because of this, LUNA uses the SCHED_RR policy.

A process can change its own realtime policy and priority using the `sched_setscheduler()` function [34], however, elevating the priority requires the `CAP_SYS_NICE` capability (see 4.2.3). Linux provides a priority range of 1 to 99, while POSIX requires a minimum difference of 32 between highest and lowest realtime priority. In most of the tests described in this chapter, LUNA uses a priority of 21. A more fine-grained approach is described in Section 4.3.4. It is important to note that realtime scheduling is available on both standard and PREEMPT_RT Linux systems, but on standard systems user space processes cannot preempt the kernel, whatever their priority may be.

Developers should keep in mind the following warning from the `sched_setscheduler()` manpage [34] when working on code running with a realtime priority:

> "Since a nonblocking infinite loop in a process scheduled under `SCHED_FIFO` or `SCHED_RR` will block all processes with lower priority forever, a software developer should always keep available on the console a shell scheduled under a higher static priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected."

If such a block occurs without the developer having taken proper precautions in advance, a hard reset of the machine in question may be the only way to break it. Other ways to limit resource usage are explained in the manpage.

4.2.3 Setting Capabilities

Traditionally, Unix style operating systems differentiated between `root` (also called the super user) and unprivileged users. Privileged operations, like locking memory, assigning one's own processes a higher priority, or binding privileged network ports required the process to run as `root`. However, many tasks require only very few privileged operations, and running them as `root` runs the risk of massive damage in case of errors in the code or even vulnerabilities that can be abused for malicious purposes.

Capabilities were introduced so processes can receive more fine grained privileges [37]. Each capability grants a process holding it certain permissions, without general super user rights. LUNA needs two capabilities: `CAP_SYS_NICE` to acquire realtime scheduling priority, and `CAP_IPC_LOCK` to lock its memory in physical RAM.

While possible, running LUNA as root is not recommended. Instead, only the required capabilities should be granted. The easiest way to do that is to assign them to the LUNA executable on the file system level. This means that the capabilities are written to an extended file attribute stored in the file system (this step requires super user privileges), and will then be assigned to the process automatically upon execution. Effectively, this is similar to setting the `setuid` bit in the traditional file permissions, but the granted permissions are far more limited.

Experiment	Max. IAT	Min. IAT	Median IAT	σ(IAT)
Standard Kernel, no adaptions	1158	845	1000	26
Standard Kernel, RT adaptions	1072	904	1000	19
PREEMPT_RT Kernel, RT adaptions	1082	922	1000	25

Table 4.1: Metrics of the IAT distributions measured with different RT priorities, all IAT values in µs, 240000 packets per measurement, best values in red

The necessary capabilities for the LUNA executable file can be set with the following command, assuming the command is executed in the directory where the binary is located:

```
$ sudo setcap CAP_SYS_NICE,CAP_IPC_LOCK=pe luna
```

Other options for granting capabilities may be available or even preferable depending on the configuration and security requirements of the individual system. A detailed description of possibilities and security considerations when using capabilities is outside the scope of this thesis.

4.3 Effects

Several experiments were done to evaluate the effects of the aforementioned adaptions and the use of a PREEMPT_RT Linux kernel. All tests were done on the same system with the following properties:

CPU: Intel Core i5 650, 4 cores, up to 3.20GHz

RAM: 8 GB synchronous DDR RAM, 1333 MHz (two 4096 MB DIMMs)

Ethernet Card: Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet controller (rev 03), PCI ID `10ec:8168`

Once again, tests were done over the loopback interface to avoid influences from any underlying physical network.

4.3.1 Fundamental Changes

Figure 4.3 and Table 4.1 show the results of the adaptions described in Section 4.2. All three tests used 240,000 packets with a target inter send time (equal to the intended inter arrival time) of 1000 µs. Interestingly the smallest standard deviation was measured using a standard Linux kernel, although the difference is small and since the overall system load was low, according to Gleixner and Niehaus [12] a large difference would have been unexpected.

Using realtime scheduling, deviations from the requested timing were below 100 µs, whether the system was using a standard Linux kernel or one built with PREEMPT_RT. With maximum deviations of +72/-96 on standard Linux and +82/-78 on PREEMPT_RT Linux the difference between the kernel types is

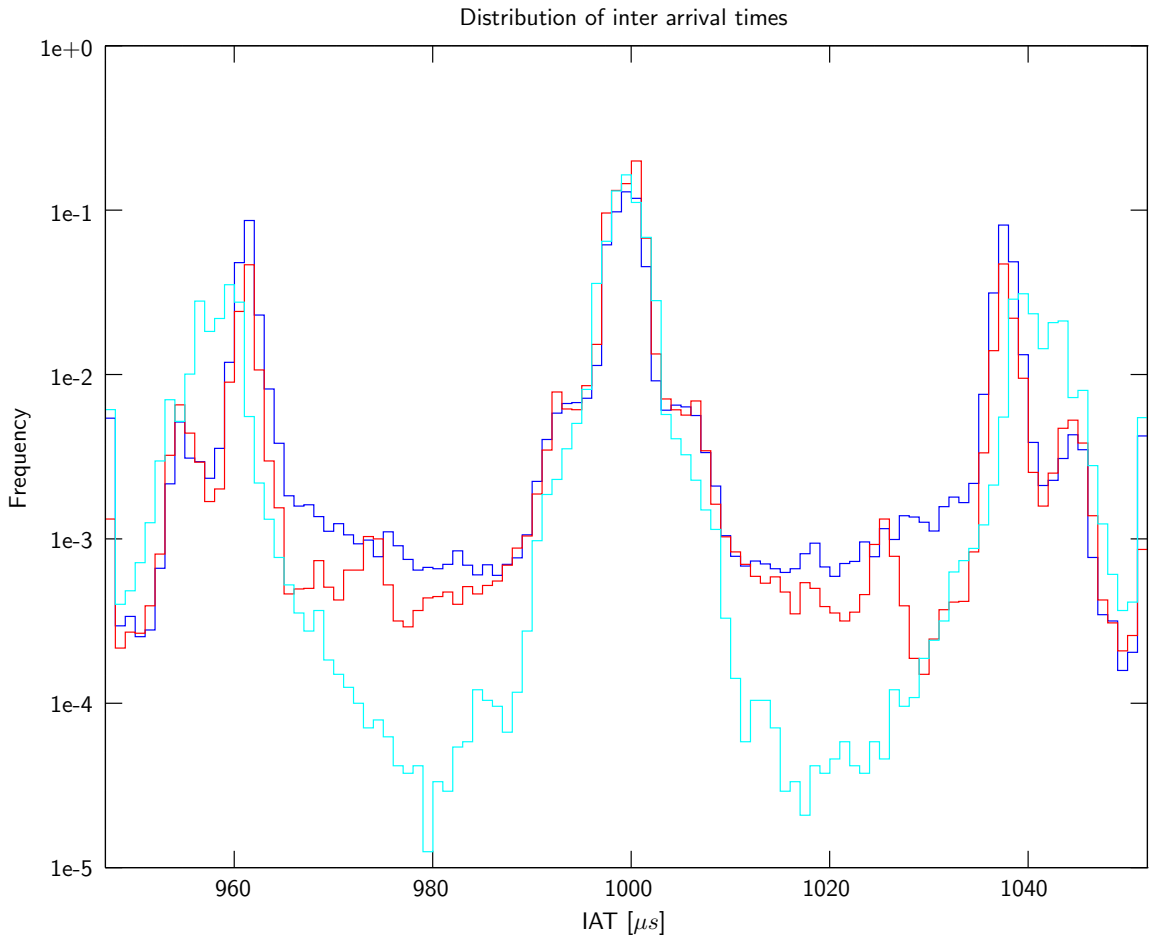


Figure 4.3: Inter arrival time distributions without RT adaptions on a standard Linux system (blue), with RT adaptions on a standard Linux system (red), and with RT adaptions on a PREEMPT_RT Linux system (cyan)

small, although the results are slightly better with PREEMPT_RT. However, compared to +158/-155 without the aforementioned adaptions the improvements are clear.

40 µs Peaks

In Figure 4.3, peaks can be seen about 40 µs above and below the target interval. This behavior occurs on both RT and non-RT systems. The reason remains unknown.

Ftrace, a probing system for the Linux kernel, shows that `ipv6_rcv` takes about 40 µs to return, but it is unclear whether there is a connection or not. For future work, experiments with Linux 3.11, which is supposed to allow lower receive latencies with certain network drivers by introducing *busy polling*, may be interesting [38].

Experiment	Max.	Min.	Average	σ
Standard Kernel, no adaptions	341	-3	43	13
Standard Kernel, RT adaptions	300	-7	40	10
PREEMPT_RT Kernel, RT adaptions	197	-6	48	18

Table 4.2: Metrics of kernel to user space transfer times, 240000 packets per measurement, all values in µs

4.3.2 Kernel to User Space Transfer Times

Table 4.2 shows metrics of kernel to user space transfer times under different conditions. With the lowest maximum by more than than 100 µs it is clear that the PREEMPT_RT kernel delivers the most reliable results here.

The minimum column shows contains negative values. What may seem like an evaluation error is actually an artifact created by measuring on an SMP (Symmetric multiprocessing) system. Each CPU core has a clock of its own, and these clocks may be slightly out of sync [39, "Note for SMP systems"]. If the kernel processes the arriving packet on one core and LUNA then reads the user space time on another, a short transfer time might look negative if the second clock is behind the first one.

Average transfer times and standard deviations were similar in all three experiments.

4.3.3 Differences between Multi Core and Single Core Systems

Time critical operations may behave differently on single and multi core systems. This was tested by selectively disabling three of the four cores on the multi core test system, so other influences could be avoided. The results can be seen in Figure 4.4 Table 4.3. The multi core data set is from the PREEMPT_RT test in Figure 4.3 and Table 4.1.

LUNA clearly shows better performance on the multi core system than on the single core one. The maximum deviation from the intended IAT is approximately 170 µs lower in the multi core test. Contrary to most figures in this chapter, Figure 4.4 shows the triple standard deviation range around the median (using the standard deviation from the single core experiment) to make the significantly wider spread easier to see.

The figure also shows that most IATs fall near the maximum on the multi core system, while in single core operation the peaks to the left and right are higher. Like the increased standard deviation, this indicates that timings are less precise on the single core system, likely due to the increased overhead from sharing one CPU with more processes than on a multi core system.

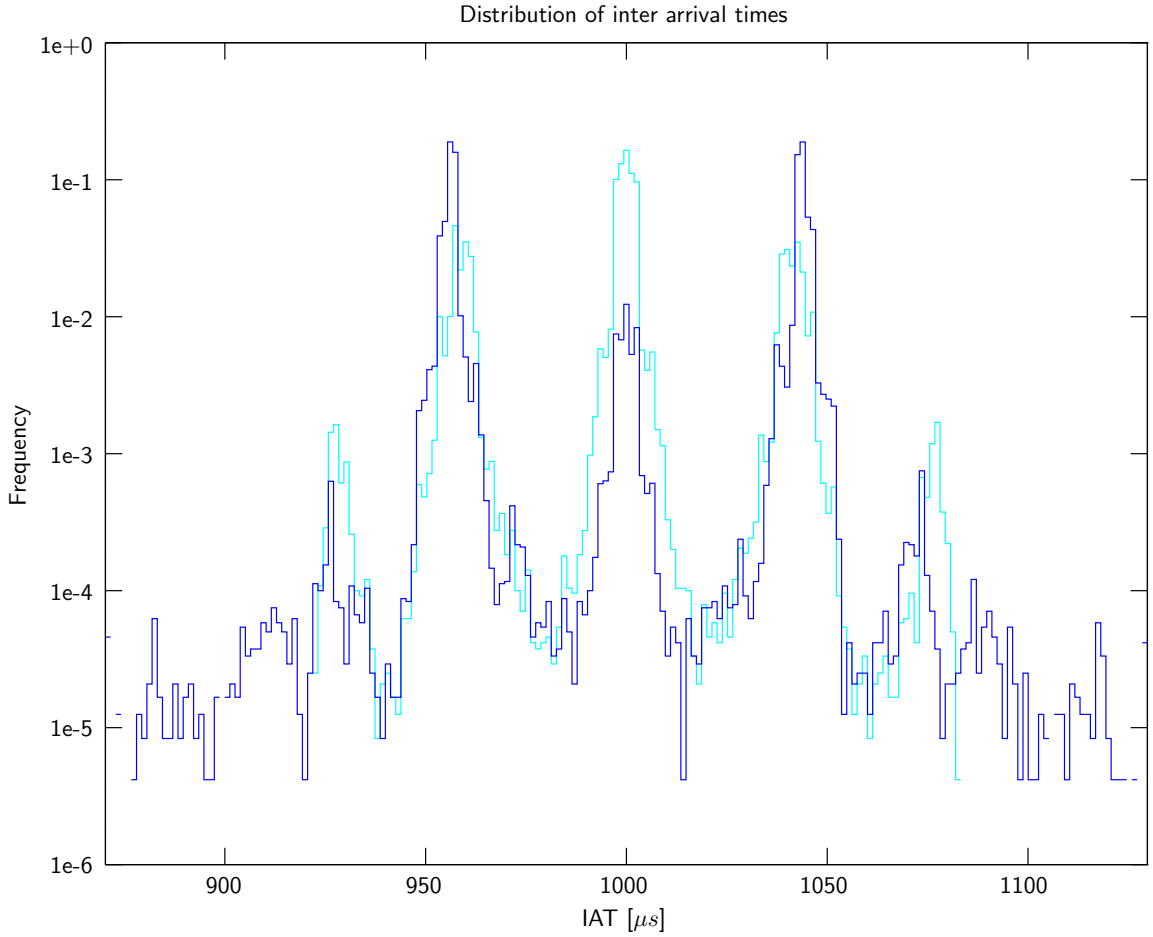


Figure 4.4: Inter arrival time distributions in multi core (cyan) and single core operation (blue)

4.3.4 Different Priorities for Client and Server

On a PREEMPT_RT system, time critical kernel threads also use realtime priorities, and can be preempted by processes with higher priorities — including user space processes, as described above. On the PREEMPT_RT test system, kernel threads use three realtime priority levels: 1, 50, and 99.

21, the priority previously used for LUNA, is obviously below 50, which raises the question whether increasing the priority might change the behavior, especially in regards to the 40 µs shifts (see Section 4.3.1). Also, client and server where running at the same priority, so one cannot preempt the other, which might force one process to wait or use another CPU.

Three more experiments were conducted to evaluate the effects of using other priorities, each using 240,000 packets over the loopback interface of the test system. The first one used a priority of 61 for client and server alike, thus being able to preempt several more kernel threads. The second test one used 61 for the client and 21 for the server, so only the client enjoys the elevated priority compared to the tests in previous sections, and can also preempt the server. Server preemption should not have a significant

Experiment	Max. IAT	Min. IAT	Median IAT	σ(IAT)
Multi core	1082	922	1000	25
Single core	1252	744	1000	43

Table 4.3: Metrics of the IAT distributions measured in multi core and single core operation, best values in red

RT Priorities	Max. IAT	Min. IAT	Median IAT	σ(IAT)
61 (Client and Server)	1074	757	1000	33
61 (Client), 21 (Server)	1074	872	1000	33
26 (Client), 21 (Server)	1079	812	1000	30

Table 4.4: Metrics of the IAT distributions measured with different RT priorities, all IAT values in µs, 240000 packets per measurement, best values in red

influence on precision, because arrival times are recorded in the kernel. In the third experiment, the client used a priority of 26 and the server 21, so the client can preempt the server, but not additional kernel threads compared to previous sections.

The results can be seen in Figure 4.5 and Table 4.4. Maximum IATs and standard deviations are very similar between the three experiments, but the minimum IATs show differences in favor of the 61 (client) and 21 (server) priority combination. However, while the maximum IAT is slightly lower (i.e.: better) than in Section 4.3.1, the minimums are lower (that is, not as good) than in the previous experiment. In conclusion, the effect of the changed priority is small, at least as long as there are no other realtime processes competing for CPU time.

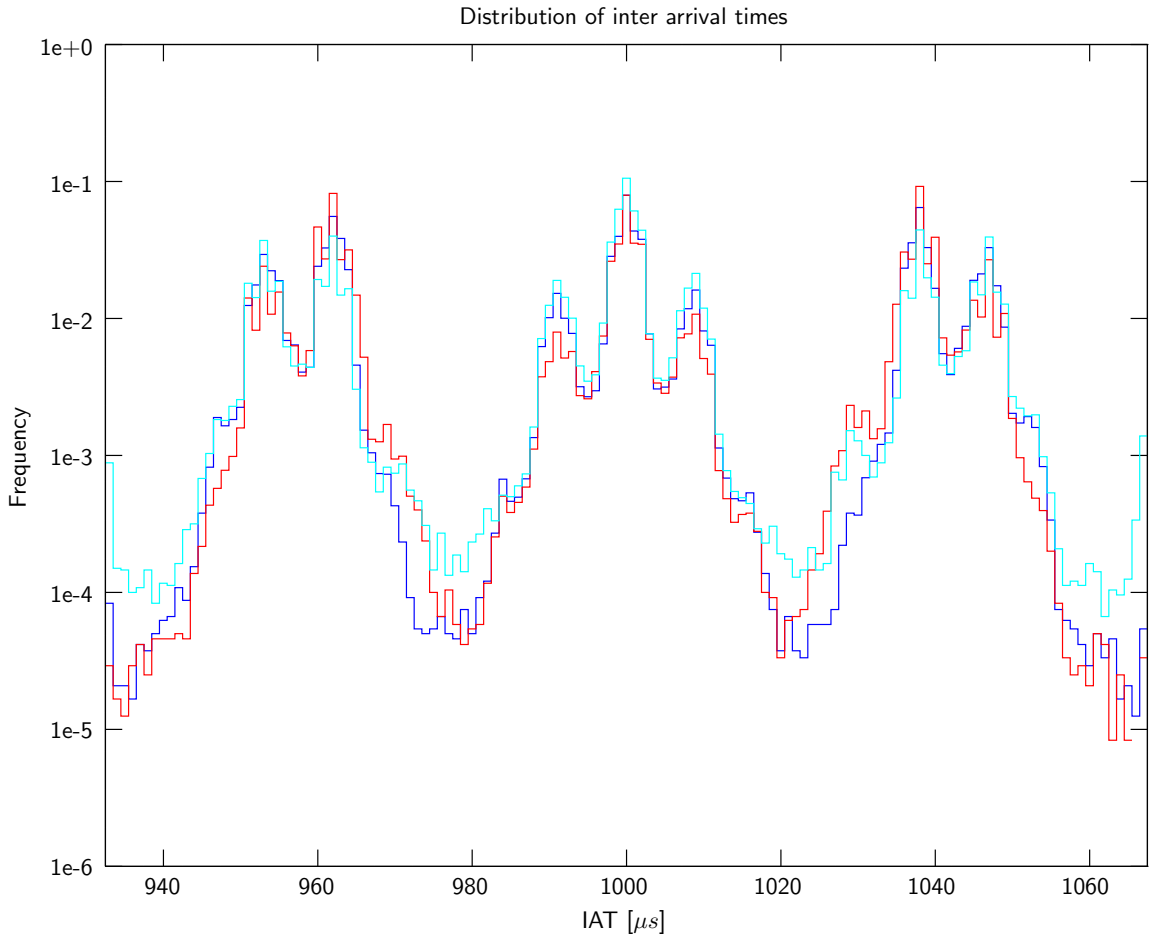


Figure 4.5: Inter arrival time distributions with different realtime priorities: 61 for client and server (blue), 61 for client and 21 for server (red), and 26 for client and 21 for server (cyan)

5 Flexible Traffic Characteristics

During the previous stages of development, LUNA was using fixed packet sizes and inter send times. However, one of the development goals is to enable flexible characteristics, for example following statistical traffic models. Different kinds of variation should be supported, and the mechanism should be designed to make adding new generators easy. In principle, two possible sources of data are possible:

Internal
: This means that packet sizes and timings are calculated in the LUNA process itself.

External
: LUNA receives packet sizes and timings from another process, or possibly uses manually defined values.

Similarly, two modes to create the data are imaginable:

Static generation
: Packet characteristics are defined before actually starting the transmission. Statically generated data could be used once or as a ring buffer.

Dynamic generation
: Packet characteristics are generated while the transmission is running. Of course, dynamic generation may require high generation speed depending on packet timings.

Table 5.1 shows possible combinations of sources and modes, and lists potential approaches for each combination.

5.1 Technical Considerations

Since LUNA is meant to be used for different scenarios, limiting it to a handful of generation methods would be beside the point. Instead, LUNA should provide a generic API that makes implementing new generation methods as easy as possible. Such implementations will be called *generators* from this point forth. Generators will likely require command line parameters of their own, so the generic interface must account for that.

Dynamic generators must meet several more requirements, especially in regards to smooth realtime operation:

	Static	Dynamic
Internal Generation	• Use fixed values • Ring buffer populated during LUNA initialization	• Use distribution functions from a math library
External Generation	• Read from file to ring buffer • Read from file, use once	• Read from socket • Read from STDIO

Table 5.1: Examples of considered generation methods

1. Any dynamic generation must work in parallel to sending. This means the generator must run in a thread of its own. The LUNA client will thus need at least two threads, a generator thread and a sending thread.

2. The generated packet parameters must be stored in a buffer. The buffer size required to prevent conflicts between sending and generation may vary between generators, depending on inter send times and generation speeds. Of course, the generator must be fast enough to keep up with the *average* send speed in any case.

3. The generator must not perform dynamic memory allocation during realtime operation. This means all memory required for dynamic generation must be allocated during generator initialization.

5.2 Implementation

> "I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships." (Torvalds [40])

Based on the technical considerations, this section describes how dynamic packet parameter generation is actually implemented in LUNA. Both data structures, namely the packet parameter buffer, and the process flow are described in detail.

5.2.1 Buffer Concept

As the primary interface between generator and sending thread, the packet parameter buffer must be defined first. For each packet to be sent, it must contain the size of the packet and the delay relative to the previous packet. The resulting data structure is shown in Figure 5.1.

Figure 5.1: Data structure for packet parameters

Managing each buffer entry individually would lead to a huge overhead for larger buffers and is therefore impractical. Thus, entries are grouped into buffer blocks. Each block will consist of an array of buffer entries and a management header. The array size can be variable and must be stored in the block header.

Because of the requirement to avoid dynamic memory allocation during realtime operation, all needed buffer blocks must be allocated before starting to send. This means the generator cannot allocate blocks, fill them, and hand them over to the sending thread to be used and released afterwards. Instead, a static approach is needed. Adding a pointer called "`next`" to another block to each block's header makes it possible to connect them to a linked list. Making the pointer in he last block point to the first one turns the linked list into a ring buffer. Figure 5.2 shows such a packet parameter block, including the array (gray) containing the actual data.

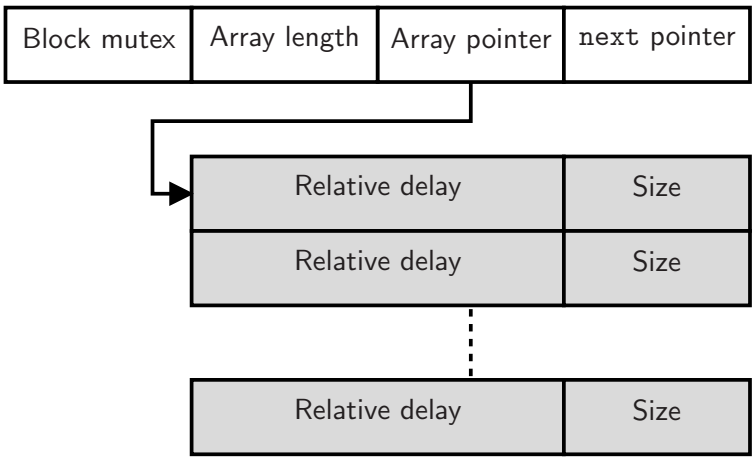


Figure 5.2: Packet parameter block with associated array

The ring structure allows the sending thread to endlessly follow the next pointers, until the operation time as set by the user is up. Similarly, the generator can update blocks that have been used, which requires a mechanism for the sending thread to let the generator know when it has used a block. Since both threads will read and write the buffer simultaneously, each block must also contain a mutex, which a thread must hold before writing or using the packet parameters stored in it.

The result is a ring buffer built from multiple blocks as shown in Figure 5.3. Each generator may select the block size and number of blocks as needed, the generic LUNA code provides functions to build blocks and assemble the ring buffer.

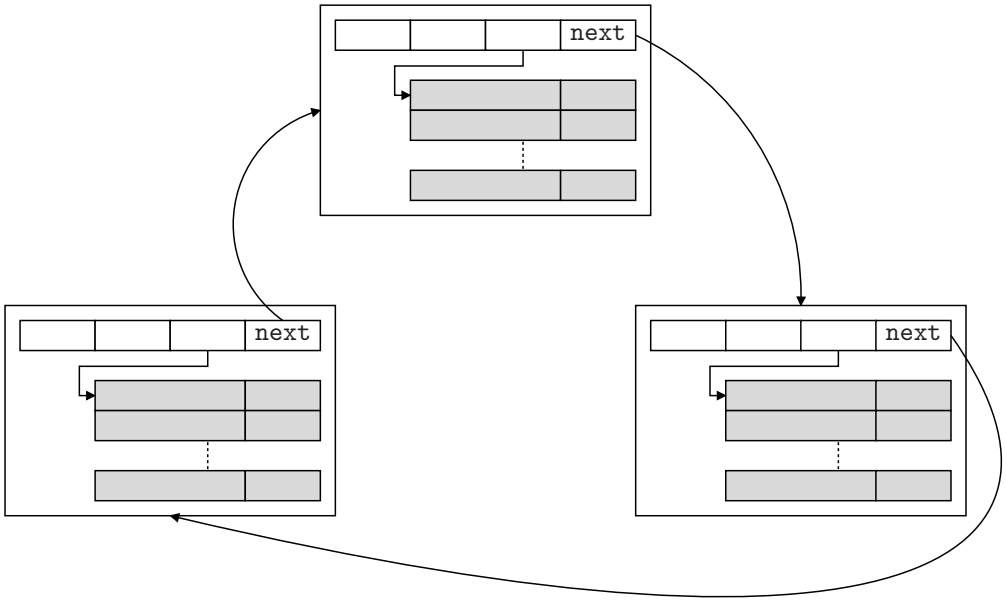


Figure 5.3: Ring buffer built from packet parameter blocks, details of which are shown in Figure 5.2

5.2.2 Generic Generator

In principle, any generator must fulfill three tasks:

1. Initialize the ring buffer
2. Fill the ring buffer with data
3. Refill blocks with fresh data after the sending thread used them (dynamic generators only)

The first two must be completed before the transmission can start, while the time when the third one must be done depends on the progress of the sending thread. This means some sort of signaling between the generator and sending threads will be necessary, the details are described in 5.2.3.

However, the tasks mentioned above can be used to define a generic processing flow for any generator thread, as shown in Figure 5.4. The steps that vary between different generators are marked in gray, while the other parts can be taken care of in a generic implementation. These generic parts are:

Reduce priority

By default, a new thread will inherit the priority of the thread that created it. However, running the generator at the same priority as the sending thread means that the latter could not preempt the former. To avoid blocking the sending thread this way, the generator will reduce its realtime priority by one if it is above the minimum realtime priority, which it will be as long as the priorities described in Chapter 4 are used.

Signal ready state

Signal to the sending thread that the buffer is ready and it can start sending.

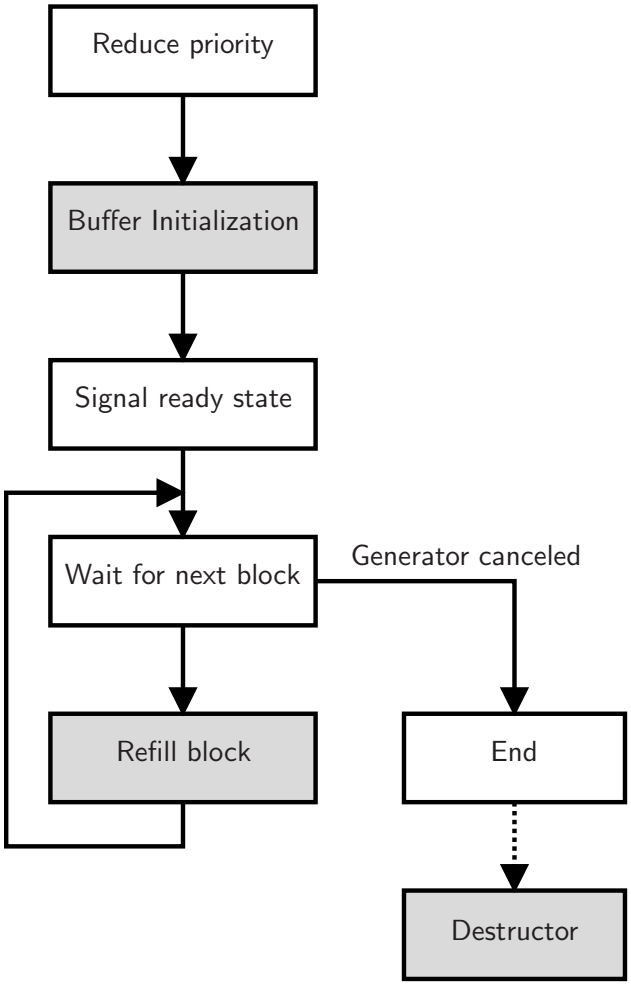


Figure 5.4: Flow diagram of a generic generator, gray parts vary between different generators. The destructor must be called from outside after the generator thread stops.

Wait for next block

Wait for a signal from the sending thread that it has completely used a block, so the generator should refill it (if the generator is dynamic).

End

When the generator is canceled, it will simply stop. However, the generator cannot be canceled while refilling a block. If a cancel signal is sent while doing that, the cancellation will be delayed until the refill is complete. This is necessary because the generator must hold the block mutex while refilling, and if the thread was canceled during that time, the mutex would never be released.

Function pointers in C make it possible to define a function signature and assign any function that matches to the pointer. The function can then be called by calling the pointer. In LUNA, this mechanism is used to implement a generic generator. Any generator is defined by combining three function pointers, one for each gray block in Figure 5.4, with other data needed for the generator into one data structure. The generic generator then calls the initialization and refill functions at the appropriate times. The

destructor function is not called by the generator thread itself, but associating it with the generator data is a simple way to ensure the right destructor is always known. A `void*` pointer for generator-specific data is also part of the generator data structure. Additionally, this concept means that anyone implementing a new generator does not need to worry about signaling and locking, because the generic generator takes care of that.

Generator-specific Functions

These are the functions shown in gray in Figure 5.4. All of them take a pointer to the generator data structure in question as the first parameter, other parameters are explained in the relevant description, if any. Additionally, each generator type needs a creation function (see "Generator Creation" below), which sets the function pointers and initializes the type-specific data field in the generator structure with any parameters and memory the other functions might need.

Buffer Initialization
: This function creates the ring buffer and fills it with initial data. A helper function to create a ring buffer with defined block size and number of blocks is available. The maximum packet size field in the generator data structure must be filled as well.

Refill block
: Called when the generator needs to refill a block, and requires a pointer to the block as the second parameter. This function pointer may be `NULL` for a non-dynamic generator.

Destructor
: Free all dynamically allocated memory connected to the generator, including the ring buffer. A helper function to free a complete ring buffer is available.

Generator Creation

In addition to the three functions described above, a generator implementation must also provide a creation function. It is responsible for:

- Parsing parameters parsed to the generator
- Populating the function pointers
- Allocating memory needed for generator-specific data
- Setting the maximum size attribute in the generator structure

To create a generator, the creation function for the desired generator implementation must be called on the generator structure, passing any arguments as well. Afterwards, the inter thread communication must be configured before starting the generator thread (see Section 5.2.3). The generic generator function takes care of the rest.

Parameters for the generator are read from the command line parameter `-a` as a list of comma-separated `name=value` pairs. Each generator can define its own parameters, but must provide default values for parameters that are not defined explicitly.

The Generator Structure

In addition to the function pointers described above, the generator structure, called `generator_t` in the C code, contains the following elements:

Buffer Pointer
: Points to one of the blocks ring buffer, as allocated in the generator initialization function. The sending thread should read this to find the buffer, but must not change it.

Ready Semaphore
: Used to signal the sending thread that the generator has completed its initialization (see Section 5.2.3).

Control Semaphore
: Used to receive signals from the sending thread when one block of data has been consumed and should be refilled if the generator is dynamic (see Section 5.2.3).

Maximum Size
: The maximum packet size the generator might request, relevant for buffer allocation. This field must be initialized by the generator creation function and must not change afterwards.

Generator Attributes
: Private to the generator thread. The structure defines only a `void*`, the creation function may place anything behind it. All this memory must be released by the destructor.

5.2.3 Threading

Threading in LUNA is implemented using Pthreads [41]. A Pthreads implementation is not only part of the GNU C Library (glibc)[1], the most commonly used C library and the de facto standard in Linux environments, but also the only one that supports priority inheritance according to the PREEMPT_RT documentation [29, section "Priority Inheritance Mutex support"], which is critical for proper realtime behavior.

[1] http://www.gnu.org/software/libc/

The sending thread is a somewhat extended version of the client as described in Section 3.2. Combining it with the generator thread from Section 5.2.2 requires some considerations on proper synchronization and inter thread signaling. The resulting process flow is shown in Figure 5.5 and described below.

The PREEMPT_RT documentation [29, section "Latencies caused by Page-faults"] warns against starting threads during realtime operation. This is not an issue here, because the generator must be started to create and fill the ring buffer before the transmission, which is the time critical part, can start. However, the sending thread needs a signal from the generator thread when the buffer is ready to start sending. Similarly, the generator thread needs a signal when it should refill a buffer block.

Semaphores

These signals are implemented using semaphores. The Linux Programmer's Manual describes the principle of semaphores as follows [42]:

> "A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)). If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero."

All semaphores in LUNA start with a value of zero, which is the default. The ready semaphore is incremented by the generator thread after it has completed initialization ("Ready signal" in Figure 5.5). In turn, the sending thread must decrement it before starting the transmission, and thus waits until the generator is ready, because `sem_wait()` blocks if the semaphore value is zero (Transition to "Start sending" in Figure 5.5) [43].

The control semaphore is used to control refilling of buffer blocks. After the generator thread has given the ready signal to the sending thread, it calls `sem_wait()` on the control semaphore and blocks until the sending thread increments it, which happens each time the sending thread has used a complete buffer block. Then, the generator thread refills the block if the generator is dynamic, and either way goes back to waiting.

It is not possible to use one semaphore for both signals, because if one thread is incrementing and decrementing the same semaphore, it is possible that it will consume the increment it placed before any other thread got a chance to run, effectively leading to a loss of signal, and likely causing other undesirable program behavior.

Memory Locking

As mentioned in Section 5.2.1, both threads will be accessing the ring buffer, which requires a locking mechanism to prevent conflicting accesses, implemented through one mutex per buffer block. Leroy [44] defines a mutex as follows:

> "A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors.
>
> A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first."

As mentioned before, each block in the ring buffer has a mutex associated with it. Before using any block, a thread must lock the block mutex, but the method varies between generator thread and sending thread. The generator uses the common `pthread_mutex_lock()` function, which blocks until the mutex is available if it cannot be locked right away, although this should not happen, because the generator only refills blocks after the sending thread has sent a signal via the control semaphore.

The sending thread, however, uses `pthread_mutex_trylock()`. This function does not block if the mutex is not available, but instead returns an error code. This might occur if the sending thread needs a new block while the generator is still busy refilling it. Returning an error message instead of blocking makes it possible to detect such a buffer underrun and alert the user rather than causing unexpected delays. A buffer underrun occurs if the generator is unable to refill the ring buffer at least as fast as the sending thread is consuming it. Possible reasons include insufficient hardware performance, conflicting processes consuming system resources, or even problems with the generator, and require user action to solve.

Stopping the Generator

After the sending thread has completed the transmission time requested by the user, the generator thread must be canceled so the program can terminate. Therefore, the sending thread calls `pthread_cancel()` [45] on the generator thread ("Cancel generator" in Figure 5.5). However, the generator thread must not be canceled while holding a block mutex, because that would lead to the mutex never getting released.

Pthreads can change their own cancelability state, and the generic generator code disables cancelability before acquiring a block mutex. After the block has been refilled and the mutex released, cancelability is reenabled. If a cancellation signal is received while cancellation is disabled, it will be delayed until cancellation is possible again.

The sending thread must call the destructor of the generator, but this must not happen before the thread has actually stopped. Therefore it joins the generator thread using `pthread_join()` [46], a function that returns only after the joined thread has stopped (Transition to "Destructor" in Figure 5.5).

5.2.4 Summary: How to implement a new Generator

Implementing a new generator requires implementing all four generator-specific functions: a creation function, buffer initialization, block refilling, and a destructor (the gray blocks in Figure 5.5). All functions must match the signatures defined in `generator.h` in the LUNA source code.

The generator will likely need a private data structure for the data stored behind the generator attributes pointer as well, but there are no requirements beyond proper memory management on this. Note that "proper memory management" not only includes ensuring that all allocated memory is freed in the destructor, but also avoiding dynamic memory allocation during realtime operation. Thus, the block refill function must not allocate memory dynamically.

To let the client process use the generator, a header that declares the creation function is required, as well as an entry in the list of known generators in `client.c`. A plugin system was not implemented as part of this thesis work, but would be a useful extension for future work.

The point here is that any kind of packet timing and size distribution can be tied into LUNA using the generator API, no matter if the generation is static or dynamic, should happen internally or externally, and what its theoretical basis is.

5.3 Example: Gaussian Packet Size Generator

The Gaussian packet size generator has been implemented as an example dynamic, random number based generator. It creates packets with sizes according to a Gaussian distribution based on two parameters: `max` (the maximum permitted packet size), and `sigma`, the desired standard deviation of the Gaussian distribution. Additionally, it takes the parameter `interval`, which defines the static IDT of the packets.

The necessary random numbers are created with a random number generator from the GNU Scientific Library[2] (libgsl), which provides multiple functions for this purpose. The function selected for packet size generation is `gsl_ran_gaussian_ziggurat()`, because it provides the fastest available algorithm according to libgsl documentation [47, Section 19.2][3]. The random number generator provides `double` values, which have to be rounded to whole bytes before they can be used as packet sizes.

Figure 5.6 shows the size distribution of recorded packets with a sample size of 120000 packets. The histogram does indeed show a Gaussian distributions, with small variations that are to be expected when calculating random numbers. The peaks at the leftmost and rightmost sides of the histogram occur because the generator has been implemented to replace values below the minimum packet size or above the configured maximum (300 bytes in this case) with the closest permitted value. In conclusion, both the generator and its connection to the LUNA packet sending mechanism work as designed.

[2] http://www.gnu.org/software/gsl/

[3] Section 20.2 in the libgsl online manual at the time of this writing, see http://www.gnu.org/software/gsl/manual/html_node/The-Gaussian-Distribution.html

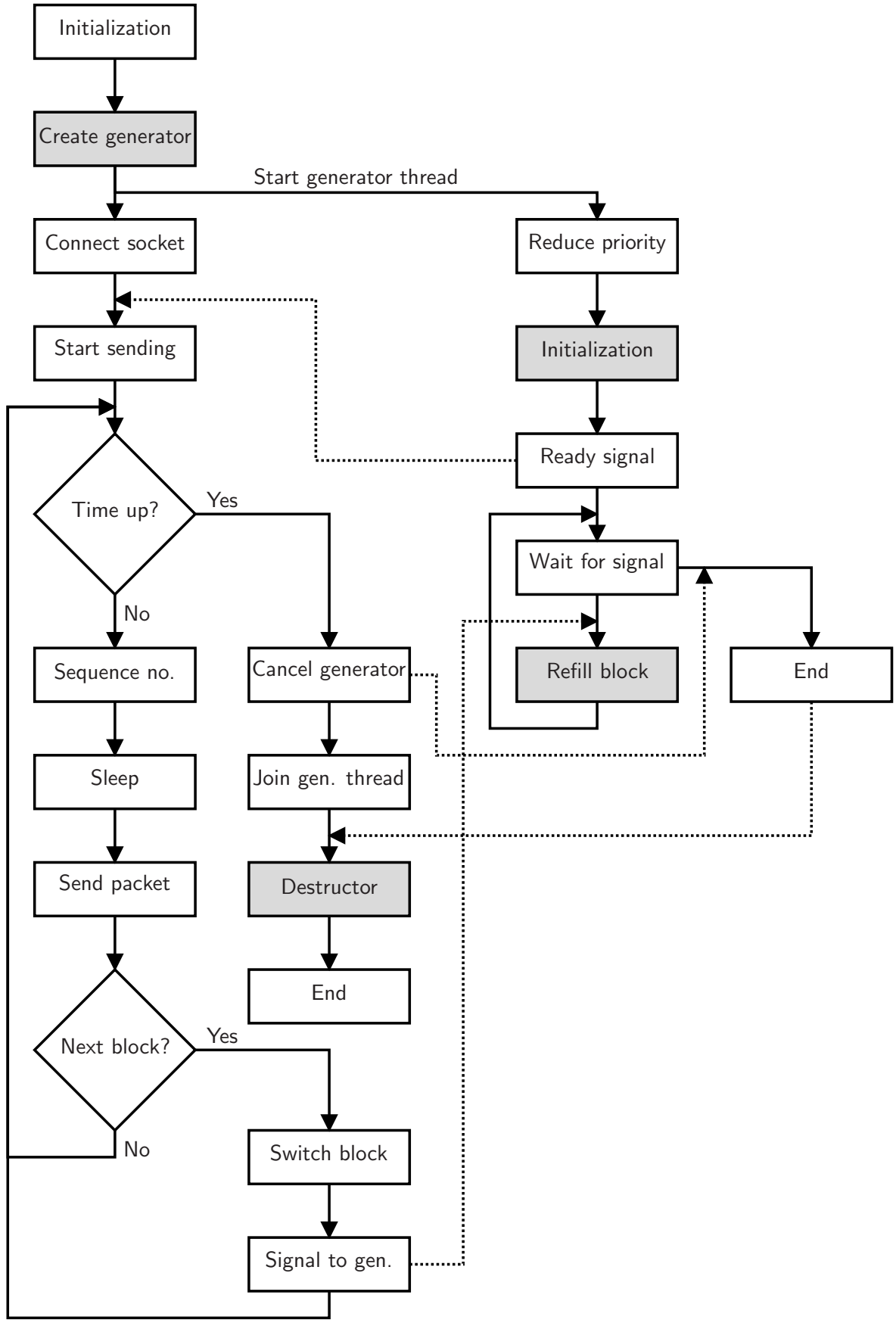


Figure 5.5: Flow diagram of generator and sending threads, gray parts are generator-specific. Dashed lines mark inter-thread signals that are necessary for a certain transition to occur.

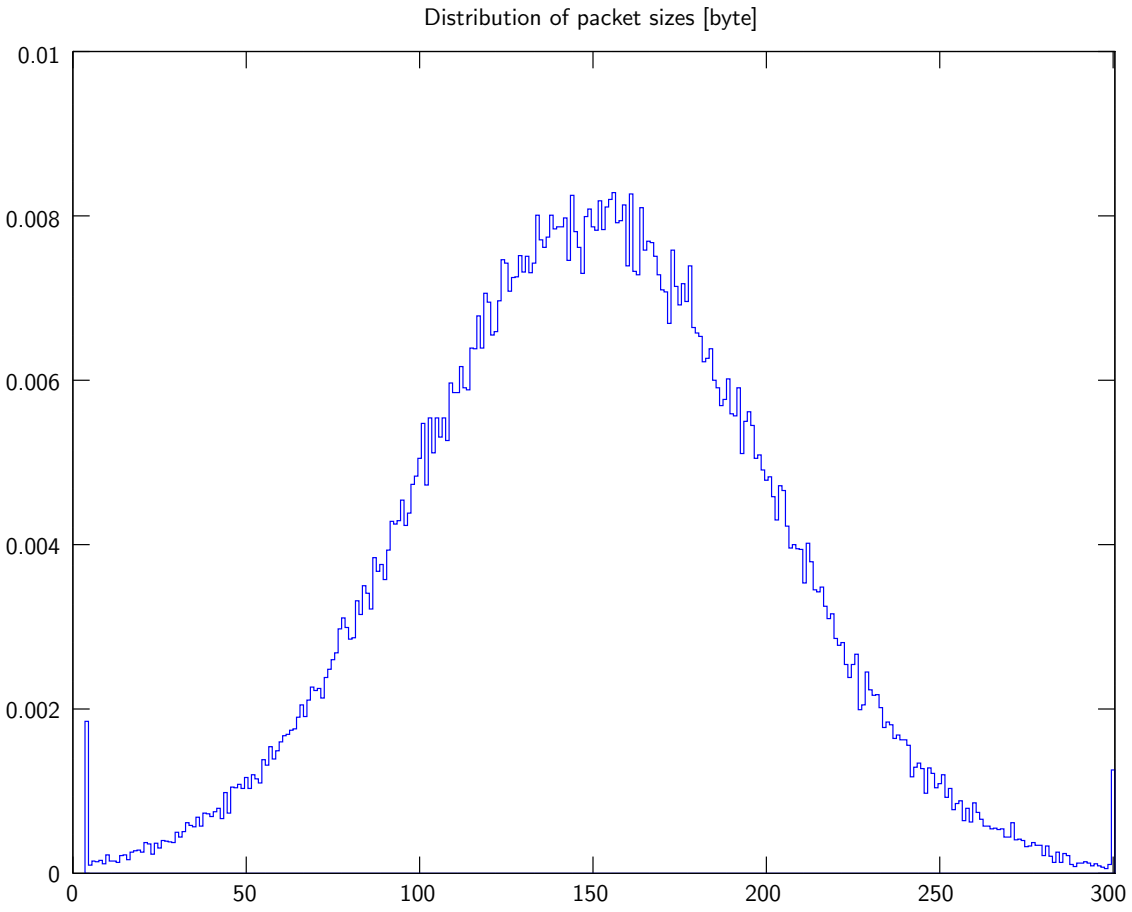


Figure 5.6: Distribution of recorded sizes of packets generated with Gaussian distribution, sample of 120000 packets

6 Round Trip Time Measurements

Measuring round trip times is a frequent requirement in network analysis, but up to this point, LUNA only supported one way transmissions. This chapter describes the enhancements to the LUNA protocol and code which were required to enable RTT measurements.

To facilitate calculating round trip times, each packet's send time will be recorded in the packet, avoiding the need for a packet database including send times on the client side. Additional, setting a flag inside the packet will allow the client to request an echo from the server, so both one way and two way transmission will be possible. This requires extending the protocol described in Section 3.1 as shown in Figure 6.1. The timestamp is stored using the 16 byte `struct timespec` format [39]. One byte is enough for the flags field, bringing the minimum packet size up to 21 byte UDP payload, plus UDP and IP headers.

Figure 6.1: LUNA protocol as extended for round trip time measurements

The flags field is set while initializing the send buffer of the client, so it does not require any additional processing during the transmission.

6.1 Recording Send Times

Just recording the send time is as easy as calling `clock_gettime()` [39] right before sending the packet in such a way that the result will be written to the appropriate location in the send buffer. However, there is one catch.

As explained in Section 3.2, the send thread sleeps until it is time to send the packet. Reading the clock before sending might introduce a delay, the uniformity of which is not known. Therefore, a direct comparison of inter arrival times between sending with and without timestamps has been performed via loopback interface on a PREEMPT_RT system, using the same packet size each time. The results can be seen in Table 6.1 and Figure 6.2. This experiment was done before changes to support echoing packets were made in the server, or processing for echo packets was added to the client.

Experiment	Max. IAT	Min. IAT	Median IAT	σ(IAT)
Without time records	1089	880	1000	30
With time records	1097	891	1000	34

Table 6.1: Metrics of the IAT distributions without and with send time record in each packet, all IAT values in µs, 240000 packets per measurement, using PREEMPT_RT kernel, best values in red

The peaks around 960 and 1040 µs become somewhat wider, while the critical metrics hardly change. The minimum IAT is even better than without time recording, although the difference is so small it might easily be within measuring inaccuracy. In conclusion, the inaccuracy introduced by recording the time is small enough to be acceptable.

6.2 Echoing Packets

Once the server receives a packet, it needs to check if the client has requested an echo. However, before that can be done, the server must verify the size of the packet. If it is below the minimum size defined by the protocol, trying to read the flags field would simply return whatever the receive buffer happens to contain at that position, leading to unpredictable results. Of course, packets below the minimum size should never be received from a proper LUNA client, but a networked process should be able to deal with transmission errors, or even packets from a completely different and unexpected source.

If the packet passes the size check, the server looks at the echo flag inside the flags field. If it is set, the packet is echoed back to the client. The echo packet is sent right from the receive buffer without any modification, sending as many bytes as were received, using the source address of the original packet as the destination.

To minimize the delay added to the round trip time, the packet is processed as described in the previous chapters only after the echo has been sent, if it was indeed requested. The resulting process flow is shown in Figure 6.3.

6.3 Echo Processing

Receiving and processing the echos on the client side is more complicated, because it cannot be handled by the same thread as sending. Receiving the echos independent of other tasks requires a third client thread, the generator thread being the second. This thread will be called the “echo thread”. All steps related to starting and managing the echo thread are only executed if the user has requested echos using the -e command line option.

The socket must be available to the echo thread, so the thread is started directly after the socket has been configured. The data structure passed to the new thread contains both the socket and a semaphore

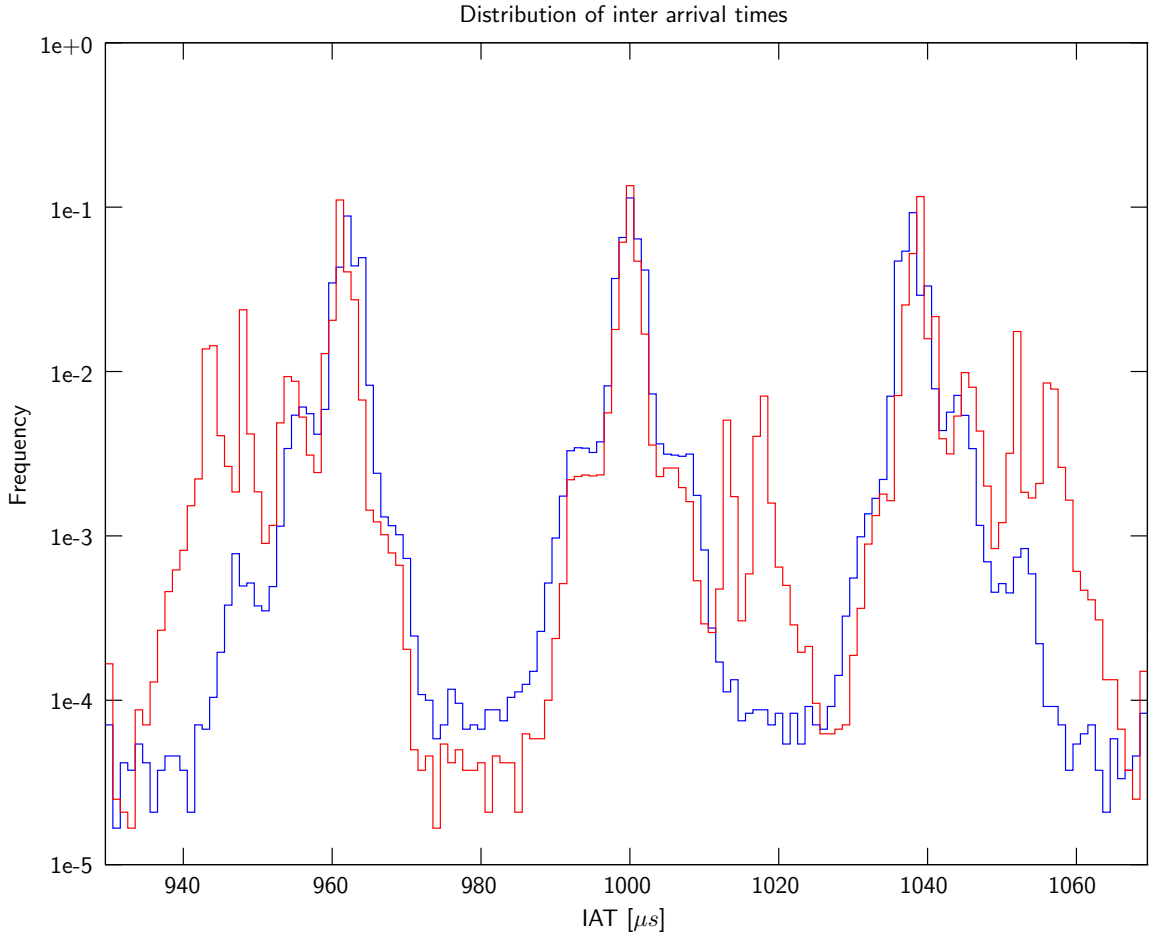


Figure 6.2: Inter arrival time distributions without (blue) and with (red) send time record in each packet

which the thread uses to signal to the main client thread that its initialization is complete, like the ready semaphore for the generator thread described in Section 5.2.3.

During its initialization, the echo thread allocates the buffers and other variables needed for receiving packets. It also reduces its priority relative to the sending and generator threads, because like the server, it can get packet timestamps from the kernel and doesn't need extremely precise internal timing, while delaying a sending or generator thread could hurt the precision of measurements taken. After initialization is complete, the echo thread posts to the semaphore provided, which the sending thread waits for before starting the transmission. The echo thread then starts waiting for packets to arrive.

When a packet arrives, the echo thread also requests the arrival timestamp from the kernel. After checking the packet size, it reads the timestamp included in the packet, and calculates the round trip time, which is then printed to standard output along with the packet's arrival time, sequence number, and size. As with the server output, evaluation of the results can be done by any means the user desires, for this thesis work evaluation was implemented in GNU Octave.

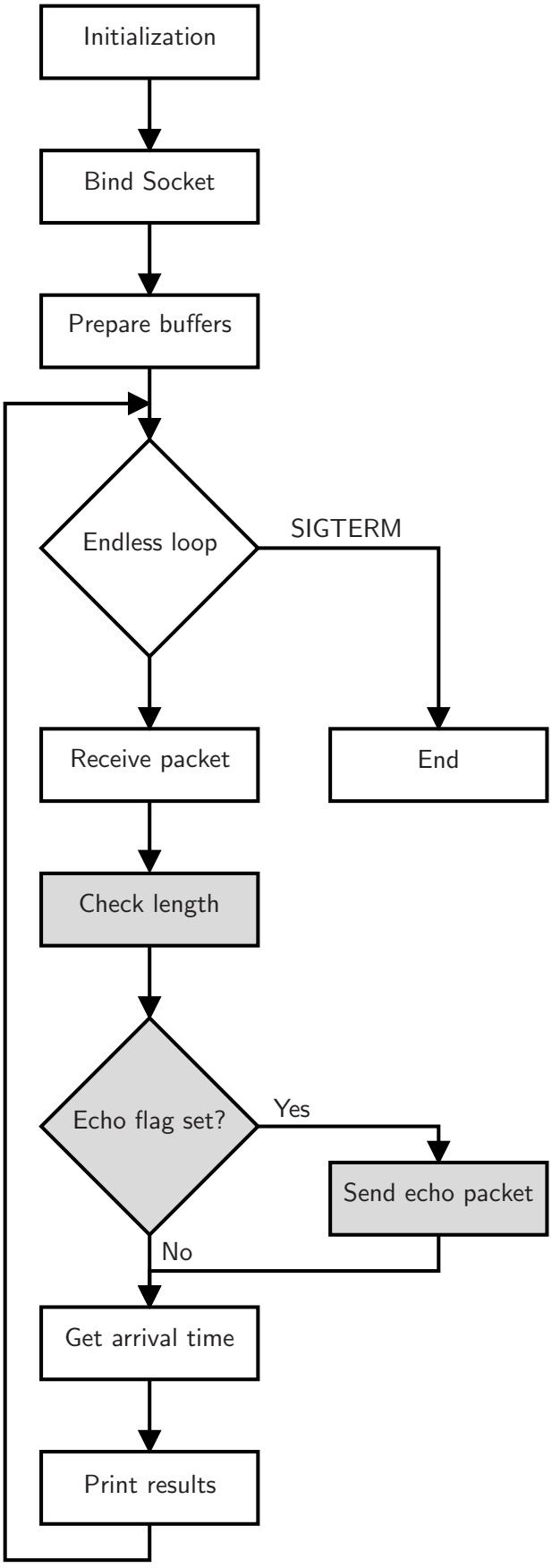


Figure 6.3: Server process flow with echo support, new elements compared to Figure 3.5 in light gray

The echo thread uses cancellation callbacks to ensure its dynamically allocated memory is freed after it stops, by pushing a call to `free()` on the cancellation cleanup stack right after allocation [48]. When the client cancels the thread after completing the transmission, the cleanup handlers get called and release the memory. After joining the echo thread, the main client thread can just release the data structure passed on thread initialization.

The process flow of the LUNA client with echo and generator threads is shown in Figure 6.4.

6.4 Round Trip Time Measurement Test

To test the performance of the round trip time measurements, a simple comparison with the well-known tool `ping` was done. Although LUNA and `ping` use different underlying protocols (UDP and ICMP, respectively), resulting round trip time should be similar. To ensure best possible comparability, packet sizes were adjusted to match. Both the UDP header [17, p.1] and the ICMP echo header [49, p.14] have a size of 8 bytes. The minimum UDP payload for LUNA is 21 bytes as described at the start of this chapter, and `ping` was configured to send a data section of the same size.

The test was done over Ethernet between the PREEMPT_RT system used in previous test, and a laptop computer running a standard Linux kernel, the latter acting as the server, because realtime performance is less important on the server side. The LUNA run was done at the default rate of 1000 packets per second over 20 seconds, the `ping` test with its default rate of 1 packet per second over 100 seconds, which should provide sufficient precision for the purpose of verifying LUNA measurements.

The overall results can be seen in Table 6.2, the round trip time distribution recorded by LUNA in Figure 6.5. The numbers show clearly that both minimum and average LUNA round trip times are lower than those measured by `ping`. The very long maximum round trip times are likely caused by delays on the server, where the non-PREEMPT_RT kernel does not allow the server process to preempt the kernel to ensure quick echos. The detailed distribution in Figure 6.5 also looks as expected: a few very low values on the left, leading up to a wide peak of common round trip times around approximately 300 µs, and trailing off to some larger values on the right. Note that the sharp peak on the rightmost side includes all values above the x-axis range of the histogram.

In conclusion, the round trip times measured using LUNA are similar to those measured using `ping`, if not closer to the technical minimum, and are distributed as round trip times are expected to be, indicating that the previously described measurement method works well.

Program used	Max. RTT	Min. RTT	Average RTT
LUNA	3133	200	323
`ping`	4666	283	460

Table 6.2: Comparison of round trip time measurements with LUNA and `ping`, all values in µs

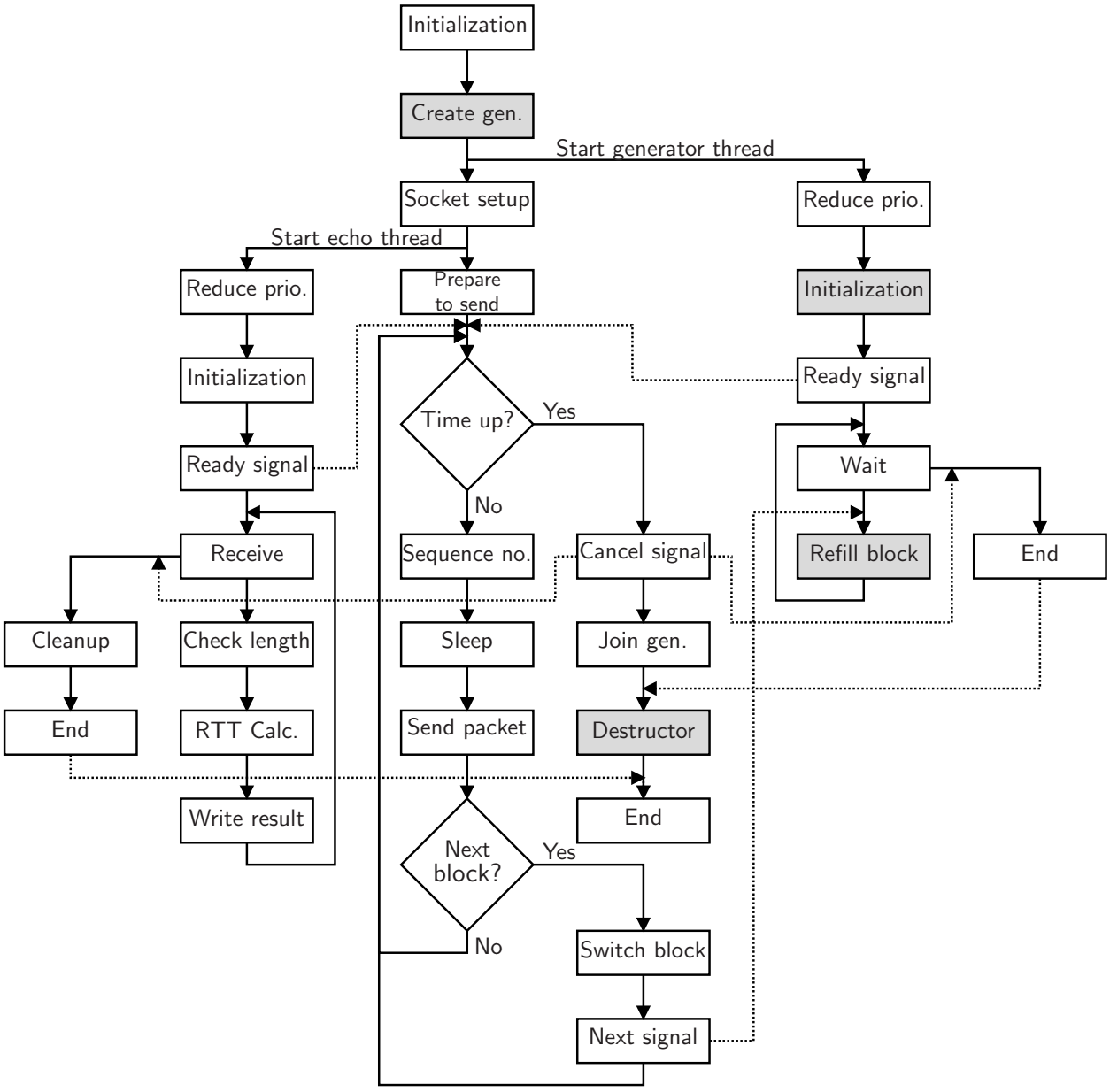


Figure 6.4: Client flow diagram with sending, generator and echo threads, based on Figure 5.5. The echo thread is only started if the user configured LUNA to request echo packets.

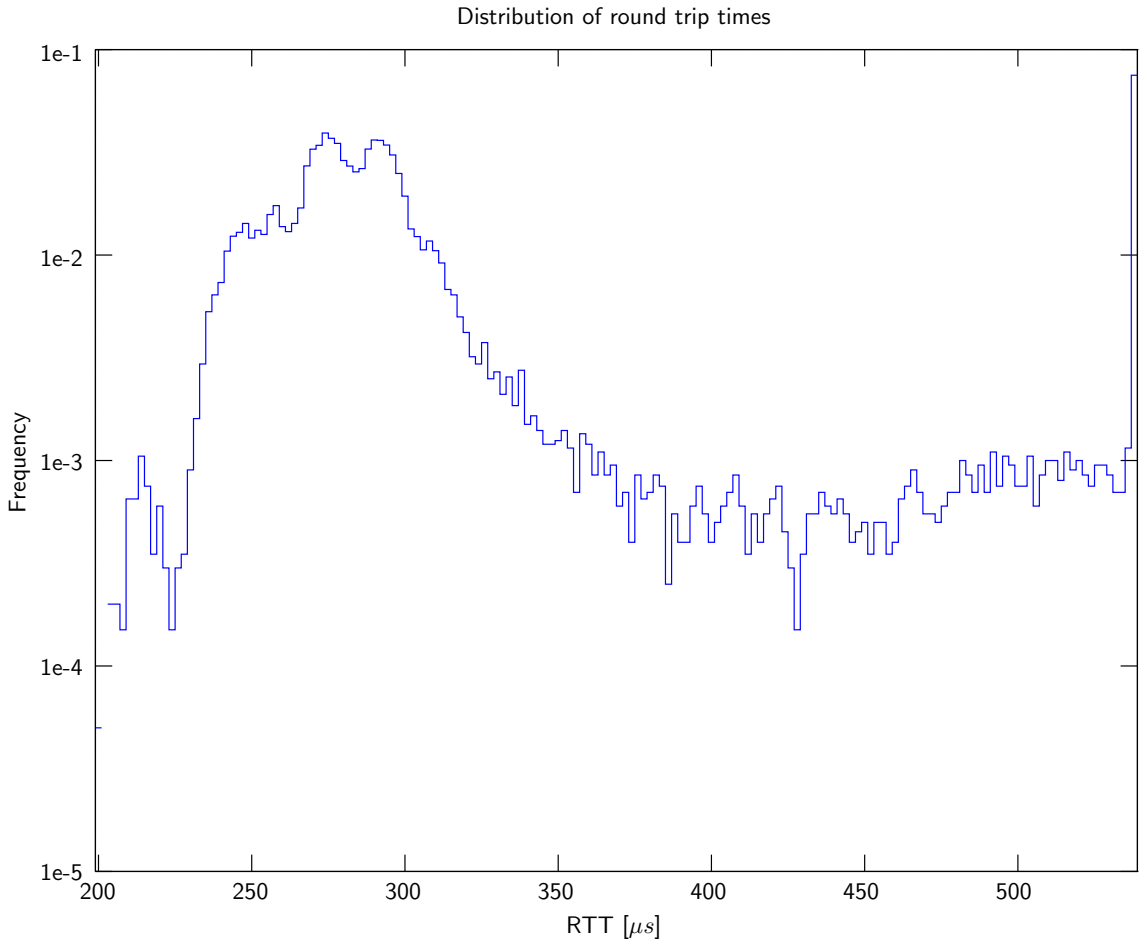


Figure 6.5: Round trip time distribution recorded by LUNA

7 Distributed Traffic Generation

When analyzing networks, the ability to control the whole experiment from one *control host* is usually desirable. LUNA needs to account for this requirement. However, remote control features should be strictly separate from the core packet generation system to avoid overhead and preserve flexibility.

Sections 8.2 and 8.3 show example applications of the remote control system described in this chapter.

7.1 Technical Considerations

A fundamental requirement is that configuration files should be stored exclusively on the control host. Requiring the user to edit transmission configurations on each or some of the controlled hosts would defeat the purpose of remote control.

At the same time, control connections must be secure. Controlled by the wrong person, a packet generator could easily be abused for packet flooding based denial of service attacks, and might provide other avenues of attack as well. While it would be possible to implement a special control protocol, it is preferable in regards to both security and maintainability to use existing, well tested protocols. Since connection setup requires a few more steps beyond just starting a LUNA process, SSH[1] [50] is an obvious choice, because it allows running other commands, too, as well as transferring files.

Another concern is that network traffic caused by transmitting control messages might influence the measurement results. Using a separate control network is the most reliable way to avoid such errors, but LUNA should not require a certain approach, and rather support both measurement setups with and without a separate control network.

7.1.1 Necessary Configuration Parameters

Table 7.1 shows the parameters needed for configuring one transmission on both sides and on the control host. The client IP address or hostname does not need to be known on the server side, because the server can record addresses from the packets it receives. The server port configured on the client side obviously must be identical to the listening port on the server, so in practice this parameter needs to be set only once. Temporary output files on both sides are necessary to record the results while the experiment is running, afterwards they will be transferred to the control host.

[1] SSH: **S**ecure **Sh**ell

Client Side	Server Side	Control Host
Server (Hostname or IP address)		Server (possibly different)
Server port	Listening port	Client
Measurement time		Output files for logs
Generator		LUNA binary path
Generator parameters		
Request Echo?		
Temp. output file (if echo requested)	Temp. output file	

Table 7.1: Parameters needed for one transmission, separated by client side, server side, and control host

It is possible to define reasonable defaults for most of these values, but client, server, and their output files (client only when echo is enabled) must be set by the user.

An important design decision here is to start one server process per transmission. This has multiple advantages compared to handling all incoming transmissions on one host in the same process:

1. Multiple processes should scale better for multiple transmissions to one host, because arriving packets can be handled in parallel as far as the hardware permits. Since UDP is a connectionless protocol, it is not possible to do the same by starting a new handler thread for each incoming transmission (in that case, incoming packets would have to be received and demultiplexed by one thread, and could then be distributed to handler threads).

2. If one server process handles one transmission, it is not necessary to use a transmission identifier, which keeps the minimum packet size small.

3. This also avoids the need to separate transmissions in the log file at evaluation time, because there is exactly one log file per transmission.

A slight disadvantage of using multiple server processes is that they will require multiple listening ports.

7.2 Implementation

The remote control has been implemented in Perl. The built-in text parsing functionality of Perl enables parsing configuration files, and the Perl package `Net::OpenSSH`[2] provides a programmable SSH client, based on OpenSSH[3]. When connecting to a remote host, `Net::OpenSSH` establishes a permanent master connection. Subsequent requests can effectively reuse that connection, reducing delays. Figure 7.1 shows the control flow of the remote control system, which is explained in more detail in the following sections.

[2] http://search.cpan.org/~salva/Net-OpenSSH/lib/Net/OpenSSH.pm
[3] http://www.openssh.org/

Using standard output redirection to write to output files as before would have made running processes in the background (see Section 7.2.3 below) impossible, so the command line option `-o FILE` to write logs to a file instead has been added to LUNA.

7.2.1 Configuration File

The configuration file should be in a format that is both reasonably accessible to humans and does not require complex code to parse. Thus, configuration options are stored in simple `name=value` pairs. Global configuration options are defined at the beginning of the file, before any transmissions are configured. Each transmission is configured in a *section*, which starts with the transmission identifier enclosed in square brackets. A section ends either at the beginning of a new section or the end of the configuration file. Lines starting with "#" and empty lines are ignored. Whitespace characters are only permitted as part of comments, variable values (not names), and to indent variable assignments. Section definitions may not contain any whitespace, section and variable names can only contain alphanumeric characters and underscores.

An example configuration file defining one transmission with packet echo enabled can be seen in Listing 7.1. If the client should use a different hostname or IP address to reach the server, it must be specified in the `target` variable inside the transmission section.

Listing 7.1: Example configuration for the LUNA remote control system

```
1 # a global variable definition
2 default_exec=/usr/bin/luna
3
4 # a transmission section
5 [conn]
6 server=192.0.2.1
7 client=192.0.2.2
8 server_output=test-server.log
9 client_output=test-client.log
10 echo=true
```

The remote control script expects the configuration file as the first and only command line parameter. As it is parsed, each transmission section is stored in a dedicated variable of Perl's hash type.

7.2.2 Starting Handler Threads

After the configuration has been parsed, a handler thread is started for each defined transmission, passing a reference to the hash structure containing the configuration variables. The handler threads operate completely independent of each other.

7.2.3 Inside the Handler Thread

SSH Connection Setup

The SSH connection is initiated using the `new` routine provided by `Net::OpenSSH`. The system is built with the assumption that public key authentication from the control host to the measurement hosts has been configured by the user before starting the measurement. Other authentication methods can be used when available, but interactive authentication is not feasible for automated connections.

Usually client and server will each require an SSH connection, but if they have been configured to run on the same host, the system will recognize this and use the same master SSH connection to control both.

Starting the Server

Before the server is started, two temporary files must be created: One to store the process identifier (PID), needed to cleanly stop the server after the transmission is complete, and another to store the output, because directly capturing the output over the SSH connection while the transmission is running would create a high risk of influencing the measurement, especially without a dedicated control network. Both files are created using the standard `mktemp` command[4], and the file names are transmitted back to the control system for use in later commands.

The server is then started through `start-stop-daemon`[5], using its capabilities to send the server into the background and store its PID to facilitate a clean shutdown later.

Running the Transmission

The client command is built using the generator, echo and time settings read from the configuration file. If packet echoes have been requested, an output file for the echo logs is needed, which is created using `mktemp` like the server output file. A PID file is not necessary, because the client is kept in the foreground so the handler thread will get a signal through the SSH connection when it terminates.

After assembling the client command, it is run. When it terminates, the "run client" stage in Figure 7.1 is done, as long as no echo has been requested.

If echoes have been requested, the log file must be transferred to the control host. This is done using the file transfer capabilities of SSH, namely the `scp_get` routine provided by `Net::OpenSSH`. Once the transfer is complete, the temporary file on the client host is deleted.

[4] http://www.gnu.org/software/coreutils/manual/html_node/mktemp-invocation.html
[5] http://manpages.ubuntu.com/manpages/raring/en/man8/start-stop-daemon.8.html

Stopping the Server

Stopping the server after the client process has terminated ensures that the server is not stopped while the transmission is still running. The PID file is read and the PID is used to send a `SIGTERM` signal [51] to the server, allowing it to terminate cleanly as described in Section 3.3.1.

Like a possible client log, the server log is copied to the control host using the SSH connection. Afterwards, both temporary files (log and PID file) on the server host are deleted.

7.2.4 Joining Handler Threads

As with pthreads, "joining" a thread means waiting for its termination. After all handler threads have started, the main thread of the remote control system tries to join them. Once all threads have been joined, meaning all transmissions have been completed, the control program terminates.

7.2.5 Possible improvements

For future work, LUNA would benefit from adding the ability to start transmissions at a fixed time, instead of starting as soon after program start as possible. This would help with experiments in real networks, where tests might be required to start at certain times, as well as optimizing the precision of experiments with multiple interfering transmissions. Approximate timings could be implemented in the remote control system, but reaching sub-second start time precision would require adjustments in the LUNA packet generator as well.

Either way, clock synchronization across hosts would need to be done independent of LUNA, for example using NTP or GPS signals.

A mechanism to assign server ports automatically might be an improvement in terms of user convenience. Currently, the user must set alternative server ports when one measurement host needs to handle more than one incoming transmission. Ideally, the user would only need to configure ports if the measurement setup expicitly requires certain ports to be used.

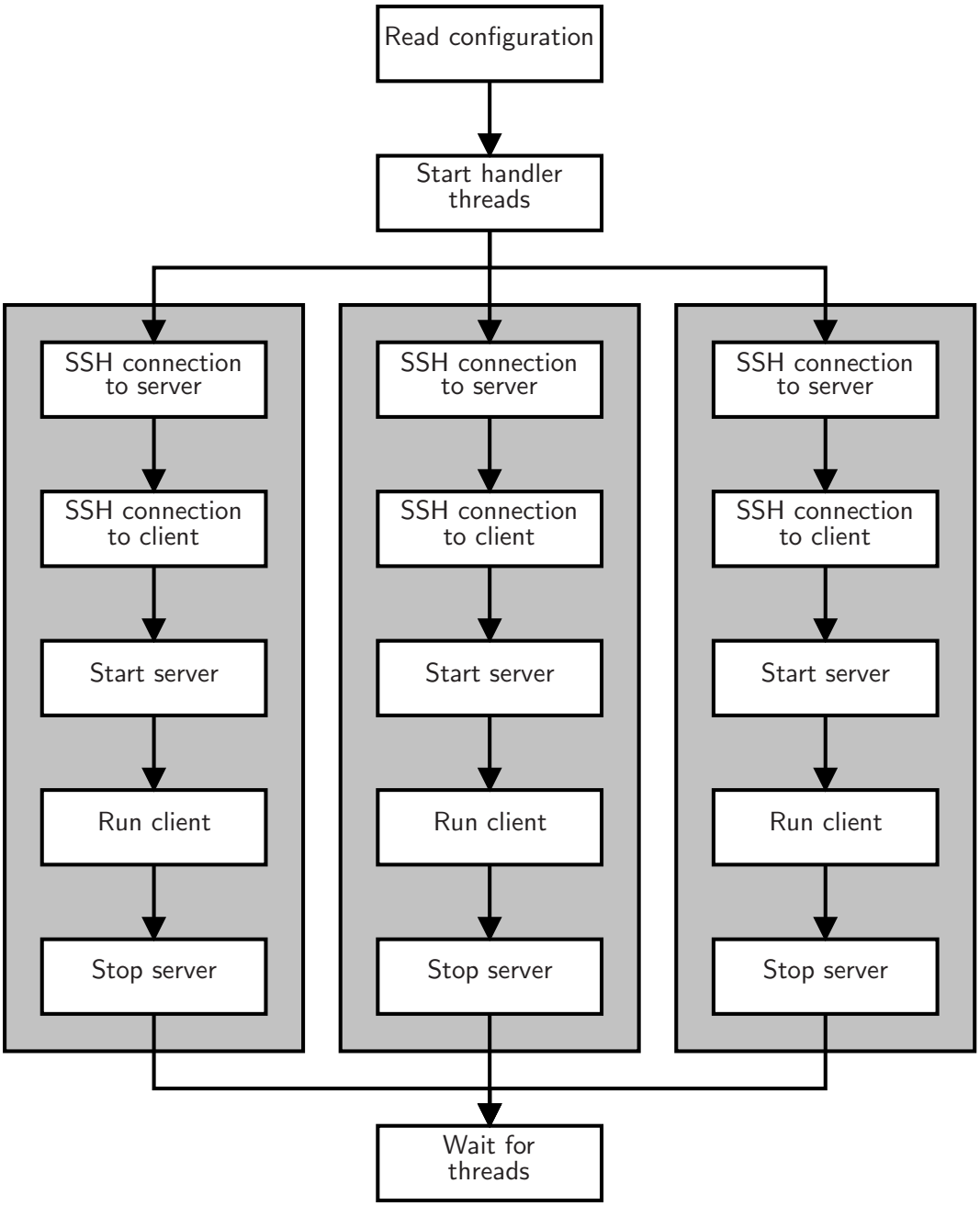


Figure 7.1: Control flow overview of the remote control system with three transmissions. Each gray box represents a transmission handler thread.

8 Performance Analysis and Exemplary Use Cases

This chapter presents an analysis of LUNA's capabilities. Section 8.1 explores the boundaries of LUNA's performance, in terms of both packet and data rates. The following two sections present examples of practical applications for LUNA: Load simulation and analysis of the resulting network behavior in an SDN-based testbed for smart grid applications in Section 8.2, and a series of round trip times measurements over an LTE mobile network in Section 8.3.

The results in Section 8.2.2 are of particular importance, because they show both the precision that LUNA can achieve, and the impact the underlying networking hardware can have on the precision of a packet generator.

8.1 Testing LUNA Performance

Aside from precision, which has been repeatedly analyzed over the previous chapters, the raw packet and data throughput is an important metric for any traffic generator. This section presents an analysis for LUNA performance, including a comparison with the results from Botta et al. [2].

All experiments in this section were done using the setup shown in Figure 8.1. Packets were sent between two hosts directly linked with Gigabit Ethernet. Both hosts had Intel CPUs with four cores each, running at 3.2 GHz, 7.7 GB (Host A) and 3.7 GB (Host B) RAM, and integrated Realtek RTL8111/8168B Gigabit Ethernet interfaces for the test link. Both hosts were running Linux 3.2.46 with PREEMPT_RT from the Debian package repository (package `linux-image-3.2.0-4-rt-amd64`). Different hardware aside, this is identical to the setup used by Botta et al. [2].

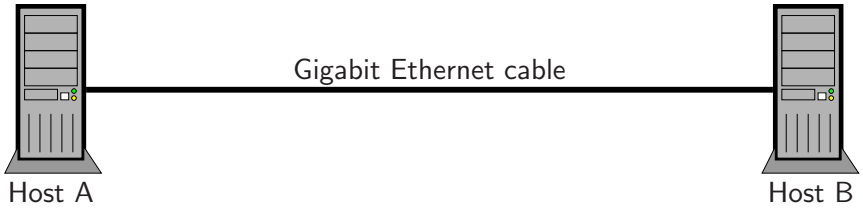


Figure 8.1: Measurement setup for performance tests

8.1.1 Short Packet Intervals and Receive Buffer Size

In preliminary experiments with very short ISTs while preparing the measurement setup described above, the server protocols showed that an extremely high packet loss had occurred. Since packet loss is detected based on sequence numbers, and the client increases sequence numbers only after sending a packet, the client must have been fast enough to actually send packets with high sequence numbers, but these packets were lost before reaching the server.

The same connection had been extremely reliable in previous tests, usually operating without any packet loss at all. This indicated that the problem might have been on the server side, and more precisely caused by receive buffer overflows. The receive buffer is a part of the Linux kernel's networking stack. Incoming packets are stored there until they can be passed to user space. However, its size is limited, and if packets continually arrive faster than user space can retrieve them, it will overflow at some point. How quickly an overflow occurs depends on both buffer size and the rate at which packets accumulate.

Both default and maximum receive buffer sizes (in bytes) for sockets can be configured through the sysctls[1] `net.core.rmem_default` and `net.core.rmem_max`, respectively. The defaults values on both systems involved in the following experiments were

```
net.core.rmem_default = 229376
net.core.rmem_max = 131071
```

and both were equally changed to 425984 bytes.

Indeed, using this increased receive buffer size greatly reduced the observed packet loss for measurements with short durations, but tests with higher durations still show very high packet loss, as described in the following section.

8.1.2 Speed Test with 6 µs IST

This experiment was aimed at comparing the performance of LUNA with those of the traffic generators analyzed by Botta et al. [2]. A target IST of 6 µs should result in a packet rate slightly above the maximum ($1.6 \cdot 10^5 \mathrm{pps}$[2]) they used in the experiment behind the graphs shown on the left side of Figure 2 in their paper. Like Botta et al. [2], I used the smallest packet size available (21 bytes for LUNA) in the measurement setup shown in Figure 8.1.

The resulting theoretical data rate R can be calculated from packet interval T or packet frequency f and packet size S:

$$R = f \cdot S = \frac{S}{T} \tag{8.1}$$

[1] Sysctls are runtime configuration variables for the Linux kernel, available through the `proc` filesystem. In the default filesystem layout they are placed at `/proc/sys/`.

[2] pps: **p**ackets **p**er **s**econd

$$\Rightarrow R = \frac{21\text{byte}}{6\mu\text{s}} = 3.5 \cdot 10^6 \frac{\text{byte}}{\text{s}} = 2.8 \cdot 10^7 \frac{\text{bit}}{\text{s}} = 28 \frac{\text{Mbit}}{\text{s}} \tag{8.2}$$

Using this setup, two tests were done: the first one with a transmission time of 1 s, the second one with 10 s. The resulting IAT distributions can be seen in Figure 8.2, and their metrics in Table 8.1.

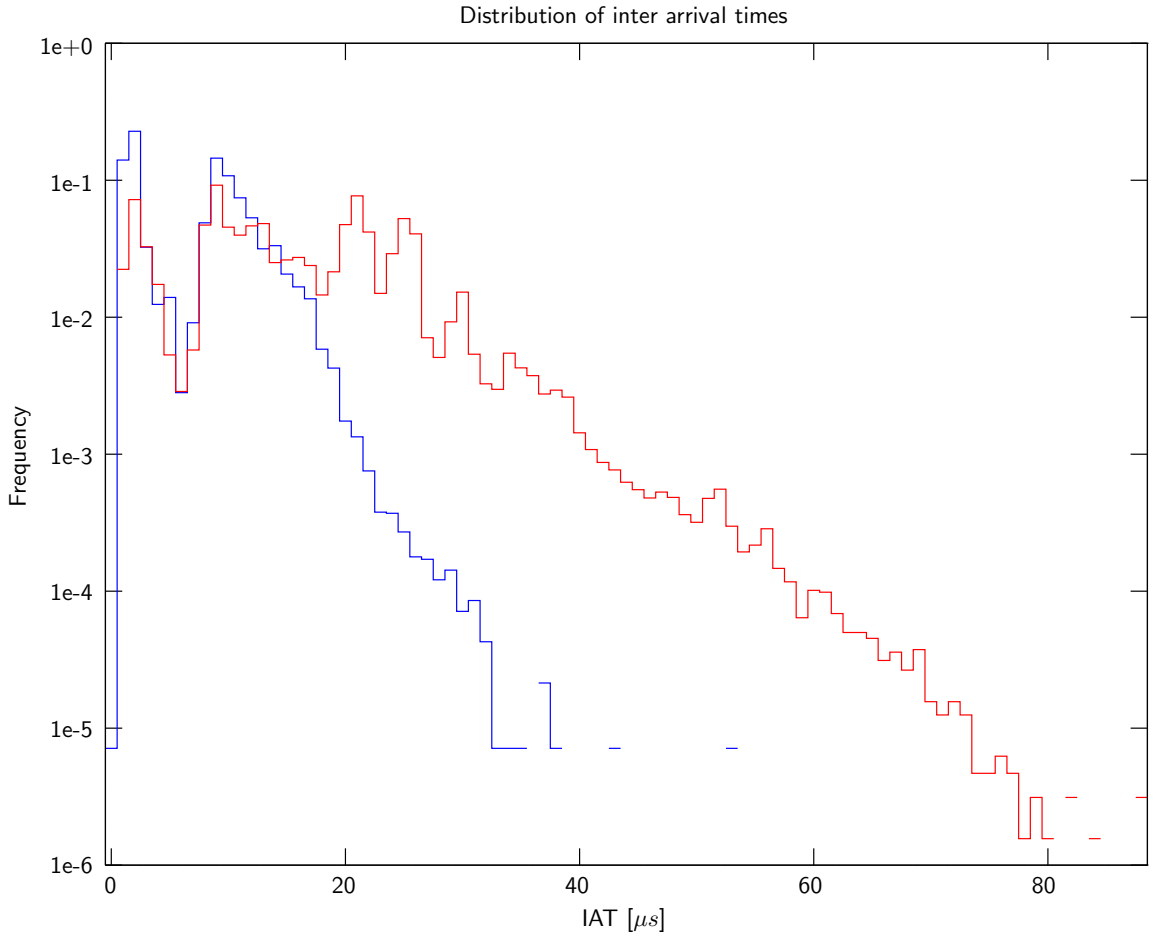


Figure 8.2: IAT distribution based on 6 µs IST, direct Ethernet connection, with 1 s measurement duration (blue) and 10 s measurement duration (red)

Experiment duration	Max. IAT	Min. IAT	Average IAT	Median IAT	σ(IAT)
1 second	53	-416	7	9	5
10 seconds	29005	1	15	14	37

Table 8.1: Metrics of the IAT distributions with IST of 6 µs, all IAT values in µs

Both distributions show peaks not exactly at, but rather around the configured packet interval, with the distribution of the 10 seconds measurement being much more drawn out to the right. Both this wider distribution and its approximately double average and median values can be explained by the packet loss statistics shown in Table 8.2, with the theoretical packet count c_t calculated from packet interval T and

Experiment duration	Recorded packets	Highest recorded sequence number	Theoretical packet count	Packets lost based on recorded packets	Packets lost based on theoretical packet count
1 second	140375	166665	166667	26291 (15.8%)	26293 (15.8%)
10 seconds	640753	1092929	1666667	452177 (41.4%)	1025914 (61.6%)

Table 8.2: Packet loss statistics of the experiments with 6 µs IST

measurement duration t_m under Equation 8.3. In the longer experiment, three packets were recorded that arrived out of order.

The minimum IAT for the measurement taken over one second may need an explanation, because the IATs of packets recorded in the order they arrive should never be negative. However, on multiprocessor systems the clock sources of different CPUs may be out of sync, leading to "*bogus results* if a process is migrated to another CPU." [39] Presumably, the negative IAT measurements were caused by the kernel handling arriving packets on different CPUs[3].

$$c_t = \frac{t_m}{T} \tag{8.3}$$

The significantly larger packet loss in the longer experiment explains the larger IATs, because the server cannot record *theoretical* packets, and they are therefore lost for the evaluation of *recorded* IATs. Considering that the percentage of lost packets (based on records) in the longer experiment is approximately twice that of the shorter experiment, the higher average and median IATs, as well as their wider distribution, are not surprising.

On the other hand, the maximum sequence number recorded in the 1 second measurement shows that the *client* could keep up with the requested packet interval, because the client only increases the sequence number after sending a packet. However, this is difficult to reproduce, because packet losses are not constant, as can be seen in Figures 8.3 and 8.4, which show the datarates recorded in both measurements over time. Note that the scales on the x-axes, which show time, differ between the two figures because of the different experiment durations.

Neither experiment reaches the theoretical datarate of $2.8 \cdot 10^7 \frac{\text{bit}}{\text{s}}$ as calculated in Equation 8.2, which is to be expected considering the packet loss rates from Table 8.2. Both figures show significant variations in the datarate, with an increase towards the end, which is likely due to the fact that at the end of the transmission the server can process packets remaining in the receive buffer at any speed, because without new packets arriving there is no threat of overflow.

[3] Both test host are equipped with what is commonly called one multicore CPU. Technically, however, each core in a "multicore CPU" is a separate CPU, integrated with the other CPUs on a single chip, and usually sharing some cache levels and interfaces with them.

Figure 8.3 is more detailed, because the time scale is wider, and clearly shows the pattern expected from buffer overflows: irregularly alternating sections of higher and lower datarate.

Ethernet frames have a minimum payload size of 46 bytes. If the provided payload is smaller, padding must be used. Combined with the 38 bytes needed for the frame header, footer, and inter frame gap [52, Sec. 3.1.1 and 4.4.2], this results in a minimum frame size of 84 bytes. The highest theoretically possible frame rate, equal to highest theoretically possible packet rate P_{max} in this experiment, can be calculated from the theoretically available datarate of a Gigabit Ethernet link R_{Gbit} and the minimum frame size S_{min} (Equation 8.4).

$$P_{max} = \frac{R}{S_{min}} = \frac{10^9 \frac{\text{bit}}{\text{s}}}{84 \frac{\text{byte}}{\text{packet}}} = \frac{10^9 \frac{\text{bit}}{\text{s}}}{672 \frac{\text{bit}}{\text{packet}}} \approx 1.5 \cdot 10^6 \frac{\text{packets}}{\text{s}} \tag{8.4}$$

P_{max} leads directly to the shortest theoretically possible time it takes to transmit one minimum size frame over the Gigabit Ethernet link, $t_{Frame,min}$ (Equation 8.5).

$$\frac{1}{P_{max}} \approx 6.72 \cdot 10^{-7} \frac{\text{s}}{\text{packet}} \Rightarrow t_{Frame,min} = 672 \text{ ns} \tag{8.5}$$

In [2], the highest packet rate any of the analyzed traffic generators could handle was approximately $1.4 \cdot 10^5 \text{pps}$ (D-ITG), equivalent to roughly 7 µs IAT, so the maximum sustained packet rate was less than one tenth of the theoretical maximum. However, when studying the data acquisition system for CERN's LHCb experiment, which uses a raw IP based protocol over Gigabit Ethernet, Barczyk et al. [53] found that the Ethernet frame rate would not exceed $2.8 \cdot 10^5 \text{pps}$, equivalent to approximately 3.6 µs IAT.

The following conclusions concerning the performance of LUNA can be drawn from these experiments:

- The LUNA *client* is able to generate packets with an interval of 6 µs using the static generator. This is comparable to the speed Botta et al. [2] measured with D-ITG, and in the same order of magnitude as the practical maximum found by Barczyk et al. [53].

- However, the *server* cannot keep up with this speed. For peaks in the packet rate or short term measurements, a large receive buffer in the kernel can compensate for this limitation.

It is important to note that the limiting factor here is the *packet rate*, not the *datarate*, as shown in Section 8.1.3 below. To support sustained packet rates at the tested level, packet processing in the server must become faster. This will likely require implementing multiple threads, which handle different stages of packet processing in parallel, and thus only be possible on systems with multiple CPUs.

8.1.3 High Datarate Test

Another possible limitation that Botta et al. [2] analyzed is the datarate that a given tool can provide, as shown on the *right* side of Figure 2 in their paper. However, there must be some kind of error in that figure, because the maximum datarate noted on both x- and y-axis is $10.6 \cdot 10^5 \frac{\text{bit}}{\text{s}}$, which is only $1.06 \frac{\text{Mbit}}{\text{s}}$, and thus far below the datarates of $350 \frac{\text{Mbit}}{\text{s}}$ to $1 \frac{\text{Gbit}}{\text{s}}$ they claim to have tested in the associated text. It seems likely, though, that only the scale is off, more precisely the exponent in 10^5. Assuming that it should be 10^8, D-ITG, the generator with the best result in that experiment, would deliver about $700 \frac{\text{Mbit}}{\text{s}}$, which seems reasonable.

Like the one by Botta et al. [2], the following analysis uses the maximum packet size possible without IP fragmentation. With the normal Ethernet MTU of 1500 bytes, minus IPv4 (20 byte) [54] and UDP (8 byte) [17, p.1] headers, the largest possible LUNA payload, and therefore the packet size to use, is $S = S_{max} = 1472 \text{ bytes}$. Based on the statement in [2] that the tools started to deviate from expected behavior starting from datarates of about $500 \frac{\text{Mbit}}{\text{s}}$, and assuming that only the scale exponent is off in their figure, $R_1 = 500 \frac{\text{Mbit}}{\text{s}}$ and $R_2 = 800 \frac{\text{Mbit}}{\text{s}}$ seem reasonable datarates to test with LUNA.

Based on Equation 8.1, it is possible to calculate the packet intervals needed to configure LUNA:

$$R = \frac{S}{T} \Leftrightarrow T = \frac{S}{R} \tag{8.6}$$

$$T_1 = \frac{S}{R_1} = \frac{1472 \text{ byte}}{500 \cdot 10^6 \frac{\text{bit}}{\text{s}}} = \frac{11776 \text{ bit}}{500 \cdot 10^6 \frac{\text{bit}}{\text{s}}} \approx 24 \mu s \tag{8.7}$$

$$T_2 = \frac{S}{R_2} = \frac{1472 \text{ byte}}{500 \cdot 10^6 \frac{\text{bit}}{\text{s}}} = \frac{11776 \text{ bit}}{800 \cdot 10^6 \frac{\text{bit}}{\text{s}}} \approx 15 \mu s \tag{8.8}$$

LUNA accepts only whole microseconds as packet intervals, so the calculated intervals must be rounded to integers. This leads to slightly different theoretical datarates:

$$R_1 = \frac{S}{T_1} = \frac{1472 \text{ byte}}{24 \ \mu\text{s}} = 490.7 \frac{\text{Mbit}}{\text{s}} \tag{8.9}$$

$$R_2 = \frac{S}{T_2} = \frac{1472 \text{ byte}}{15 \ \mu\text{s}} = 785.1 \frac{\text{Mbit}}{\text{s}} \tag{8.10}$$

Other than transmission configuration, the measurement setup was the same as in the experiments in Section 8.1.

Measurements were taken with both configurations for 60 seconds each. The datarates recorded at the server are shown in Figure 8.5. Despite a few lost packets, the $490.7 \frac{\text{Mbit}}{\text{s}}$ transmission matches

the expected rate very well, while the $785.1\frac{\text{Mbit}}{\text{s}}$ one shows a far higher variation around approximately $700\frac{\text{Mbit}}{\text{s}}$.

Target Datarate	IST [µs]	Packets recorded	Highest sequence number	Lost packets	Average Datarate
490.7 Mbit/s	24	2482026	2499999	17974 (0.7%)	487 Mbit/s
785.1 Mbit/s	15	3564837	3999999	435163 (10.9%)	700 Mbit/s

Table 8.3: Packet loss statistics from datarate experiments

Table 8.3 shows the actual packet loss statistics for both experiments. Knowing the number of actually received packets, the overall data volume that has been transferred can be calculated easily:

$$V_1 = 2482026 \text{ packets} \cdot 1472\frac{\text{byte}}{\text{packet}} \approx 3484 \text{ Mbyte} \tag{8.11}$$

$$V_2 = 3564837 \text{ packets} \cdot 1472\frac{\text{byte}}{\text{packet}} \approx 5004 \text{ Mbyte} \tag{8.12}$$

And based ob these data volumes, the average transfer rates are:

$$R_1 = \frac{3484 \text{ Mbyte}}{60 \text{ s}} \approx 487\frac{\text{Mbit}}{\text{s}} \tag{8.13}$$

$$R_2 = \frac{5004 \text{ Mbyte}}{60 \text{ s}} \approx 700\frac{\text{Mbit}}{\text{s}} \tag{8.14}$$

This result shows that LUNA can compete with D-ITG in terms of datarate performance. Note that considering that these $700\frac{\text{Mbit}}{\text{s}}$ are a net data rate, and UDP, IP, and Ethernet headers for each packet would need to be added when comparing with the theoretical maximum datarate which the Gigabit Ethernet connection should provide.

Standard Ethernet frames have a maximum payload size of 1500 bytes, plus 38 bytes for the frame header, footer, and inter frame gap [52, Sec. 3.1.1 and 4.4.2]. Adding the 20 byte IPv4 [54] and 8 byte UDP [17, p.1] headers, this results in 66 bytes of overhead per 1538 byte Ethernet frame. In this experiment, all IP packets sent by LUNA utilize this maximum payload size, so the protocol overhead is the minimum of approximately 4.3% per packet as calculated in Equation 8.15.

$$\frac{66 \text{ byte}}{1538 \text{ byte}} \approx 4.3\% \tag{8.15}$$

Ignoring other overhead like ARP[4] packets, this leads to a theoretical maximum UDP payload datarate of $957\frac{\text{Mbit}}{\text{s}}$, assuming that packets align perfectly and no collisions occur.

[4] Address Resolution Protocol, see [55]

Interestingly, the IAT distributions (Figure 8.6) have only small peaks corresponding to the configured ISTs, far bigger ones at lower values, and also many higher IATs. This may be a hardware issue, as the similar analysis in Section 8.2.2 shows.

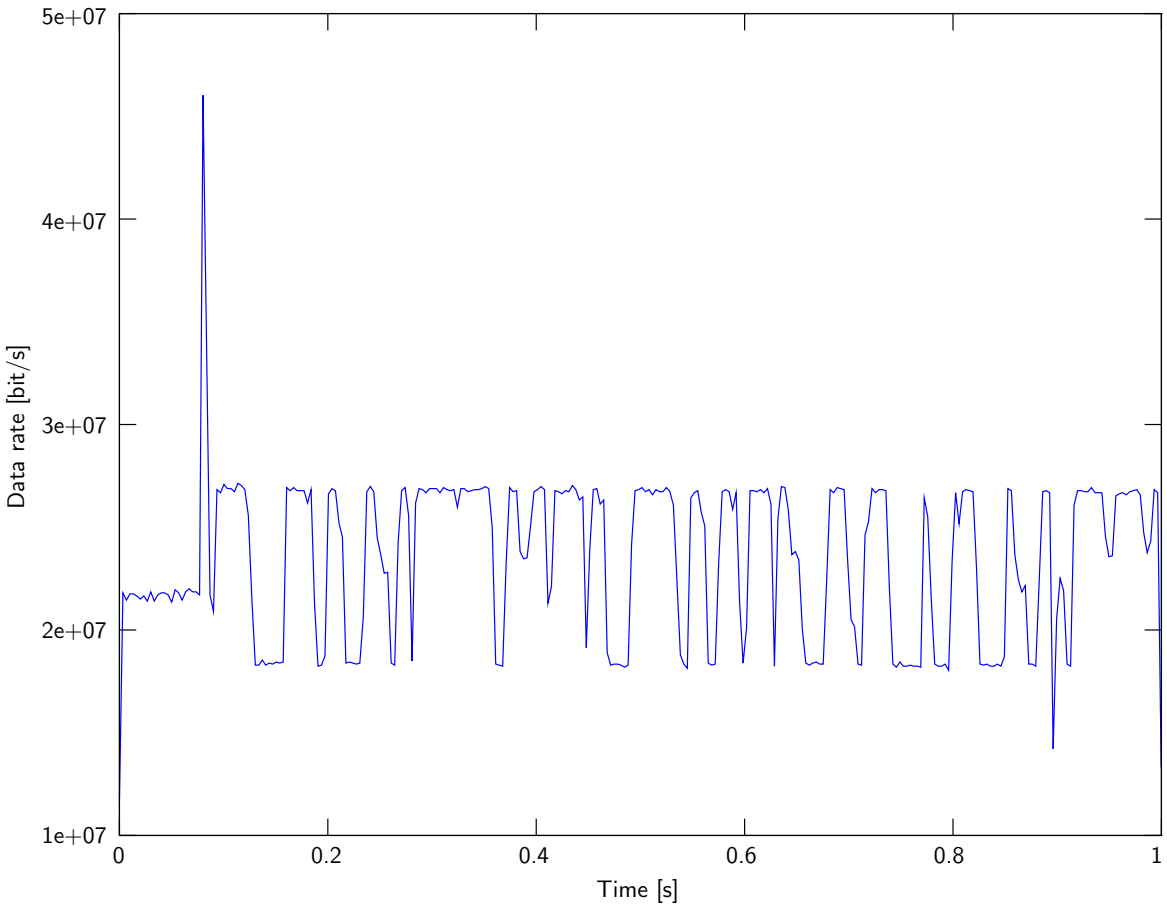


Figure 8.3: Datarate over time with 6 µs IST, direct Ethernet connection, 1 s measurement duration

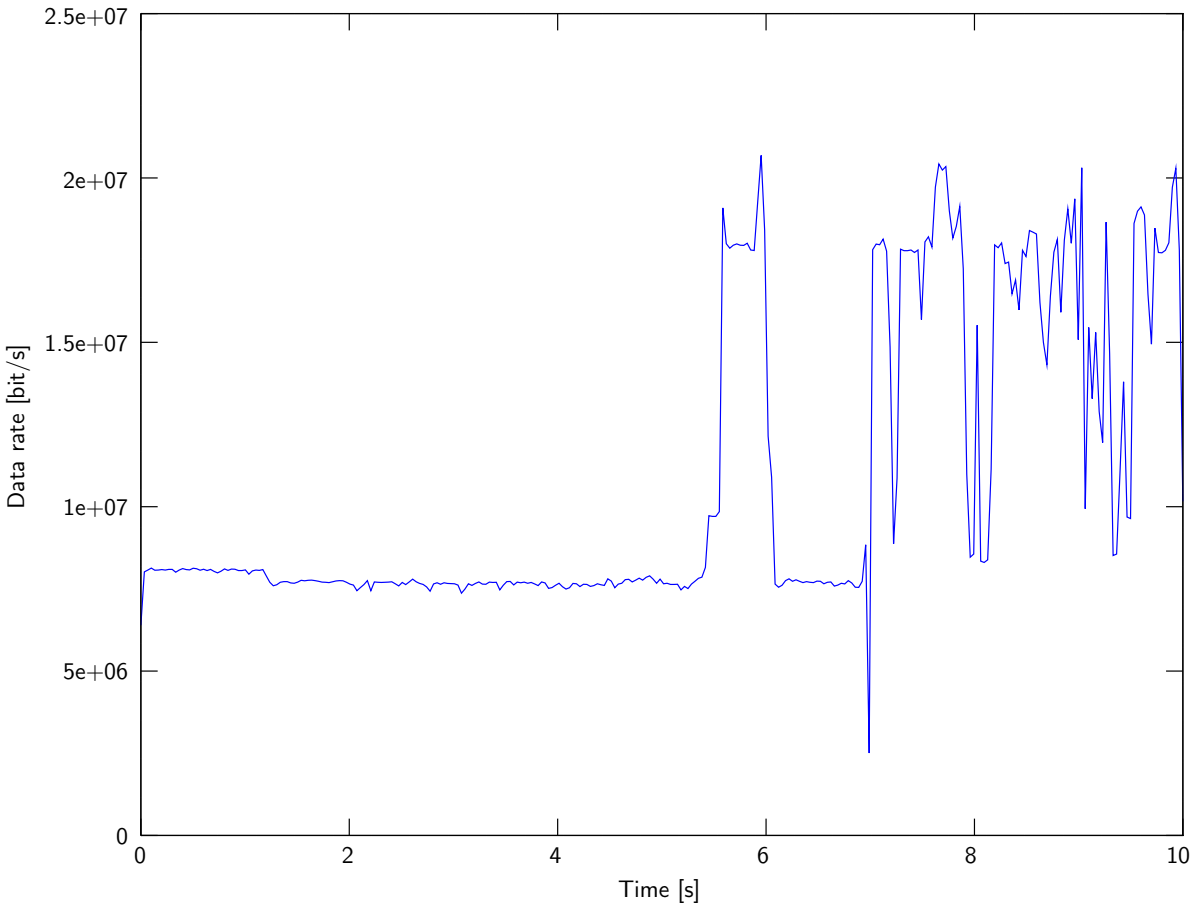


Figure 8.4: Datarate over time with 6 µs IST, direct Ethernet connection, 10 s measurement duration

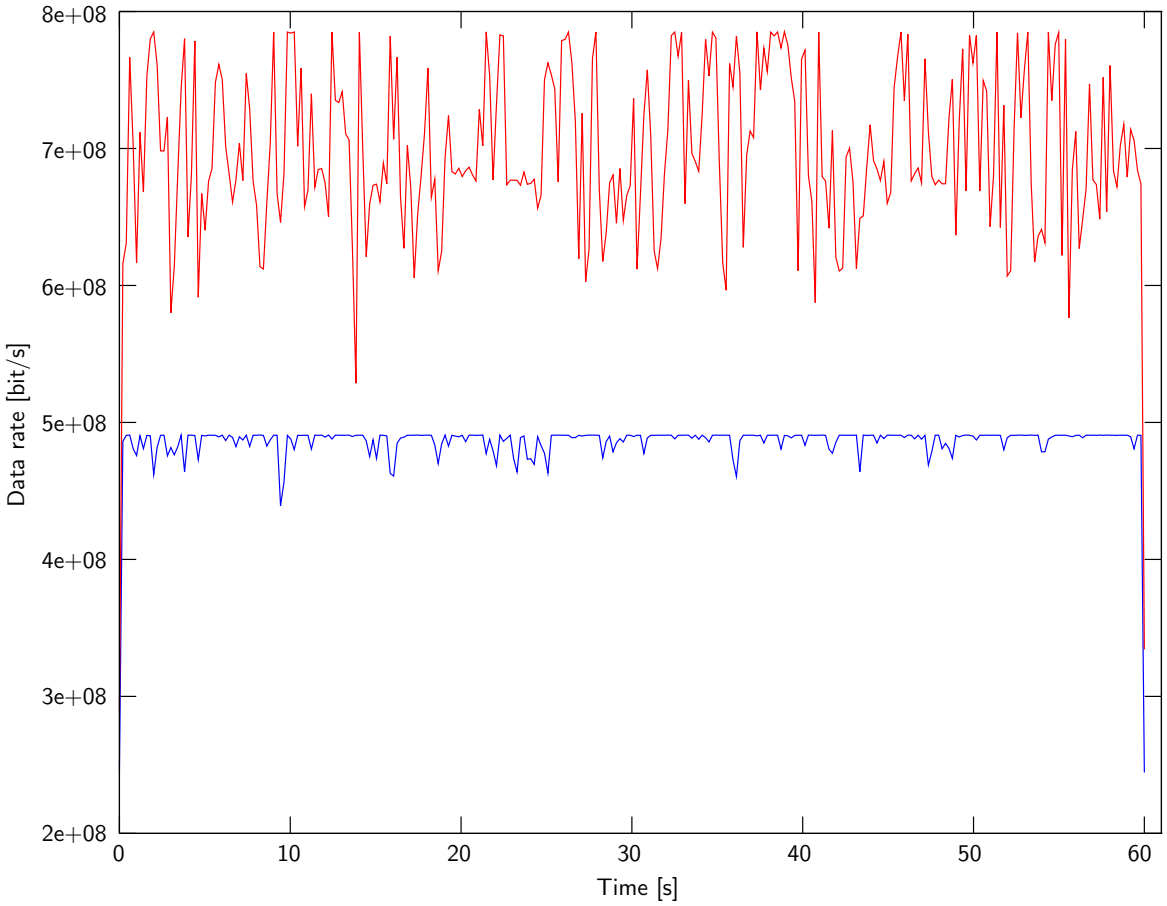


Figure 8.5: Datarates with one 1472 byte packet every 24 µs (blue) and every 15 µs (red), over a direct Gigabit Ethernet link

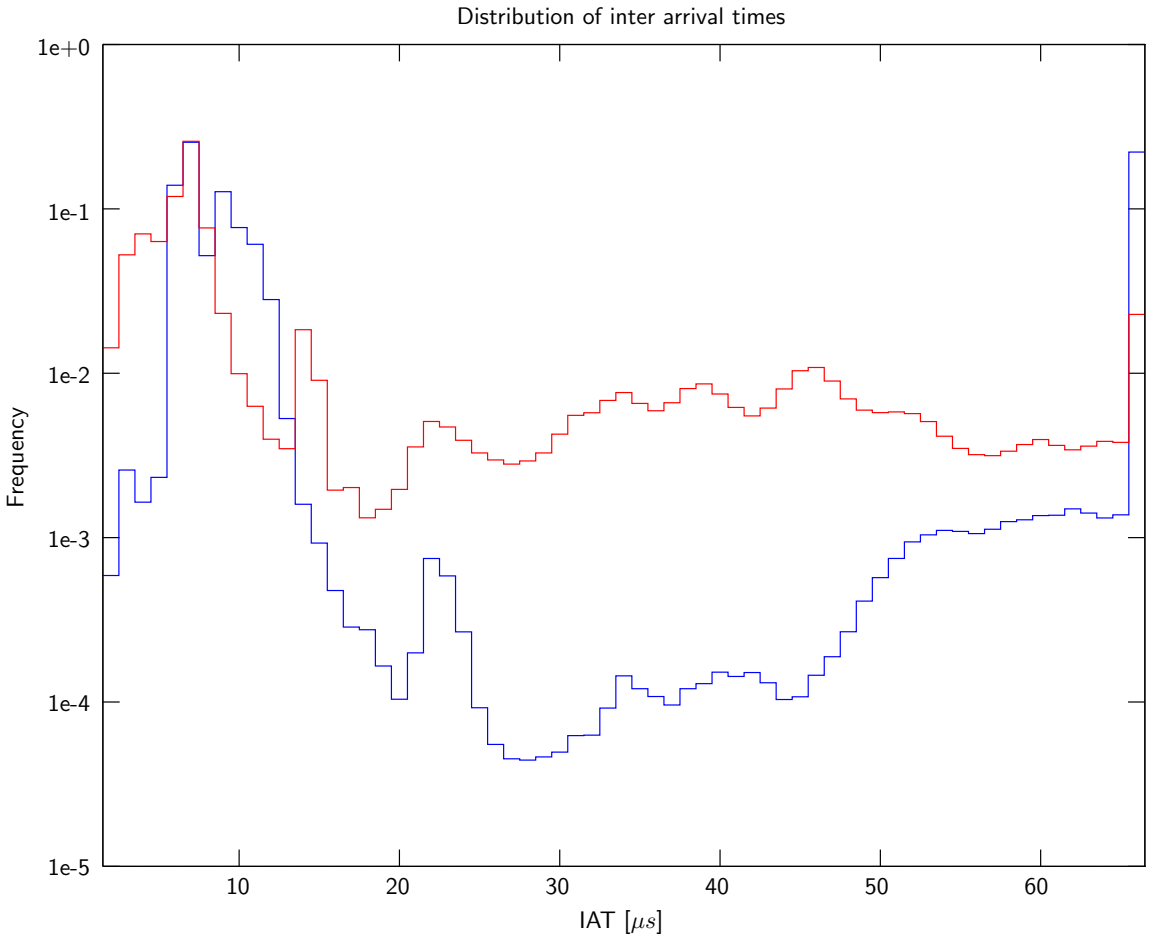


Figure 8.6: IAT distributions for the high datarate tests with one 1472 byte packet every 24 µs (blue) and every 15 µs (red), over a direct Gigabit Ethernet link

8.2 Example Use Case: CNI SDN Testbed

The TU Dortmund Communication Networks Institute (CNI) is currently in the process of building a testbed for smart grid applications, based on a software defined network (SDN). This network was used to test the practical application of LUNA.

The measurement setup is shown in Figure 8.7. It represents what is called a "smart substation", where hosts A and C are "merging units", which monitor power supply conditions and report the results to hosts B and D, represent control devices like the "bay controller" or a "protection device". Thus, transmissions run from A to B and from C to D. The SDN control network, to which the routers are connected as well, is not shown. Samples should be sent every 78 or 250 µs, depending on configuration. According to the researchers involved in the project, the smart grid network works directly on the Ethernet layer. The sample packets have an Ethernet payload of 200 bytes, so UDP packets with 172 bytes payload plus IPv4 (20 byte) [54] and UDP (8 byte) [17, p.1] headers should result in equally large Ethernet frames. Details of the smart grid functionality are outside the scope of this work, please see the related standard IEC 61850 [56] for more.

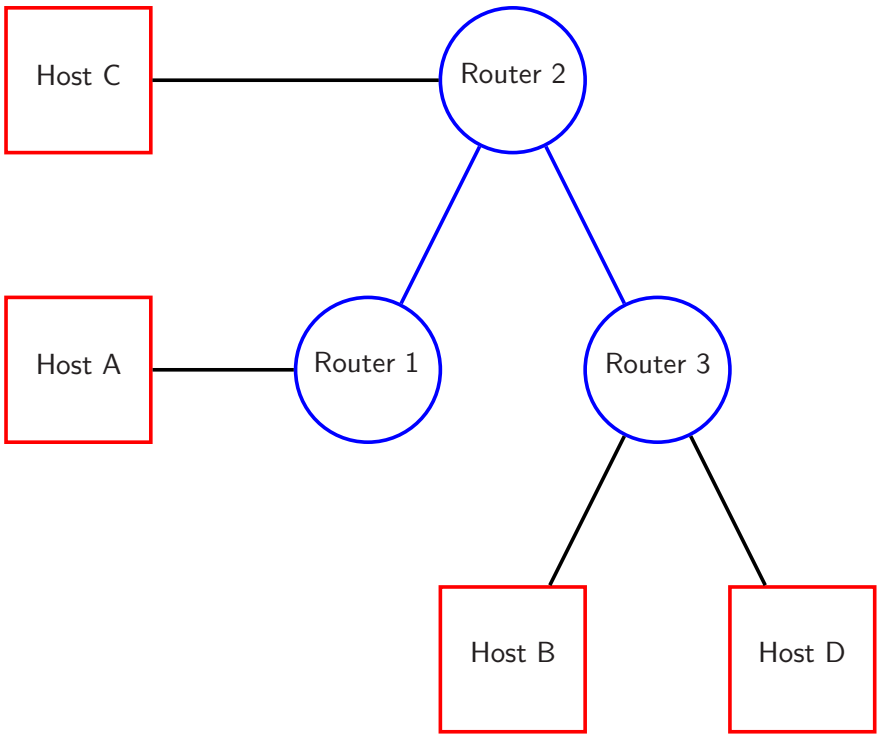


Figure 8.7: Network structure of the CNI SDN Testbed (blue), connected to four LUNA measurement hosts (red)

8.2.1 Experiments on the SDN Testbed

The distributed control mechanisms described in Chapter 7 were used to run the tests, with Host B as the control host. Originally, Host A was planned for this role, but during the initial test runs the network could not handle the outgoing UDP transmissions and SSH connections from the same host, so a change of roles was necessary. A separate control network for the measurement hosts was not available.

The machines for Host A and Host B where the same measurement hosts used in Section 8.1, C and D where laptops, one with an Intel dualcore CPU, 2 GB RAM and an Intel 82567LF Gigabit Ethernet controller (Host C), the other with an Intel quadcore CPU, 4 GB RAM and an Intel 82579LM Gigabit Ethernet controller (Host D).

Four pairs of measurements were taken, to cover all possible combinations of host pairs and packet intervals:

- 250 µs IST on both connections
- 78 µs IST on both connections
- 78 µs IST from A to B, 250 µs IST from C to D
- 250 µs IST from A to B, 78 µs IST from C to D

In all figures comparing them, the blue plot shows data from the connection from A to B, while the red one shows data from the same measurement from C to D. The measurement duration was 60 seconds in all four experiments.

Figure 8.8 shows the IAT distribution with an IST of 250 µs for both connections. Both distributions show wide peaks around approximately 2500 µs, which is about ten times the intended one, with the histogram of the transmission from A to B spread out wider, which is to be expected for the slightly longer transmission (4 hops to the 3 from C to D). Packet loss was massive as shown in Table 8.4, based on recorded sequence numbers alone. Packet reordering occurred on both connections, mainly at the beginning of each transmission. This is likely a property of the SDN, which tries to classify data streams based on their first packets, and then decides how to route them.

Connection	IST [µs]	Packets recorded	Highest sequence number	Lost packets
A to B	250	25894	239983	214090 (89.2%)
C to D	250	24627	239998	215372 (89.7%)

Table 8.4: Packet loss statistics with 250 µs IST on both connections through the SDN testbed

With 78 µs IST on both connections (Figure 8.9 and Table 8.5), differences between the connections became minimal, while the packet loss increased further to about 99.7% (!) on both connections. Peaks in the histograms showing some packets arriving closer to each other became more significant, but that

may be a result of the main peak moving further to the right. Once again, packet reordering caused the first packet of each connection to arrive later than a few with higher sequence numbers, but it was less prevalent than in the previous experiment.

Connection	IST [µs]	Packets recorded	Highest sequence number	Lost packets
A to B	78	2437	762946	760510 (99.7%)
C to D	78	2352	769106	766755 (99.7%)

Table 8.5: Packet loss statistics with 78 µs IST on both connections through the SDN testbed

Behavior with a different IST for each transmission varies significantly depending on which IST is assigned to which pair of hosts. With 78 µs IST from A to B and 250 µs from C to D (Figure 8.10 and Table 8.6), each transmission behaves similarly to the way it does in the experiments with equal ISTs.

If ISTs are assigned the other way around, however, the behavior of the connection with both the shorter distance *and* IST (C to D) becomes dominant (Figure 8.11 and Table 8.7).

The most important conclusion to be drawn from the measurement results is that the SDN in its current form is not prepared to handle high numbers of packets arriving at fairly short intervals.

Based on Equation 8.1, it is possible to calculate the payload data rate for each transmission:

$$R = \frac{S}{T} = \frac{172\text{byte}}{78\mu\text{s}} = 2.21 \cdot 10^6 \frac{\text{byte}}{\text{s}} = 1.76 \cdot 10^7 \frac{\text{bit}}{\text{s}} = 17.6 \frac{\text{Mbit}}{\text{s}} \tag{8.16}$$

Even with two such streams and considering protocol overhead, a network based on 100baseT Ethernet links should not experience packet loss rates far above 80 or even 90%. This suggests that the problem lies in the SDN software, which cannot handle packets fast enough, at least not using the available hardware resources.

8.2.2 System Verification

To verify that the high packet loss rates observed in Section 8.2.1 are indeed results of the underlying network, both pairs of hosts (A and B, C and D) were subjected to additional tests using 78 µs ISTs over direct Ethernet links, with transmission durations of 60 seconds, equal to that of the SDN experiments. The resulting IAT distributions can be seen in Figure 8.12. For comparison, a loopback measurement taken on Host A with a transmission duration of 2 seconds is included as well.

Both the loopback test and the transmission between C and D show the expected properties: A strong peak at the configured IST, with some distribution around it. The peak from the loopback test is much sharper, while the C to D transmission shows another, smaller peak of very low IAT values. The transmission from A to B, however, shows a peculiar IAT distribution, with almost no IATs falling in the expected range. Instead, there is a sharp peak of very low IATs and a much wider one around approximately 250 µs. Since both pairs of hosts were configured with the same IST and packet size but

Connection	IST [µs]	Packets recorded	Highest sequence number	Lost packets
A to B	78	3060	769165	766106 (99.6%)
C to D	250	35269	239996	204728 (85.3%)

Table 8.6: Packet loss statistics for connections through the SDN testbed with ISTs of 78 µs (A to B) and 250 µs (C to D)

are equipped with different Ethernet hardware, this is most likely a characteristic of the hardware or the associated driver.

The average IAT is nonetheless very close to the intended value for all three experiments, as can be seen in Table 8.8. Interestingly, while the IAT distribution between the hosts with Realtek Ethernet controllers (A and B) deviates much more visibly from the intended value, the *maximum* deviations are smaller than those between the hosts with the Intel Ethernet controllers (C and D). This shows the importance of carefully choosing the hardware for measurement systems.

Another important result here is the IAT standard deviation measured in the loopback test (marked red in Table 8.8). At just one microsecond it shows the high precision of LUNA packet generation. For the negative minimum IATs the statements from Section 8.1.2 apply: They are artifacts of out of sync CPU clocks on an SMP[5] system.

In conclusion, both LUNA and the hardware of the measurement hosts used in the SDN experiments can very well support the required data and packet rates, so the high packet loss rates can confidently be attributed to the network itself.

[5]SMP: **S**ymmetric **m**ulti**p**rocessing

Connection	IST [µs]	Packets recorded	Highest sequence number	Lost packets
A to B	250	2811	239791	236981 (98.8%)
C to D	78	1988	768635	766648 (99.7%)

Table 8.7: Packet loss statistics for connections through the SDN testbed with ISTs of 250 µs (A to B) and 78 µs (C to D)

Experiment	Max. IAT	Min. IAT	Average IAT	Median IAT	σ(IAT)	Packets lost
Host A (loopback), 2 sec.	110	-18	77	78	1	0
A to B (Ethernet), 60 sec.	328	-98	77	8	106	0
C to D (Ethernet), 60 sec.	1922	-150	77	78	143	1

Table 8.8: Metrics of the IAT distributions in the verification experiments, all IAT values in µs

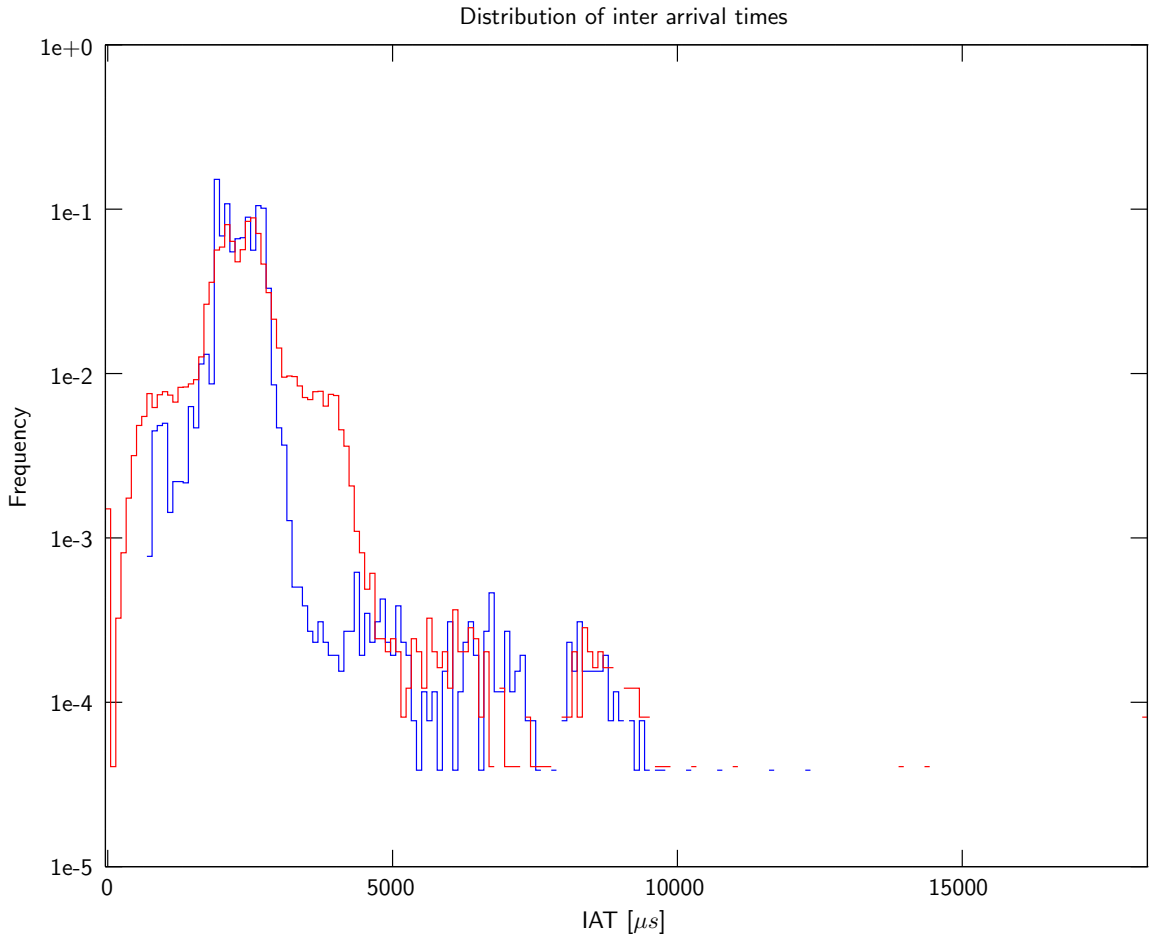


Figure 8.8: IAT distribution with 250 µs IST on both connections through the SDN testbed

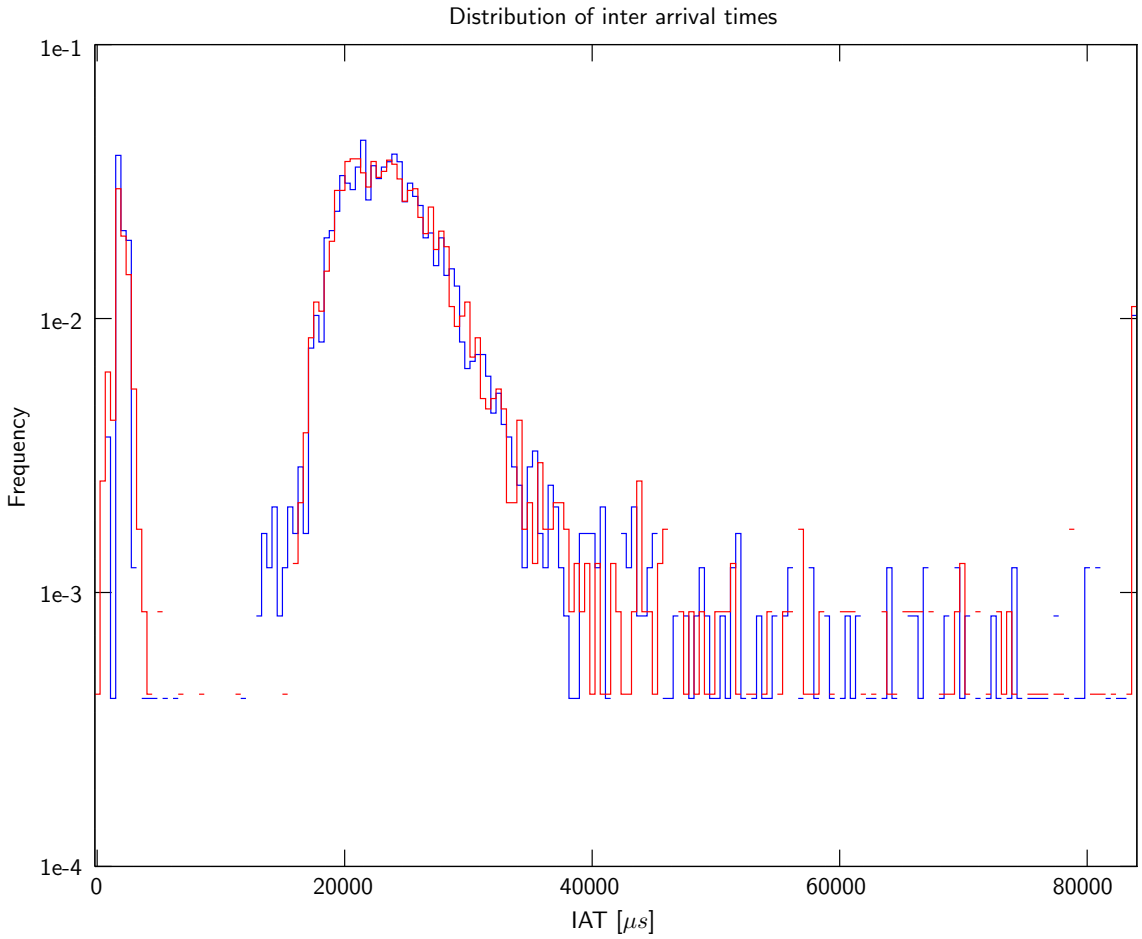


Figure 8.9: IAT distribution with 78 µs IST on both connections through the SDN testbed

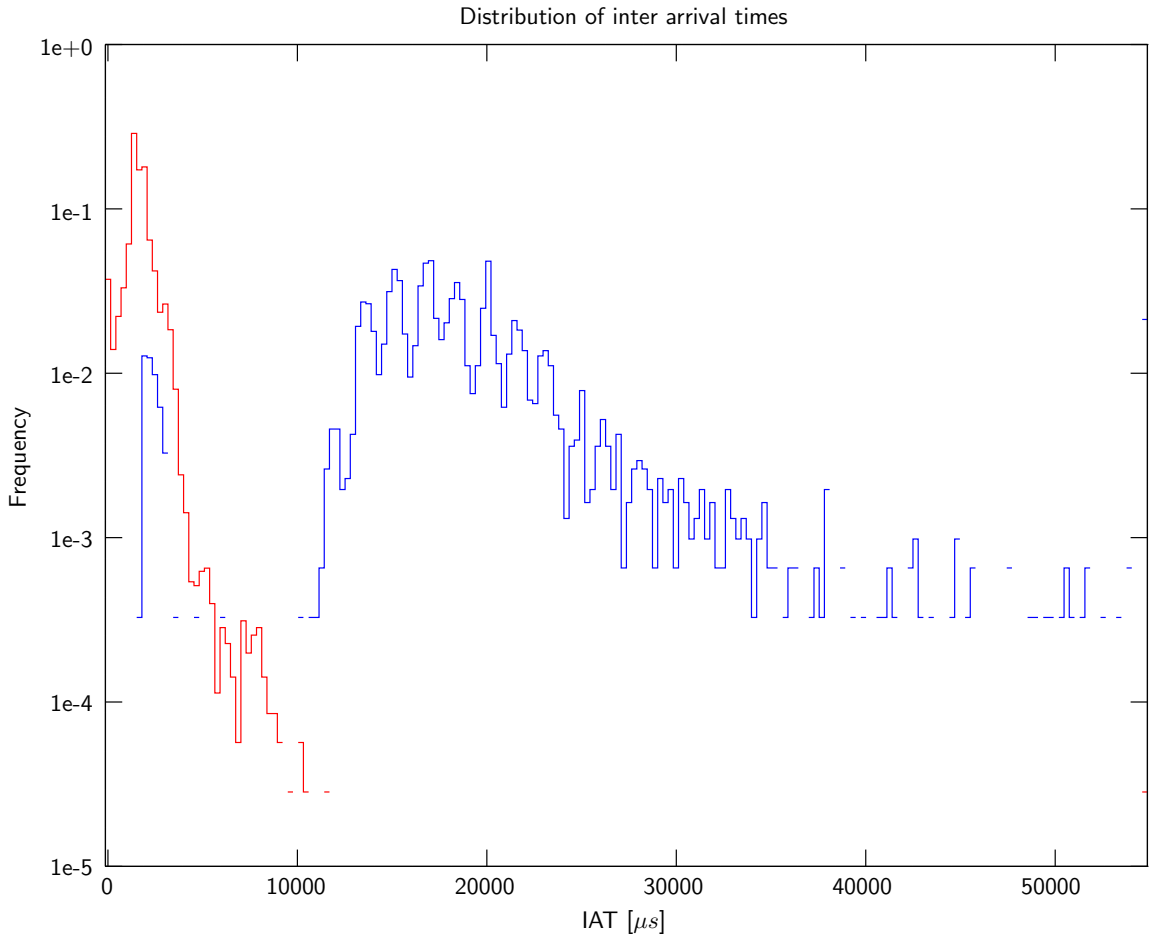


Figure 8.10: IAT distribution for connections through the SDN testbed with ISTs of 78 µs (A to B) and 250 µs (C to D)

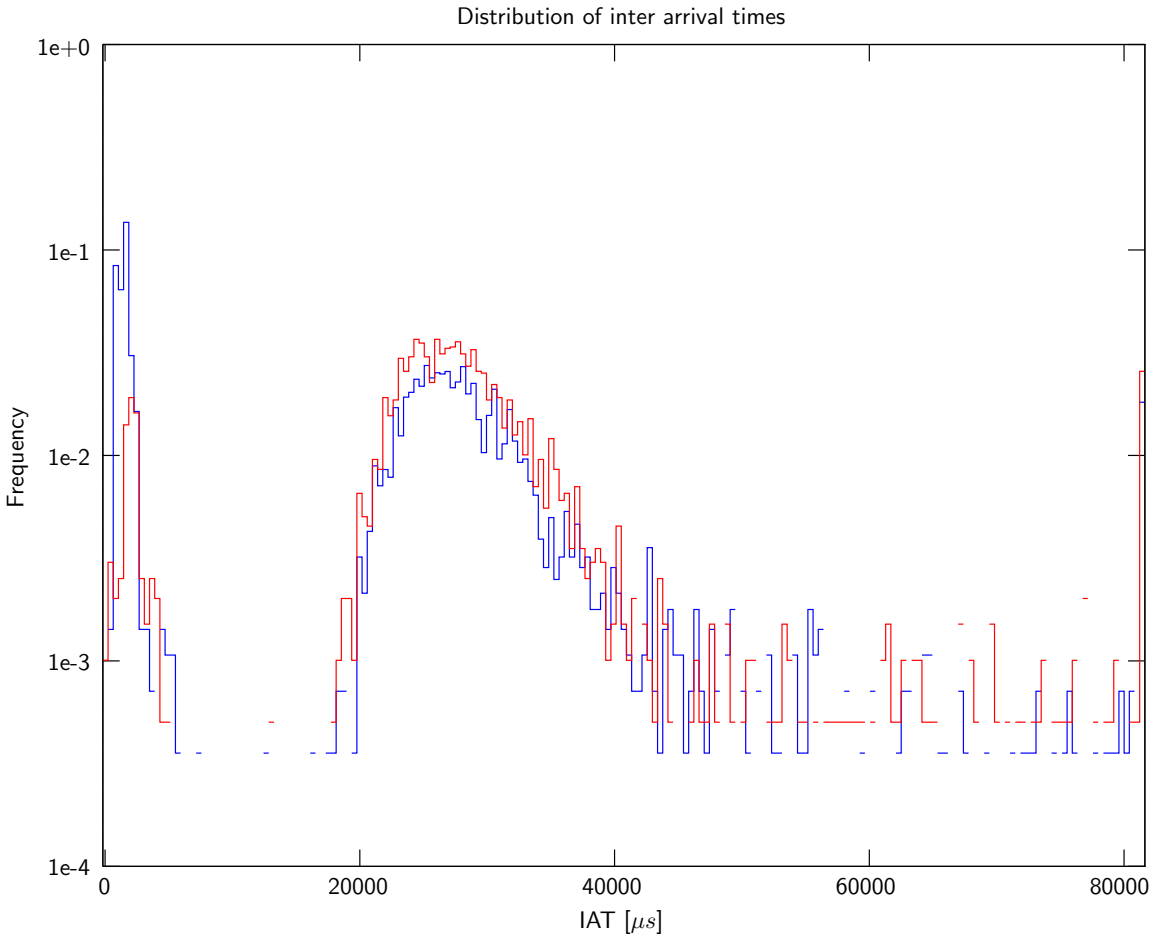


Figure 8.11: IAT distribution for connections through the SDN testbed with ISTs of 250 µs (A to B) and 78 µs (C to D)

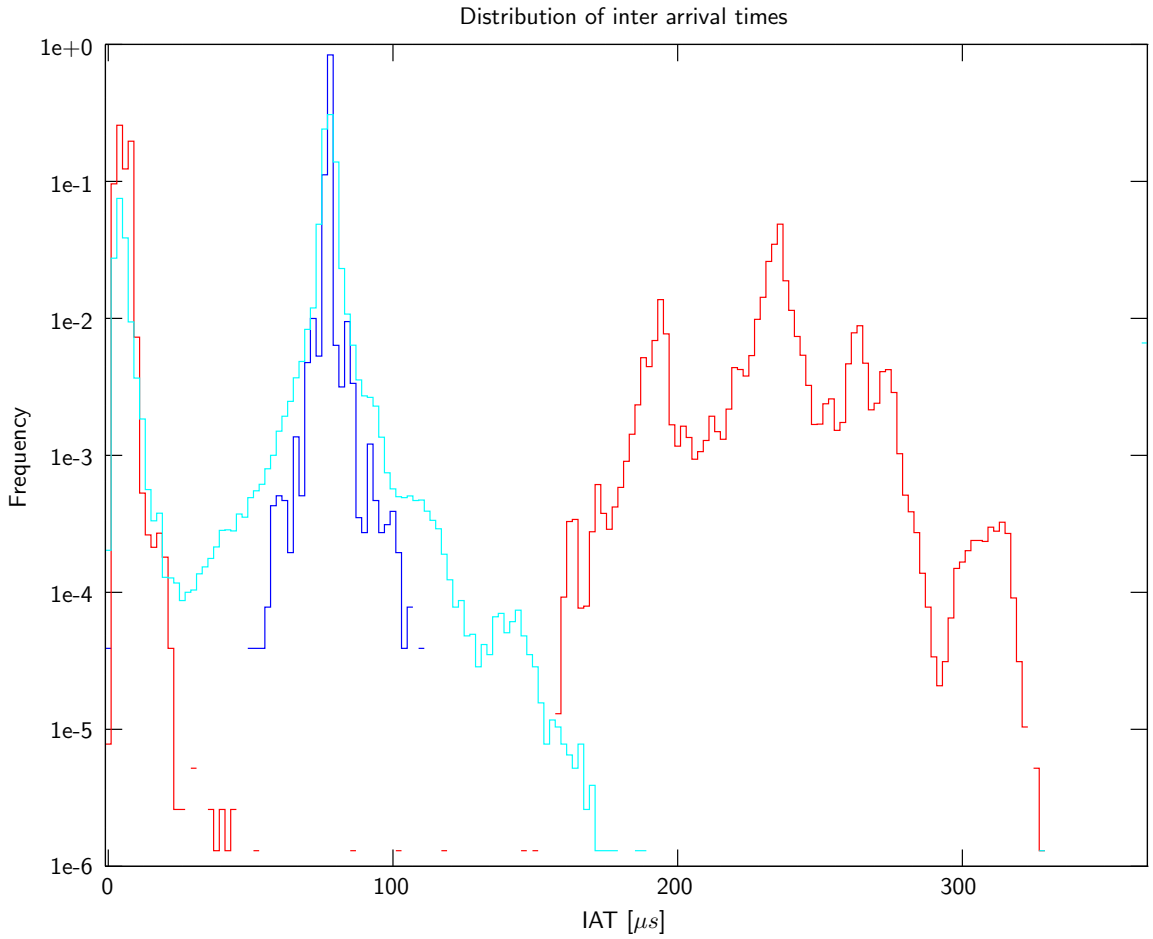


Figure 8.12: IAT distributions from tests with 78 µs IST, via loopback on Host A (blue), direct Ethernet link using Realtek Ethernet controllers between Hosts A and B (red), and direct Ethernet link using Intel Ethernet controllers between Hosts C and D (cyan)

8.3 LTE Round Trip Time Measurements

The measurement setup for another practical application of LUNA is shown in Figure 8.13. In this experiment, LUNA was used to evaluate the influence of packet size and inter send times on the round trip time of a transmissions (red) between a laptop, accessing the Internet through a commercial LTE (4G) mobile network via a USB modem, and a server located at CNI. LUNA instances on both hosts were controlled from a separate control host via LAN using the mechanisms described in Chapter 7.

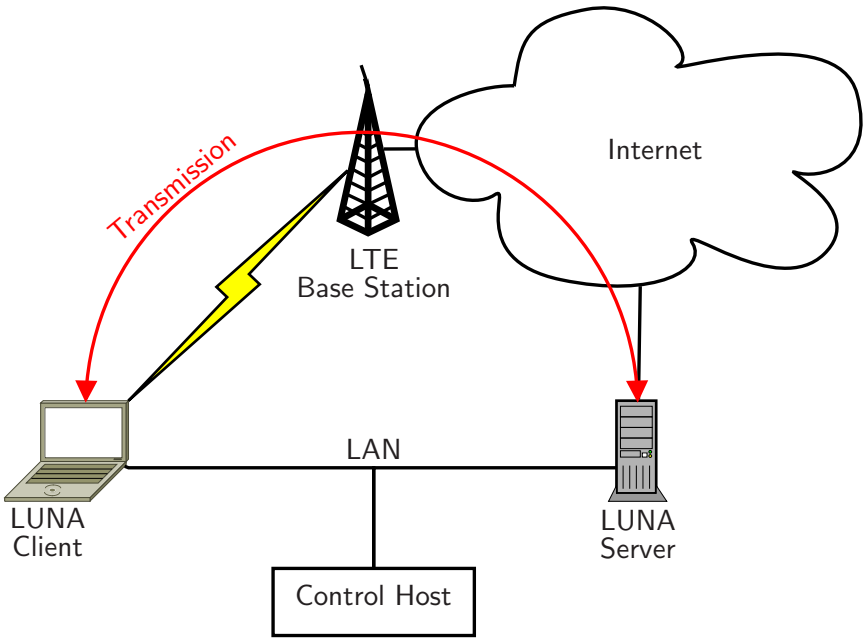


Figure 8.13: Network structure for the LTE measurements

The experiment consisted of three sets of measurements with different ISTs of 50 ms, 300 ms, and 1000 ms. For each IST, measurements were taken with packet sizes of 50 to 300 bytes in 50 byte steps, as well as 500, 700, and 1000 bytes. Each data point in Figure 8.14 shows the average of the measured RTTs for several hundred packets. The bars show the associated standard deviations.

As expected, larger ISTs slightly increase round trip times, but develop towards saturation. In particular, the differences between the curves for large ISTs (red and cyan in Figure 8.14) are small, especially considering the vast overlap between their standard deviation surroundings, which also suggest that the crossing of these two curves between 700 and 1000 bytes on the x-axis can be attributed to measurement imprecision.

While larger packet sizes tend to lead to larger round trip times, the increase remains small within the surveyed range. However, Figure 8.14 shows clearly that the RTT standard deviations increase with packet size as well, and quite substantially at that. This confirms that transfer times for larger packets vary more than those for smaller packets on the same channel.

All three curves show a similar low in RTT around 300 bytes packet size, or a little earlier in case of the

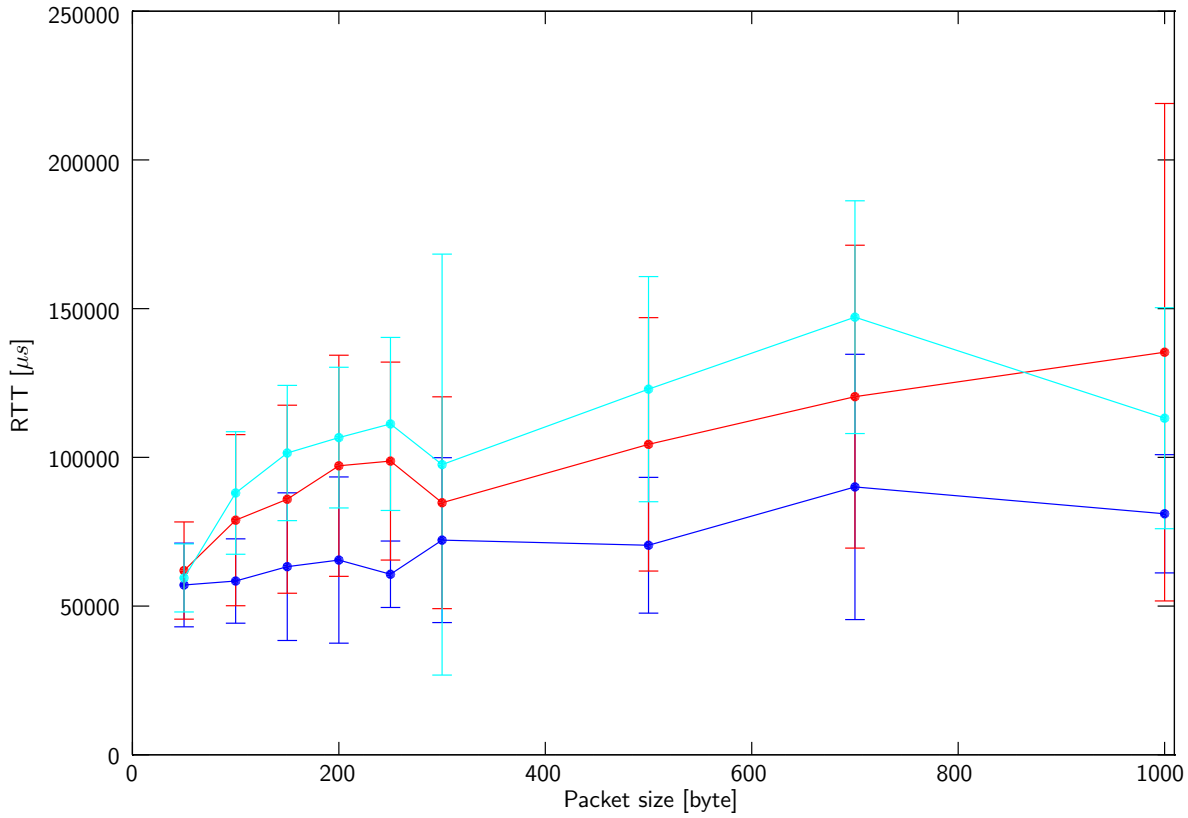


Figure 8.14: Average RTT over LTE with standard deviation bars. ISTs were 50 ms (blue), 300 ms (red), and 1000 ms (cyan)

50 ms IST series. The results might be an indication of which packet sizes should be used to optimize responsiveness in an LTE network, although it would be premature to make such assumptions based on a demonstration experiment alone. This does, however, show how such visible patters can provide valuable hints as to which areas warrant further study, both for understanding network behavior itself and for optimizing software using the network.

To demonstrate the capabilites of LUNA's analysis system, a three-dimensional plot based on the same data is shown in Figure 8.15. The main advantage of this plot format is that the IST is directly visible in the plot, although it has the disadvantage of not showing the standard deviation. Both plotting styles have their uses, and it is up to the user to choose the optimal style for a certain application.

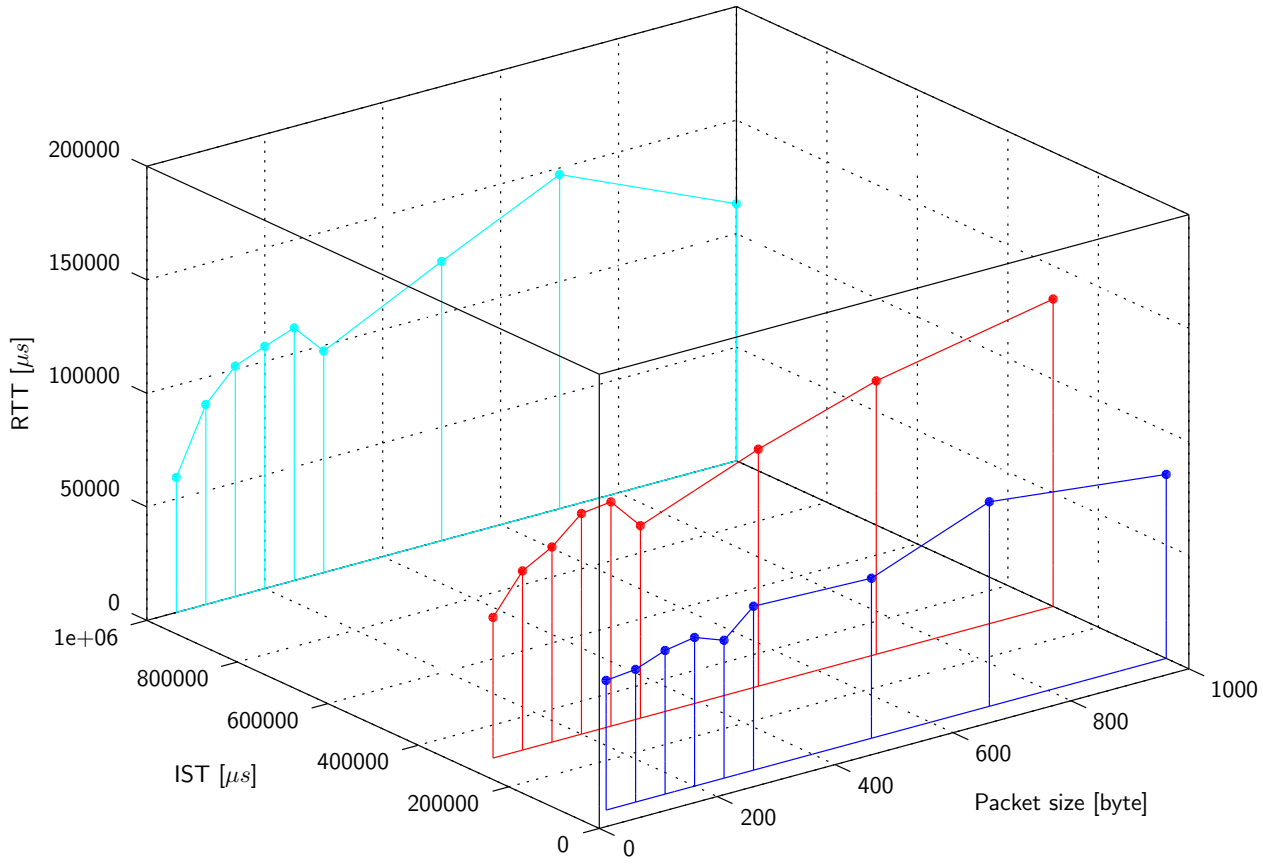


Figure 8.15: Average RTT over LTE as three-dimensional plot over packet size and IST

9 Future Work

This chapter presents potential improvements for future development of LUNA, as well as potential topics for future work.

9.1 Improving LUNA Server Performance

As seen in Section 8.1.2, the server could not handle all arriving packets in a transmission with 6 µs IST, leading to packet loss. A promising way to improve server performance would be to split receiving and processing packets into multiple threads. Such multithreading would require hardware that can actually run them in parallel to provide a benefit, but almost all modern server, desktops, and laptop computers come with multicore CPUs. On systems with only one CPU, like most embedded systems, a carefully designed multithread application would still perform reasonably well, so the sustainable packet rate would be similar to the current situation.

In this context exploring alternative output formats might be useful. A binary format could save both disk space and calculation time for creating human readable representations of the recorded data. This would also require adapting the analysis tools, or writing new ones. Either way, alternative output formats should remain optional.

9.2 Optionally Randomize Padding Contents

When padding is required, LUNA currently uses non-zero, but constant padding bytes. From a performance point of view constant padding is ideal, because it does not require any action to maintain while sending. However, packet contents can affect some aspects of network behavior like power consumption or channel compression, the latter of which is used by some VPNs. Providing an option to create random padding might be useful in such cases, but it should not be a default setting for performance reasons.

9.3 Plugin System for Packet Parameter Generation

The current state of the packet parameter generation framework supports generator develop with generic many generic functions, so developers can focus on the actual generation, but generators still have to

be compiled into the LUNA binary. A plugin system for generators would allow them to be written separately and loaded at runtime, making generator development more convenient.

New generators implementing additional distribution functions would likely be useful as well. Integration with network simulators to let the simulator define packet sizes and intervals would be another possibility.

9.4 More Flexible Server Replies

The echo flag introduced in Chapter 6 instructs the server to mirror the packet as it was received. However, many practical uses of communication networks create asymmetric data flows. For example, a relatively small HTTP request could lead to the transmission of a large file, often requiring many near-MTU size packets, while TCP acknowledgment packets from both sides are fairly small.

Expanding the LUNA server to allow configuring non-symmetric response patterns may be useful to simulate such behavior.

9.5 Using LUNA to generate raw Ethernet packets

Researchers at CNI have expressed interest in doing packet generation on the Ethernet layer, that is, generating raw Ethernet packets instead of UDP. Of course it is possible to create Ethernet packets with a certain payload size by adjusting the UDP payload to that size minus the size of UDP and IP headers as done in Section 8.2, but convenience aside, direct generation on the Ethernet layer could offer some advantages for such experiments:

- Removing the necessity to configure IP addresses at all, particularly in an environment where they are not commonly used
- Avoiding overhead from ARP (Address Resolution Protocol, see [55]) traffic
- No processing overhead from IP and UDP stacks in the operating system

Similarly, generating raw IP packets would allow replicating setups like the one at CERN described by Barczyk et al. [53]. Other protocols like TCP or SCTP could be added as well if needed. However, adding stream based protocols would require significant changes to the fundamental socket management in LUNA.

9.6 Systematic Study of Ethernet Hardware

In Section 8.2.2 and Figure 8.8 in particular it became apparent that different hardware implementations of the same kind of network infrastructure (Ethernet controllers in that case) can have quite different transmission characteristics. A systematic study of the behavior of different Ethernet hardware might be interesting, and could provide practical advice to researchers and engineers who are trying to build measurement and transmission systems with precise characteristics.

Another important topic for such a study would be how hardware drivers influence transmission characteristics, and how far they could be optimized for certain applications. Such insights could be very useful for network driver development in general.

Similar studies could be done for other networking technologies.

9.6.1 Effects of Busy Polling

As noted in Section 4.3.1, optional *busy polling* has been introduced to the Linux drivers for certain network controllers[1] during the development cycle for Linux 3.11 [38]. While busy polling may severely hurt overall system performance, the developers behind the busy polling patches hope to reduce receive latencies for applications which can accept the performance impact. A systematic study of the effects on measurement accuracy could be useful.

[1] See Miller [57] for details.

Bibliography

[1] S.S. Kolahi, S. Narayan, D.D.T. Nguyen, and Y. Sunarto. Performance monitoring of various network traffic generators. In *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*, pages 501–506, 2011. doi: 10.1109/UKSIM.2011.102.

[2] A. Botta, A. Dainotti, and A. Pescape. Do you trust your software-based traffic generator? *Communications Magazine, IEEE*, 48(9):158–165, 2010. ISSN 0163-6804. doi: 10.1109/MCOM.2010.5560600.

[3] M. Paredes-Farrera, M. Fleury, and M. Ghanbari. Precision and accuracy of network traffic generators for packet-by-packet traffic analysis. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006. 2nd International Conference on*, pages 6 pp.–37, 2006. doi: 10.1109/TRIDNT.2006.1649124.

[4] A. Zimmermann, A. Hannemann, and T. Kosse. Flowgrind - a new performance measurement tool. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6, 2010. doi: 10.1109/GLOCOM.2010.5684167.

[5] G.R. Ash and B.D. Huang. An analytical model for adaptive routing networks. *Communications, IEEE Transactions on*, 41(11):1748–1759, 1993. ISSN 0090-6778. doi: 10.1109/26.241755.

[6] Quanyou Feng, Jiannong Cao, Yue Qian, and Wenhua Dou. An analytical approach to modeling and evaluation of optical chip-scale network using stochastic network calculus. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1039–1046, 2012. doi: 10.1109/HPCC.2012.152.

[7] M. Moradian and F. Ashtiani. Analytical modelling of a cognitive ieee 802.11 wireless local area network overlaid on a cellular network. *Communications, IET*, 6(15):2455–2464, 2012. ISSN 1751-8628. doi: 10.1049/iet-com.2011.0305.

[8] Sebastian Subik, Dennis Kaulbars, Patrick-Benjamin Bök, and Christian Wietfeld. Dynamic Link Classification based on Neuronal Networks for QoS enabled Access to Limited Ressources. In *Proc. of the 22nd International Conference on Computer Communication Networks (ICCCN) 3rd International Workshop on Context-aware QoS Provisioning and Management for Emerging Networks, Applications and Services (ContextQoS)*. IEEE, July 2013.

[9] J. Fabini, L. Wallentin, and P. Reichl. The importance of being really random: Methodological aspects of ip-layer 2g and 3g network delay assessment. In *Communications, 2009. ICC '09. IEEE International Conference on*, pages 1–6, 2009. doi: 10.1109/ICC.2009.5199514.

[10] H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. Cited according to Marwedel [11].

[11] Peter Marwedel. *Embedded System Design*. Springer Science+Business Media B.V., 2011. ISBN 978-94-007-0256-1. URL http://www.ub.tu-dortmund.de/katalog/titel/1308341.

[12] Thomas Gleixner and Douglas Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In *Proceedings of the Linux Symposium*, volume 1, pages 333–346. Linux Symposium, July 2006. URL http://www.linuxsymposium.org/2006/linuxsymposium_procv1.pdf.

[13] Jonathan Corbet. Clockevents and dyntick. *LWN.net*, February 2007. URL http://lwn.net/Articles/223185/.

[14] Arnold Robbins. *Linux Programming by Example: The Fundamentals*. Prentice Hall, April 2004. ISBN 978-0-13-142964-2. URL http://www.informit.com/articles/article.aspx?p=173438. Chapter 3.1.

[15] Keith Owens. Linux kernel documentation, kernel stacks, 2010. URL https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/x86/x86_64/kernel-stacks?id=7974891db234467eaf1fec613ec0129cb4ac2332. Last checked: 2013-09-02.

[16] Julian Onions. RFC 1606: A Historical Perspective On The Usage Of IP Version 9, April 1994. URL http://tools.ietf.org/html/rfc1606. Last checked: 2013-10-06.

[17] Jon Postel. RFC 768: User Datagram Protocol, 1980. URL http://tools.ietf.org/html/rfc768. Last checked: 2013-08-20.

[18] Michael Kerrisk et al. Linux man-pages project, udp(7), 2013. URL http://man7.org/linux/man-pages/man7/udp.7.html. Last checked: 2013-09-20.

[19] Michael Kerrisk et al. Linux man-pages project, getaddrinfo(3), 2013. URL http://man7.org/linux/man-pages/man3/getaddrinfo.3.html. Last checked: 2013-10-05.

[20] Michael Kerrisk et al. Linux man-pages project, nanosleep(2), 2009. URL http://man7.org/linux/man-pages/man2/nanosleep.2.html. Last checked: 2013-05-27.

[21] Michael Kerrisk et al. Linux man-pages project, clock_nanosleep(2), 2013. URL http://man7.org/linux/man-pages/man2/clock_nanosleep.2.html. Last checked: 2013-09-27.

[22] Michael Kerrisk et al. Linux man-pages project, ioctl(2), 2000. URL http://man7.org/linux/man-pages/man2/ioctl.2.html. Last checked: 2013-05-27.

[23] Michael Kerrisk et al. Linux man-pages project, socket(7), 2013. URL http://man7.org/linux/man-pages/man7/socket.7.html. Last checked: 2013-05-27.

[24] Michael Kerrisk et al. Linux man-pages project, ipv6(7), 2012. URL http://man7.org/linux/man-pages/man7/ipv6.7.html. Last checked: 2013-05-27.

[25] Michael Kerrisk et al. Linux man-pages project, gettimeofday(2), 2012. URL http://man7.org/linux/man-pages/man2/gettimeofday.2.html. Last checked: 2013-05-27.

[26] Jonathan Corbet. Approaches to realtime Linux. *LWN.net*, October 2004. URL http://lwn.net/Articles/106010/.

[27] Paul McKenney. A realtime preemption overview. *LWN.net*, August 2005. URL http://lwn.net/Articles/146861/.

[28] Luotao Fu and Robert Schwebel. RT PREEMPT HOWTO, 2013. URL https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO. Last checked: 2013-07-05.

[29] Remy Böhmer et. al. HOWTO: Build an RT-application, 2013. URL https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application. Last checked: 2013-07-05.

[30] Michael Kerrisk et al. Linux man-pages project, mlockall(2), 2011. URL http://man7.org/linux/man-pages/man2/mlockall.2.html. Last checked: 2013-07-08.

[31] Michael Kerrisk et al. Linux man-pages project, sysconf(3), 2013. URL http://man7.org/linux/man-pages/man3/sysconf.3.html. Last checked: 2013-07-08.

[32] Michael Kerrisk et al. Linux man-pages project, getrusage(2), 2010. URL http://man7.org/linux/man-pages/man2/getrusage.2.html. Last checked: 2013-07-08.

[33] Michael Kerrisk et al. Linux man-pages project, localtime(3), 2013. URL http://man7.org/linux/man-pages/man3/localtime.3.html. Last checked: 2013-07-08.

[34] Michael Kerrisk et al. Linux man-pages project, sched_setscheduler(2), 2013. URL http://man7.org/linux/man-pages/man2/sched_setscheduler.2.html. Last checked: 2013-07-16.

[35] Michael Kerrisk et al. Linux man-pages project, getpriority(2), 2013. URL http://man7.org/linux/man-pages/man2/getpriority.2.html. Last checked: 2013-07-16.

[36] Michael Kerrisk et al. Linux man-pages project, sched_rr_get_interval(2), 2013. URL http://man7.org/linux/man-pages/man2/sched_rr_get_interval.2.html. Last checked: 2013-10-07.

[37] Michael Kerrisk et al. Linux man-pages project, capabilities(7), 2013. URL http://man7.org/linux/man-pages/man7/capabilities.7.html. Last checked: 2013-07-16.

[38] Jonathan Corbet. Ethernet polling and patch-pulling latency. *LWN.net*, July 2013. URL http://lwn.net/Articles/558305/. Related main line kernel commit is available at https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=0a4db187a999c4a715bf56b8ab6c4705b524e4bb.

[39] Michael Kerrisk et al. Linux man-pages project, clock_gettime(3), 2013. URL http://man7.org/linux/man-pages/man3/clock_gettime.3.html. Last checked: 2013-07-31.

[40] Linus Torvalds. Re: Licensing and the library version of git, 2006. URL http://lwn.net/Articles/193245/. Last checked: 2013-09-06.

[41] Michael Kerrisk et al. Linux man-pages project, pthreads(7), 2010. URL http://man7.org/linux/man-pages/man7/pthreads.7.html. Last checked: 2013-08-02.

[42] Michael Kerrisk et al. Linux man-pages project, sem_overview(7), 2012. URL http://man7.org/linux/man-pages/man7/sem_overview.7.html. Last checked: 2013-08-07.

[43] Michael Kerrisk et al. Linux man-pages project, sem_wait(3), 2012. URL http://man7.org/linux/man-pages/man3/sem_wait.3.html. Last checked: 2013-08-07.

[44] Xavier Leroy. Linuxthreads documentation, pthread_mutex_init(3), 2009. URL http://www.eglibc.org/cgi-bin/viewvc.cgi/trunk/linuxthreads/linuxthreads/man/pthread_mutex_init.man?view=markup&pathrev=9218. Last checked: 2013-08-08.

[45] Michael Kerrisk et al. Linux man-pages project, pthread_cancel(3), 2008. URL http://man7.org/linux/man-pages/man3/pthread_cancel.3.html. Last checked: 2013-08-08.

[46] Michael Kerrisk et al. Linux man-pages project, pthread_join(3), 2008. URL http://man7.org/linux/man-pages/man3/pthread_join.3.html. Last checked: 2013-08-08.

[47] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual — Third Edition (v1.12)*. Network Theory Limited, January 2009. ISBN 978-0-9546120-7-8. URL http://www.network-theory.co.uk/gsl/manual/. Full text available online at http://www.network-theory.co.uk/docs/gslref/.

[48] Michael Kerrisk et al. Linux man-pages project, pthread_cleanup_push(3), 2008. URL http://man7.org/linux/man-pages/man3/pthread_cleanup_push.3.html. Last checked: 2013-08-19.

[49] Jon Postel. RFC 792: Internet Control Message Protocol, 1981. URL http://tools.ietf.org/html/rfc792. Last checked: 2013-08-20.

[50] Tatu Ylonen. RFC 4251: The Secure Shell (SSH) Protocol Architecture, 2006. URL http://tools.ietf.org/html/rfc4251. Last checked: 2013-09-17.

[51] Michael Kerrisk et al. Linux man-pages project, signal(7), 2013. URL http://man7.org/linux/man-pages/man7/signal.7.html. Last checked: 2013-09-18.

[52] Ieee standard for ethernet - section 1. *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)*, pages 1–0, 2012. doi: 10.1109/IEEESTD.2012.6419735.

[53] A. Barczyk, D. Bortolotti, A. Carbone, J-P Dufey, D. Galli, B. Gaidioz, D. Gregori, B. Jost, U. Marconi, N. Neufeld, G. Peco, and V. Vagnoni. High rate packets transmission on ethernet lan using commodity hardware. *Nuclear Science, IEEE Transactions on*, 53(3):810–816, 2006. ISSN 0018-9499. doi: 10.1109/TNS.2006.874546.

[54] RFC 791: Internet Protocol, 1981. URL http://tools.ietf.org/html/rfc791. Last checked: 2013-10-06.

[55] David C. Plummer. RFC 826: An Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware, 1982. URL http://tools.ietf.org/html/rfc826. Last checked: 2013-10-05.

[56] IEC TC57. IEC 61850: Communication networks and systems for power utility automation, 2011.

[57] David Miller. [GIT] Networking, July 2013. URL http://lwn.net/Articles/558310/. Last checked: 2013-10-06.

A Changelog

This appendix lists changes made to this published version of the thesis relative to the version submitted for grading. The printed version additionally included a legal statement on due academic process, which had to be physically signed and would thus be meaningless in online publication.

- Figure 6.4: The label of the box above the sending loop has been changed from "Start sending" to "Prepare to send" for clarity (2013-10-28).
- Section 8.2, second paragraph: Host C was listed as both a merging unit and a control device. The second instance has been corrected to Host D (2013-11-19).
- Section 7.2.1, last paragraph: The word "variable" was missing and has been added (2014-02-09).