BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FAST GENERATOR OF NETWORK FLOWS

RYCHLÝ GENERÁTOR SÍŤOVÝCH TOKŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAKUB BUDISKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ MATOUŠEK**
VEDOUCÍ PRÁCE

BRNO 2016

Brno University of Technology - Faculty of Information Technology

Department of Computer Systems Academic year 2015/2016

Master Thesis Specification

For: **Budiský Jakub, Bc.**
Branch of study: Computer and Embedded Systems
Title: **Fast Generator of Network Flows**
Category: Networking

Instructions for project work:

1. Become acquainted with selected protocols of the TCP/IP protocol suite (IPv4, IPv6, TCP, UDP, ICMP).
2. Study related work in the area of network flow generation.
3. Design a software tool for generation of realistic network traffic. Optimize the design from a performance perspective.
4. Implement the tool and examine its functional and performace characteristics.
5. Compare the characteristics of the tool with related work.
6. Identify performance bottlenecks of the tool and discuss possible ways how to improve its performance.

Basic references:

- According to instructions of the supervisor.

Requirements for the semestral defense:

- Items 1 to 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Matoušek Jiří, Ing.**, DCSY FIT BUT
Consultant: Puš Viktor, Ing., Ph.D., CESNET
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2
L.S.

Zdeněk Kotásek
Associate Professor and Head of Department

Abstract

This master's thesis analyzes existing solutions for generating network traffic designed for testing network components. It focuses on IP network flows and aims towards a flow generator capable of producing flow-based synthetic traffic with speeds up to several tens of gigabits per second. A tool with such purpose, called FLOR, is designed and implemented, taking a stochastic approach to flow planning. It is evaluated and compared to the related work afterwards. Several performance improvements are proposed.

Abstrakt

Tato diplomová práce se věnuje analýze existujících řešení pro generování síťového provozu určeného k testování síťových komponent. Zaměřuje se na generátory na úrovni IP síťových toků a pokrývá návrh a implementaci generátoru, zvaného FLOR, schopného vytvářet syntetický síťový provoz rychlostí až několik desítek gigabitů za sekundu. K plánování toků využívá náhodného procesu. Vytvořená aplikace je otestována a porovnána s existujícími nástroji. V závěru jsou navrženy další vylepšení a optimalizace.

Keywords

FLOR, generator, network flow, TCP/IP, stochastic planning, C++

Klíčová slova

FLOR, generátor, síťový tok, TCP/IP, stochastické plánování, C++

Reference

BUDISKÝ, Jakub. *Fast Generator of Network Flows*. Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Matoušek Jiří.

Fast Generator of Network Flows

Declaration

I declare that this master's thesis is my own work which was written under the supervision of Ing. Jiří Matoušek. I also certify that all the contributions of others in this work are properly referenced.

. . . .
Jakub Budiský
May 23, 2016

Acknowledgements

I wish to thank my supervisor Ing. Jiří Matoušek for his guidance and helpful notes. I also share the credit of my work with my consultant Ing. Puš Viktor, Ph.D., for additional support and help. This thesis would not be possible without their commitment.

© Jakub Budiský, 2016.
This thesis was created as a school work at the Brno University of Technology, Faculty of Information Technology. The thesis is protected by copyright law and its use without author's explicit consent is illegal, except for cases defined by law.

Contents

1 Introduction **3**

2 TCP/IP Protocol Suite **4**
2.1 Basic TCP/IP Protocols 4
2.1.1 IPv4 5
2.1.2 IPv6 5
2.1.3 TCP 6
2.1.4 UDP 7
2.1.5 ICMP and ICMPv6 8
2.2 Flow in the IP Network 8
2.3 IP Flow Generator 9

3 Related Work **11**
3.1 Types of the IP Network Traffic Generators 11
3.2 Overview of the Freely Available Traffic Generators 12

4 IP Flow Generator Design **14**
4.1 Packet Assembler 15
4.2 Packet Scheduler 16
4.3 Flow Scheduler 17
4.4 Communication Channels 17
4.5 User Interface 18
4.6 Proposed Architecture 18

5 Implementation **20**
5.1 Feature Overview 20
5.2 Communication Queues 21
5.3 Thread Mapping 21
5.4 Core Affinity 23
5.4.1 Non Uniform Memory Access 23
5.5 Time Measurements 23
5.6 Pseudo-random Number Engine 24
5.7 Stochastic Flow Planning 25
5.7.1 Interference of Multiple Flows 25
5.7.2 Inserting New Flows 25
5.8 Packet Transmission 27
5.9 User Interface 28
5.9.1 Sessions 28

5.9.2 Graphical User Interface 28
5.9.3 Command Line Interface 29
5.10 Resulting Software Architecture 29

6 Testing **31**
6.1 Testing Environment 31
6.2 Timing Tests 31
6.2.1 Flow Interference 31
6.2.2 Flow Insertion 32
6.3 Performance Tests 34
6.3.1 Flow Count 34
6.3.2 Multi-core Scalability 35
6.3.3 Payload Length 36
6.3.4 Payload Length with Checksum Offloading 37
6.3.5 Flow Insertion 39
6.3.6 Meta-flow Insertion 39
6.3.7 Slow Packet Scheduler 40
6.4 Comparison with Related Work 40
6.5 Performance Bottlenecks 41

7 Conclusions **42**

Bibliography **43**

Appendices **45**
List of Appendices 46

A Distribution Tree **47**
A.1 Tree Traversal 47
A.2 Node Insertion 48
A.3 Node Removal 49

B Build System and Dependencies **51**

C Graphical User Interface **52**

D Deterministic Flow Planning **54**

Chapter 1

Introduction

Communication plays an important role in everyday life. This also applies in the context of information systems, which are growing and getting more complicated every day. Usefulness of almost every information system is dependent on the network and its ability to transfer information across the world quickly and reliably.

The biggest (and still expanding) computer network known today is called the Internet. Its growth property creates a need for developing new, faster and more reliable devices used to maintain communication channels between its users (hosts or applications). It is also making us conscious about the security and tools we need to protect those users.

Devices and applications developed for these purposes need to be thoroughly tested to make sure they will work properly when used in a production environment. One possibility of testing those devices and applications is using traffic generators, which, more or less, try to simulate the real-life network.

This project addresses some of the issues of testing found when developing new technologies for the faster Internet. As none of the related work provides necessary features, it does so by designing and implementing a modular, fast and resource friendly traffic generator working on the IP flow level, the *FLOR*.

The thesis is organized as follows. To cover the requirements of the designed tool, overview of the basic protocols used to communicate over the Internet network is provided in Chapter 2. In Chapter 3 existing generators are reviewed and in Chapter 4 numerous design choices are made to overcome all the encountered problems. Based on those design choices, the architecture of the new generator is proposed.

Chapter 5 summarizes all the implemented features and brings out some of the implementation details. It also shows the software architecture and how it changed compared to the proposal. Tests and their results can be found in Chapter 6, conclusions in Chapter 7.

Chapter 2

TCP/IP Protocol Suite

Also referred as Internet Protocol Suite, the TCP/IP suite is a network model and a collection of protocols originally developed by Defense Advanced Research Project Agency. It provides end-to-end connectivity for the Internet users by specifying how the data exchanged should be processed, transmitted, addressed, routed and received. This functionality is described with help of four abstraction layers. Several protocols have been designed to satisfy the functionality requirements of those layers.

The idea behind the layers used and the exact functionality they implement is out of scope of this project. It is expected that the reader is aware of some basic concepts used in the Internet. Exhausting explanation can be found in many sources, e.g. RFC 1122 [12].

However, for the purpose of this project, it is worth recalling that protocols of the Internet layer, namely IPv4 and IPv6, are responsible for connecting the independent networks and assigning a unique identifier to every device connected in the network. Transport layer protocols like TCP and UDP are used to handle host-to-host communication. Port numbers used in these protocols also refer to applications running on the hosts willing to communicate through the Internet.

Higher level protocols are put into as a payload of the underlaying protocol in a process called encapsulation. Each of the protocols can be distinguished by its header placed at the beginning.

When an application needs to send some data, they are put into a transport layer *segment* and an Internet layer *datagram*. This part of encapsulation is usually done by abstract objects provided by the underlying operating system. The result, also called a packet, is copied over to the *Network Interface Card (NIC)* where it is further encapsulated into link layer and media dependent protocols, producing a *frame*.

In Section 2.1 the essential TCP/IP protocols are presented. Following Section 2.2 discusses IP flows and 2.3 covers their generation.

2.1 Basic TCP/IP Protocols

Below, the basic protocols found in the TCP/IP protocol suite are presented. The chosen protocols are expected to be supported by the generator due to their proportion in the Internet traffic. Even though most of the performance tests are not protocol dependent, more advanced tests and higher level parsers may use the protocol information for their processing. In the future, the list of supported protocols can be expanded to cover more testing scenarios and better approximate the real traffic.

2.1.1 IPv4

Internet Protocol version 4 was originally defined in RFC 791 [16] and its header format can be found in Figure 2.1. Semantic of some of the fields was updated by other standards.

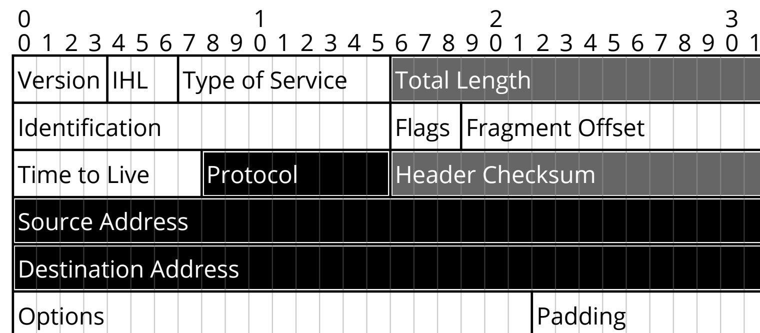

Figure 2.1: IP version 4 header format

The *Version* field in the header must be set to the value '4' and *Internet Header Length (IHL)* contains the length of the whole header in 32-bit long words. *Total Length* is used to determine the sum of header and payload lengths, counted in octets.

Type of Service provides an indication of desired quality of service. Possible values for this field can be found in the referenced protocol specification.

Identification, *Flags* and *Fragment Offset* are designed for the fragmentation feature found in the IPv4 protocol. *Identification* aids the assembly process of the datagram and is set to an arbitrary value, same for all datagram fragments. *Fragment Offset* tells the recipient a position in the datagram and the *Flags* field specifies the fragmenting policy and informs the recipient if more fragment follows.

Time to Live indicates remaining time for the datagram to reach its destination. Datagram must be discarded when the value reaches zero. According to the standard, time is measured in seconds, but every module processing the datagram must decrement its value at least by one. *Time to Live* prevents clogging the network by undeliverable datagrams.

Information about the payload, for which the transport layer protocol is usually used, can be obtained from the *Protocol* field. Defined numeric constants which are used in this field can be found in the online database maintained by IANA[1].

The main purpose of this protocol, addressing clients in the network, is covered by the *Source Address* and *Destination Address* fields. These addresses are used to look up possible ways to the destination in a process called routing which usually takes place on network borders.

Header Checksum is included as a way of detecting transmission errors and must be updated every time the header fields are changed. *Options* contains an optional information provided by the sender and padded to the 32-bit boundary with zeros (the *Padding* field). The available options are included in the standard.

2.1.2 IPv6

Described in RFC 2460 [5] and shown in Figure 2.2, the basic header is even simpler than the one found in IPv4. Some of the functionality was removed (e.g. checksum, which is also

[1] http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

provided by the transport layer) and some was moved into separate extension headers (for example the fragmentation). Few of those headers are defined in the referenced standard, others were added later.

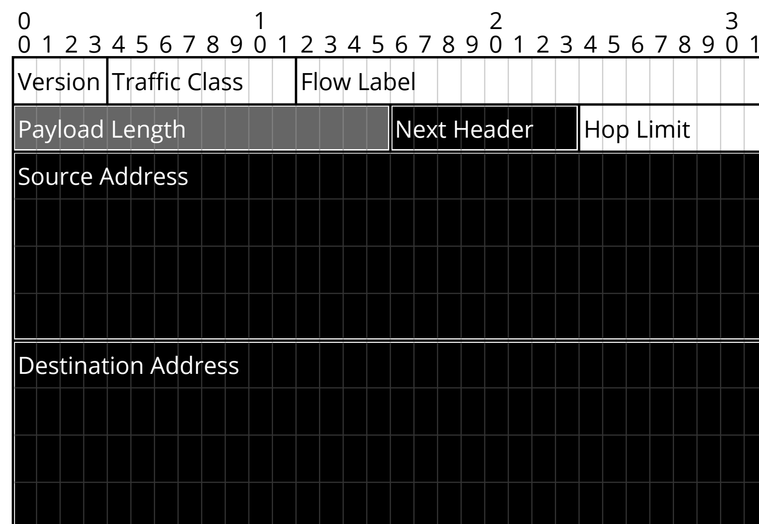

Figure 2.2: IP version 6 header format

The *Version* field contains the value '6' and *Time to Live* found in the version 4 was transformed into *Hop Limit*, no longer containing the value in seconds. *Traffic Class* is intended to provide a functionality similar to *Type of Service* in the previous IP version.

Flow Label is supposed to flag a sequence of packets which should be handled in a specific way. Usage of this field is not strictly defined by the standard. Its value should be set to zero when not used. The *Next Header* field informs about consecutive header. It can reference some of the available IP extensions or the payload, similarly to IPv4. The addresses fulfill the same purpose but were expanded to occupy 128 bits each.

2.1.3 TCP

The definition of Transmission Control Protocol can be found in RFC 793 [15] and its header format in Figure 2.3. The goal of this protocol is to provide connection-oriented, stream-like data transfer while maintaining the integrity and correct ordering. Other functions include flow control and identification of communicating processes.

Starting from the back, the fields used to identify the processes are *Source Port* and *Destination Port*. Some of the values used are reserved for specific services, database of such port numbers can be found on the IANA's website[2].

Flow control is achieved using an algorithm with the help of the *Window* field containing the number of segments sent simultaneously. This number is updated as a reaction to a network state, trying to avoid congestion of the network. Its purpose is to inform the opposite side about the size of receiving buffer for the incoming segments.

The order of segments is well-defined using the *Sequence Number*, which grows with the number of octets sent. The initial value is chosen randomly (for further requirements see the

[2]http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

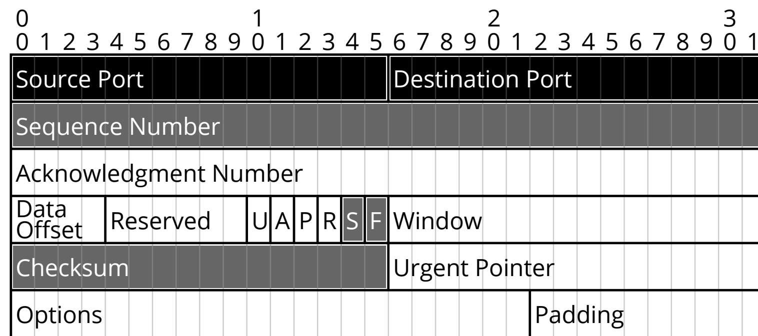

Figure 2.3: TCP header format

standard) and exchanged during the connection initialization, called a 3-way handshake. *Acknowledgment number* is sent back to the sender to inform it about the last data received in order. Its value is derived from the *Sequence Number*.

In order to make *Acknowledgment Number* meaningful, the *ACK (A)* flag must be set. *SYN (S)* flag is involved in the handshake, a segment with this flag set declares the initial *Sequence Number*. Flag *RST (R)* is used to re-initiate the connection after a fault. *FIN (F)* is used to inform the other side that the transmission is over. Communication ends when both sides sent a segment with this flag and the confirmation of receiving it.

Data Offset contains length of the header in 32-bit words, thus informing where the payload starts. Header has the variable length due to *Options* field, but it is always padded to a multiple of 32 bits. *Urgent Pointer* field is interpreted only when the *URG (U)* flag is set. It contains offset to the urgent data found in the segment.

The *Checksum* included to check data integrity and it is calculated from the header, payload, and also covers pseudo-header of the prefixed protocol. Pseudo-header is composed of transport layer protocol identification, length but mainly the addresses to make sure the datagram is being delivered to the correct host. This also means that the calculation is Internet layer protocol dependent.

2.1.4 UDP

Defined in RFC 768 [14], the User Datagram Protocol is much simpler than TCP. Its header is shown in Figure 2.4. Offering only a small overhead it does not provide reliable transfer nor the other advanced features of TCP.

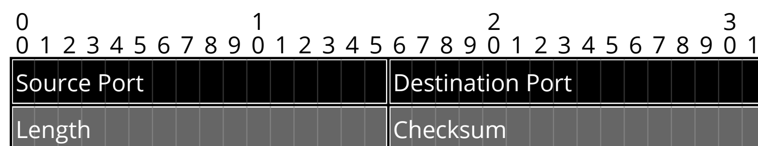

Figure 2.4: UDP header format

Based on the fields found in the header, its function is to identify communicating processes (*Source* and *Destination Port*), provide the data length and check if the data were transmitted

without errors. Similarly to the TCP, *Checksum* calculation includes some of the fields found in the IP header.

2.1.5 ICMP and ICMPv6

The Internet Control Message Protocol [13, 4] is another basic protocol found in the Internet Protocol Suite. It differs from UDP and TCP in a way that its primary goal is not to transfer data. It is used by network devices to inform other devices or query them to get desired information. Typical usage scenarios include destination unreachable messages or similar errors. It is used to get the fragmentation details (the highest possible MTU on a path) or test the routing and device liveness using ping (*ICMP Echo*) messages.

The ICMPv6 is similar to the ICMP for IPv4 networks but its functionality is extended to cover *Neighbor Solicitation* messages (replacement for the standalone ARP protocol) and *Multicast Listener Discovery* (replacement for the IGMP protocol).

A general ICMP header can be found in Figure 2.5. The semantic of the *Type-specific data* is dependent on the actual *Type* used. This 32-bit word can be followed by a payload such as original (trimmed) datagram content within *Destination Unreachable* message.

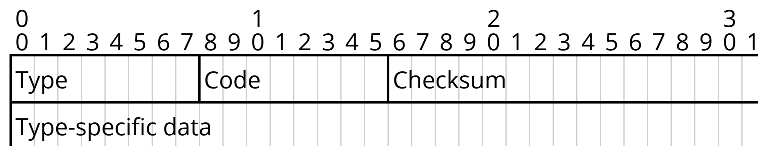

Figure 2.5: ICMP generic header format

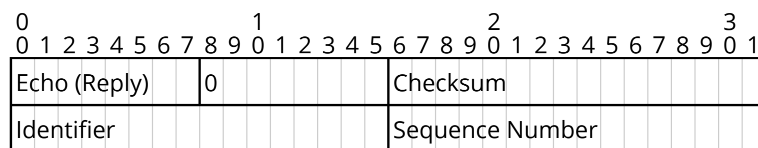

Figure 2.6: ICMP Echo header format

One of the simplest valid ICMP message possible is *ICMP Echo Request* without any payload (Figure 2.6). The whole content can be static as *Identifier* and *Sequence Number* are allowed to be zero.

2.2 Flow in the IP Network

A flow is generally defined as a set of packets passing an observation point in the network during a defined time interval. Apart of these conditions, packets belonging to the particular flow usually share some other common properties, such as header field values or characteristics not explicitly contained in the packet [11].

A flow *record* is then specified as a set of flow identifiers complemented by aggregated information, such as beginning and end times, number of packets and bytes transferred, etc.

This definition is too vague for our case. A common specialization, that is also used by the targeted systems, is the *transport layer* (*L4*) flow. Packets of such flow must share these common criteria:

- Network interface (observation point)
- IP protocol version
- Source IP address
- Destination IP address
- Transport layer protocol
- Source port (if present)
- Destination port (if present)

The time interval limitation still applies, but it cannot be explicitly read from the headers and it is not strictly specified. In flow-measuring applications it is usually implemented as a set of configurable timers.

A flow defined like this can be intuitively interpreted as a set of packets traversing from a specific application to another one. Time restrictions are splitting individual sessions, although this works only approximately. As an example you can imagine yourself checking several simple web pages on a daily basis, using a regular web browser. Each page load will generate two flows, one of which is request and the other response. For each web page a dedicated connection is opened, thus producing different flows even if the web page is located on the same server. Sessions are separated thanks to the time constrains. Even if the same ports are reused on the following day, the existing flow has already expired (thus new flow record is created).

2.3 IP Flow Generator

From the generator point of view, it is clear that if we want to generate the packets of a particular flow, we need to craft all the packets with the same values in the specified 7-tuple. All of them, except the network interface, are present in the Internet and transport layer headers. This is not a problem because the interface used by the monitoring application is well-defined and all generated traffic has to be directed at this particular observation point. As long as we provide a network device capable of joining different streams, a distributed generation is possible.

Generator cannot influence the flow timers present on the tested device, thus observed flows can still differ from the flows created by the generator. If undesirable, this can be avoided by a proper configuration of both generator and tested device.

With all the presented protocols we are able to generate the following set of packet types:

- $\{\mathrm{IPv4}, \mathrm{IPv6}\} \times \{\mathrm{UDP}, \mathrm{TCP}\}$ and a payload
- $\mathrm{IPv4}, \mathrm{ICMP}$ and an optional payload
- $\mathrm{IPv6}, \mathrm{ICMPv6}$ and an optional payload

Most of the traffic consists of UDP and TCP packets, ICMP was chosen due to its specific flow properties. It does not contain source nor destination port and the flow usually consists of one or several packets with long inter-packet time.

Since the statistical flow properties is all that matters, there is no need to be able to generate every possible ICMP message type. Similarly, there is no need to support all the (optional) features of other protocols. Protocol fields at Figures 2.1, 2.2, 2.3, 2.4 and 2.6 are colour-coded as follows:

- Fields with a white background are either static or not present (*Options* and *Padding* in the IPv4 and TCP protocols). They can be the same for all generated traffic.
- Fields with a black background differs among flows as they contain flow identifying values. They need to be specified by the user or generated procedurally. The *SYN (S)* and *FIN (F)* TCP flags also belong to this group because the former should be set for the first packet of the flow while the latter in the last packet containing a payload.
- Fields with a gray background can differ per packet. They need to be specified by the user or can be generated using a stochastic process. Checksums are also included in this category as they need to be recalculated every time a packet is send.

One of the simplest possible flow generator is assumed. More protocols could be added or some of the features can be extended at the cost of its speed, which is our primary concern at the moment.

Chapter 3

Related Work

In this chapter we will present multiple approaches used to generate synthetic traffic and some relevant examples available under a free license.

Few considerable proprietary solutions exist but, due to their information policy, it is impossible to objectively evaluate or compare those products without being a customer of the respective companies.

Although we are aware of the fact that some properties, such as precise packet timing, are required for reproducibility of some of the test cases, choosing a packet generator is generally a trade-off between precise timing, speed and flexibility. Our main focus is on the last two, with the ability to generate flows in mind. It is also difficult to control packet timing from the pure software generator, some more or less limited attempts exist [7].

3.1 Types of the IP Network Traffic Generators

As they are used in many different situations, numerous types of generators exist. They differ in their ability to simulate several properties of the network, provide meaningful application payload or simulate real-life attacks.

Open-loop generators, also referred as *Packet-Level Traffic Generators* [8], use an approach where packets are injected to the network so that some level of similarity with a real link traffic can be observed. These generators can either replay a previously captured traffic (not necessarily with the original payload) or use packets crafted from scratch with the goal to match some statistical properties of the real network.

This type of generators is ideal for experiments where the network has no influence on the traffic, like collector testing or profiling, evaluating caching policies of various network devices or some attacks simulations, such as SYN flood attack. They do not interact with the network in any way.

Opposite case are the *closed-loop* generators. Their main advantage is that they provide a feedback to the network state, which allows them to simulate congestion control mechanisms, delays, measure jitter and other metrics. Their goal is to model the *sources* of traffic (web server, SIP client, peer node. . .). Payload can be meaningful and stateful which make them ideal for an application level testing. Resulting traffic is more realistic for most of the scenarios, although these generators are much more resource intensive.

Generators can occupy a *single node* in the network or can be *distributed*, which can be true especially for closed-loop generators. Many of the closed-loop generators are implemented using the server-client architecture.

Another way to separate the generators is according to their purpose which influences the level of abstraction available when setting up the experiment. Some of them allows the user to set all the headers in various protocols the packet is composed of. Other generators, especially the application specific ones, require a user to specify only the application parameters and some general settings, for example the number of sessions.

3.2 Overview of the Freely Available Traffic Generators

In this section we provide examples of existing generators and discuss their ability to generate flows suitable for our testing. Only a few examples are listed; most of the arguments used apply to the other well-known available alternatives.

MGEN[1]

MGEN stands for Multiple-Generator, open-source multi-platform tool capable of generating a traffic using a scripting interface. Within the scripts the user is able to specify multiple flows along with their beginning which allows its use as a basic flow generator. Disadvantage of this method is a lack of flexibility caused by the need to generate traffic on a per-flow basis.

Ostinato[2]

Ostinato is open-source, cross-platform packet generator. It supports a wide range of protocols together with graphical and scripting user interfaces. By definition of *streams* the user is able to generate a flow or even multiple flows. This can be achieved using per-packet header manipulation feature, which lets the user to set a range of values.

It is impossible to easily model the flow creation and deletion; having the same drawback as *MGEN*. Per packet header manipulation allows the creation of multiple flows at once, but all of them begin and end roughly at the same time or are non-deterministic (with the random option).

D-ITG[3]

A different approach was taken when designing the *Distributed Internet Traffic Generator, D-ITG*. As the name suggests, the architecture of this open-source multi-platform tool is distributed and spans across multiple nodes in the test environment. It is an example of closed-loop generator focused on investigation of models suitable for heterogeneous networks and their scaling.

Generator has the ability to specify multiple pseudo-random distributions for inter departure times and packet sizes. With the aim to simulate the network along with interactivity and measurements, achieved throughput around 250 Mbps at packet rate of 30 kpps does not satisfy our needs [6].

Tcpreplay[4]

Tcpreplay is a set of open-source utilities designed to edit and replay previously captured traffic. It is an obvious solution for generating flows with distribution seen on the real links, although using it for this purpose has several drawbacks.

First of all, we need to acquire or capture the traffic in prior to the replay. This can be a difficult task due to privacy concerns and publicly available anonymized traffic may not be suitable for the chosen test.

Another problem we have ran into is the flexibility of this solution. If we want to saturate a high-speed link without replying the traffic in a short loop, we need several hundreds of gigabytes of the captured information. Pulling data from a storage device at required speed to a NIC can also be quite challenging.

Harpoon[5]

A tool with desired functionality, flow generation. *Harpoon* is a tool designed to extract information about flows from *Netflow* traces to generate traffic imitating statistical characteristics of the network.

Although usage of closed-loop client-server architecture allows for more realistic traffic, it is also a main disadvantage of this tool. Apart from limiting the throughput, multiple machines are necessary to run the tests. It was designed to test active network devices, such as routers, with all the advantages of closed-loop generator [17].

Swing[6]

Another generator with closed-loop architecture is called *Swing*. It is an open-source project with focus on capturing packet interactions in observed traffic. It promises to generate statistically similar traffic which correctly reproduces packet-length and flow distributions [20]. User is required to provide a network model which can be then populated with packet traces from captured traffic or given by the user.

As with *Harpoon*, this tool is focused on the realism more than throughput and we cannot expect it to perform better than an open-loop generator.

MoonGen[7]

MoonGen [7] can be considered as a state of the art open-loop generator of today. Despite being focused on precise packet departure timing for reproducible testing, it is much more flexible because of its ability to run per-packet Lua scripts. This can be faster in many use cases by limiting the code execution to operations needed by the specific scenario. It would also allow us to implement our flow generator, but Lua does not provide efficient data model for this purpose. We also do not expect divergent code in the packet sending loop which could be eliminated by this approach.

[1] http://www.nrl.navy.mil/itd/ncs/products/mgen
[2] http://ostinato.org
[3] http://traffic.comics.unina.it/software/ITG/
[4] http://tcpreplay.appneta.com/
[5] http://cs.colgate.edu/~jsommers/harpoon/
[6] http://cseweb.ucsd.edu/~kvishwanath/Swing/
[7] https://github.com/emmericp/MoonGen

Chapter 4

IP Flow Generator Design

Our goal is to simulate the flows and achieve the highest traffic bandwidth possible. As we do not need any reactions to the network state and tested devices are monitoring tools (no interaction is required), open-loop generator is a perfect choice.

Although a single node architecture is preferred over distributed solutions (due to latency an synchronization problems), some degree of modularity must be achieved for making performance optimization possible. As different parts of the system can limit the performance in diverse situations, modular design allows us a better resources management on the node. We can allocate more of the processing power for parts of the system that need it. With carefully though-out problem decomposition it is possible to create a flexible, scalable and extensible tool.

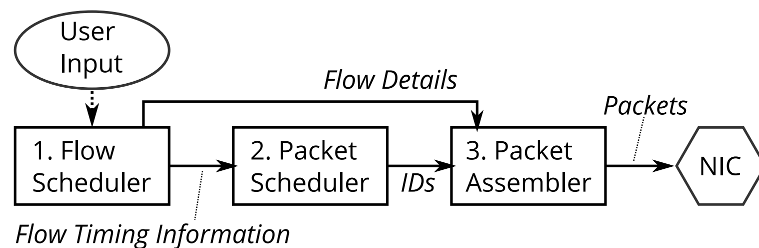

Figure 4.1: Basic decomposition of the problem

The problem can be divided into three main parts (Figure 4.1). The first unit, *Flow Scheduler*, takes care of flow scheduling, thus deciding when to start planned flows. The *Packet Scheduler* is responsible for scheduling the packets from different flows in a way that respects flow speeds (which can differ among the flows in general). Packets are assembled and transmitted in the third unit.

The first two units need to exchange information about flow length and intensity. Apart from that, the *Flow Scheduler* passes the detailed flow information to the *Packet Assembler* directly as it is not needed for deciding when to send out a packet. *Packet Scheduler* just tells the *Packet Assembler* which packet it should transmit.

If a bottleneck is found, the unit can be optimized or with an addition of input multiplexing and output demultiplexing whole unit can be duplicated (Figure 4.2). It also allows us to consider even more advanced optimization such as hardware-software co-design by offloading the work of one or more units into the application specific hardware.

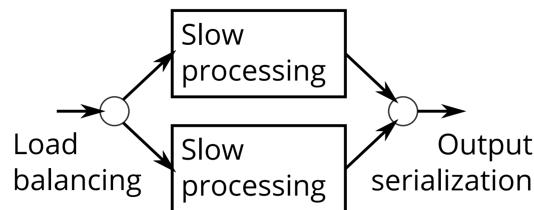

Figure 4.2: Dealing with bottlenecks by unit duplication

Another important feature of this basic design is that there is no feedback. That means the processing can be pipelined and each of the units can be executed in parallel, on its own.

With this basic idea in mind we need to examine the functionality of the proposed units in the target environment and find a solution for the encountered problems.

4.1 Packet Assembler

The main work of this unit consists of data copying. It is done multiple times and not only in the user space. System-provided raw sockets, used for packet injection, give us no guarantee on the latency, nor the performance. Various OS services can influence the available throughput and delay, from interrupt management to task scheduling. Since we do not really need any of the features provided by the Linux kernel, we can rely on a widely used optimization - kernel network stack bypassing. Using this approach we can also eliminate copying packet to the kernel space. This technique is also known as *zero-copy*. Few projects on this topic exists, Netmap[1] for example.

There is also a part of the job that requires quite a big amount of computational power. Since we are crafting the packets ourselves and their content can vary, each time the packet is transmitted we need to recalculate checksums. Checksums can be found in almost all used protocols except IPv6. UDP is not required to contain a checksum if it is encapsulated in the IPv4 protocol.

To address this issue, in most of the situations it would be possible to partly calculate the checksum and save the value for later. Drawback of this solution is limited extensibility of the *Packet Assembler* unit.

Data Plane Development Kit[2] is a project that could possibly overcome both problems. It supports checksum offloading and bypasses the kernel networking stack with its own kernel module. The downside is limited hardware support, but with other free options it is not any better.

CESNET has its own zero-copy library, *Sze2* [1]. Support is limited to the COMBO Ethernet cards. A big advantage is the availability of a NIC capable of transmitting 100 Gbps, but available firmwares do not support checksum offloading on the Internet and Transport layer.

[1] http://info.iet.unipi.it/~luigi/netmap/
[2] http://dpdk.org/

4.2 Packet Scheduler

The algorithm this unit is supposed to execute is illustrated in Figure 4.3. In every time interval, based on the current speed (expressed in packets per second), this unit has to pick a flow from the active flows and signal the *Packet Assembler* that the packet belonging to this flow should be transmitted. If the sent packet was the last one, flow has to be removed from the set of active flows.

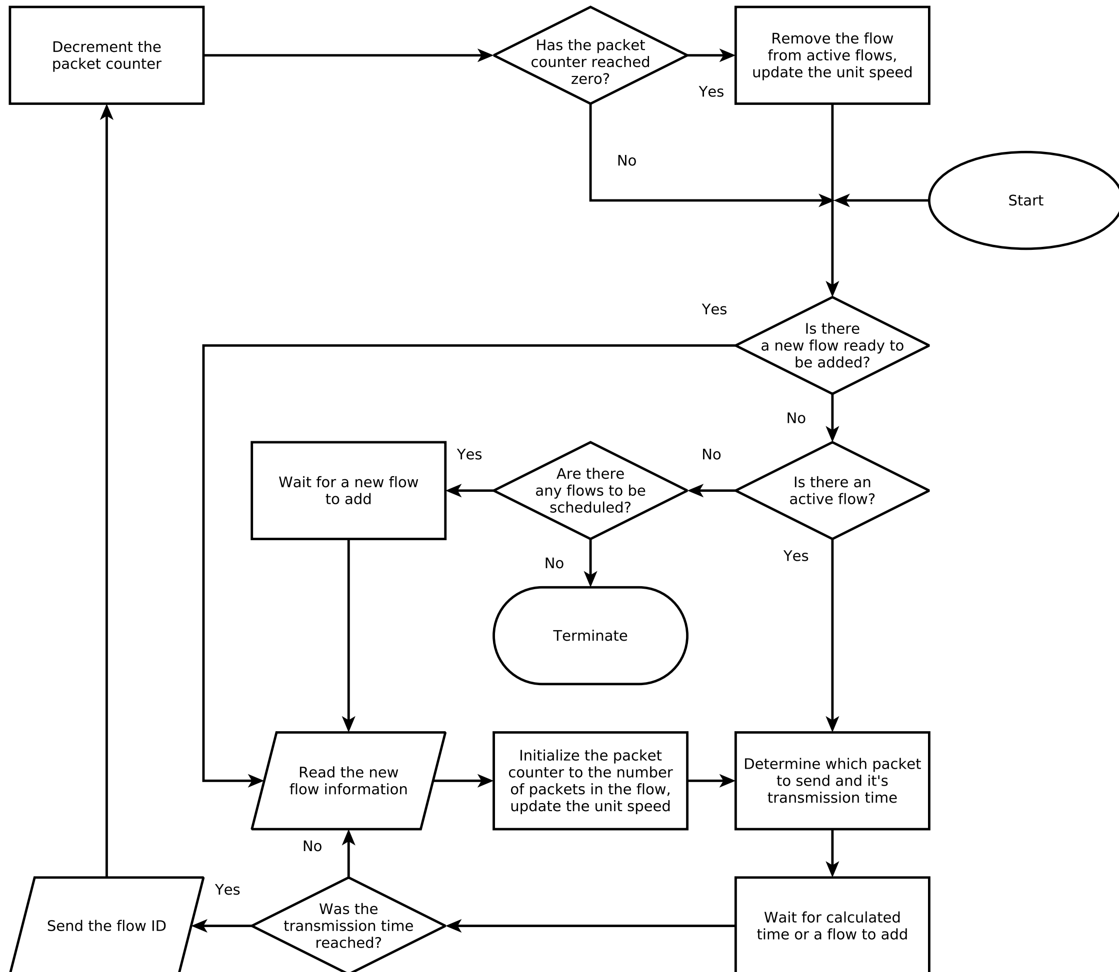

Figure 4.3: Algorithm executed by the *Packet Scheduler*

If we take into account modern hardware used today, supported link speed starts at 10 Gbps. To saturate such a link using packets with average length of 1000 bytes, the *Packet Scheduler* needs to make a decision every 800 nanoseconds. Considering processor's clock running at 2.6 GHz, the decision must be made in 2080 processor cycles, which is not many. In addition, Ethernet supports several times shorter packets and much faster hardware, with rates up to 100 Gbps, is available.

Amount of active flows can be in an order of hundreds of thousands. Even worse, the number of active flows is dynamic and each flow has its own speed. Although some sort of a simulation calendar could be used for this purpose, that would require keeping a sorted structure of planned packets. We decided to try a different approach and use a stochastic flow planning.

Task of picking a flow of which the packet should be sent can be transformed to a stochastic process, where the probability of the flow being selected is proportional to its speed. As an outcome, we need to be able to generate numbers (IDs from an active flow set) with a general discrete distribution. We need either to generate such distribution directly or pick an algorithm generating random numbers with uniform distribution and transform it.

Several approaches for transforming uniform distribution into a general discrete distribution exists. Some of them offer value transformation with constant time complexity but require non-trivial preprocessing with time complexity of $O(n)$ or worse, where n is a number of discrete values to generate [21]. This fact makes them less favourable for the *Packet Scheduler*. A simple data structure called *Distribution Tree* was made with $O(\log_2 n)$ complexity for inserting, removing and transforming a value. Details about this data structure can be found in Appendix A. Although the lookup and modification complexity is still the same as for a simulation calendar based on a heap, our approach will provide us more flexibility and ability to trade-off some of the generation aspects.

Because of performance the *Distribution Tree* is using integer values to express probability. This creates a new problem, representing flows that are slower than 1 packet per second. We can think of the probability as a fixed point number instead of the integer, but we can always come up with a flow slower than a resolution of such number. Moreover, the maximum speed would be limited instead. To get around this complication we can make a standalone unit for managing the slow flows. It can be based on a discrete simulator and use a simulation calendar to pick the packets to transmit.

Flows with varying transmission rate can be approximated using multiple flows whose rate is constant. This would introduce an increased load due to the higher number of flows to schedule. We are not aware of any monitoring tool capable of capturing such flow metrics, thus this feature is not planned yet.

4.3 Flow Scheduler

As the performance of the *Flow Scheduler* depends on the amount of flows dispatched per time interval, it does not have to be optimized as heavily as the *Packet Scheduler*. The number of flows created per second is not expected to be much more than a few thousands. Implementation of this unit can be done as a simple discrete simulator with an event calendar.

The only addition to a basic calendar is the possibility to plan a new flow in the future. This allows the user to create many similar flows at once in a declarative way. It is also possible to expand the features with some model providing more realistic generation, for example *ON/OFF source* model [18].

4.4 Communication Channels

All of the presented units need a fast communication channels to pass information between each other. In most cases this information is short - for example an ID of the flow or its speed. Since there is always only a single producer and a single consumer, the obvious way to exchange the information quickly seems to be a lock-less FIFO queue.

In a case of unit duplication a multiplexer and demultiplexer can be implemented in a separate thread, distributing or gathering the items in round-robin manner. It is also possible to add some sort of heuristic if it will not lead to race conditions, for example picking the output based

on the load of assigned unit. This will allow us to keep the one producer/consumer condition and will avoid the usage of synchronization primitives.

4.5 User Interface

Last but not least, an important problem to solve is the user interface. It should provide the user a possibility to create real-like traffic in a declarative way. Even better, it should provide an option to generate traffic based on already observed one. Hence, some sort of flow record information import should be possible.

Performance-wise, all of the given information can be processed in advance and should not cause any troubles. The only limitation is the available system memory for the resulting simulation calendar.

4.6 Proposed Architecture

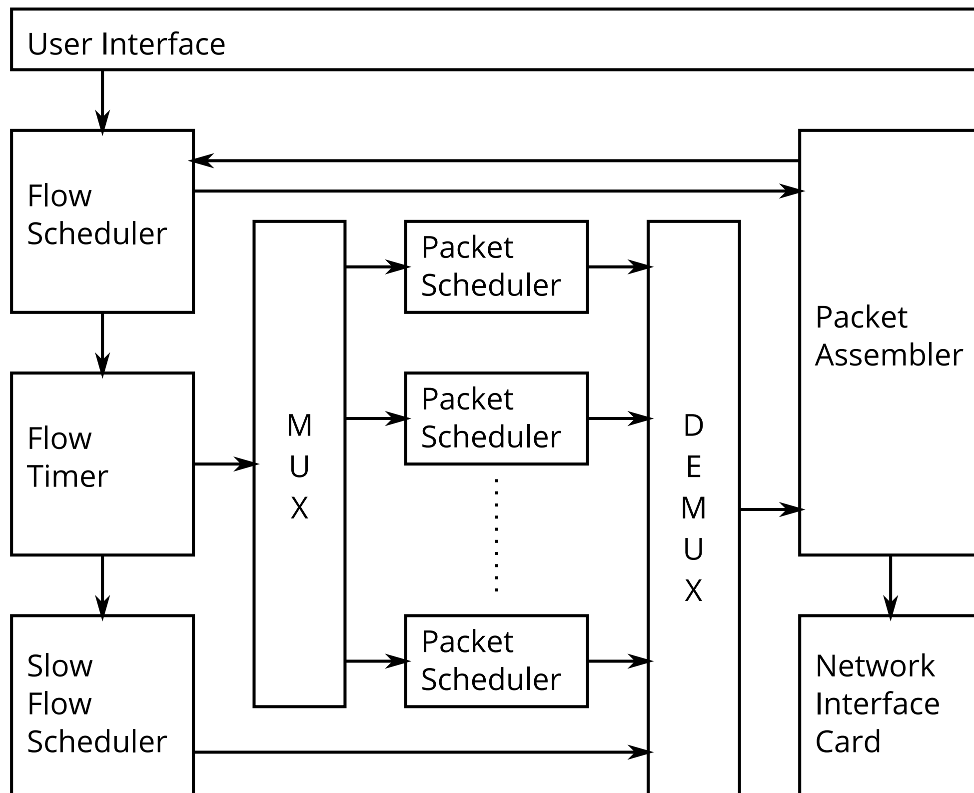

Figure 4.4: Resulting architecture of the designed tool

The initial idea presented in this chapter (Figure 4.1) was extended to the final proposal found in Figure 4.4. It reflects all the points discussed in previous sections and a few more thoughts.

Packet Assembler provides the *Flow Scheduler* a list of available flow IDs. This way they can be simply used to address a flow record memory in the *Packet Assembler* with low ID management

overhead. The drawback of this solution is a feedback loop in the design it introduces. Given a sufficient number of IDs and size of the queue big enough to contain them all, this should not be a problem. The *Packet Assembler* would not have to wait on the queue as there will always be a free space for every one of the IDs.

Slow Flow Scheduler was added to address issues with slow flows discussed in Section 4.2. Because there are multiple sources producing IDs for the *Packet Assembler*, demultiplexer was added into the design. Its presence can be also used for the unit duplication discussed earlier, as the *Packet Scheduler* is expected to be a bottleneck.

Another unit was introduced, the *Flow Timer*. Whole purpose of the unit is to dispatch flows in the right time. It also allows some sort of preprocessing, for example filling up the *Packet Assembler* flow cache with information in advance. The *Flow Scheduler* is allowed to run at full speed with the *Flow Timer* acting like a buffer, eliminating a possible jitter caused by the *Flow Scheduler* processing.

Chapter 5

Implementation

Before we proceed further into a more detailed look at the implementation, Section 5.1 summarizes implemented features along with several name conventions used further on. The performance of a parallel application heavily depends on the used communication technique. Section 5.2 digs into lock-less queues that were proposed for this purpose. Obtained information is used along with other considerations to refine a per-thread work assignment (Section 5.3). Mapping those threads onto CPU cores is discussed in Section 5.4.

Details about a time measurement (Section 5.5) and pseudo-random number generation (Section 5.6) are important for the stochastic flow planning, covered in Section 5.7.

We wanted from FLOR to be usable in most environments, which requires configurable way of packet transmission. We look into that in Section 5.8. Section 5.9 talks about the user interface, and the last section, 5.10, provides an updated overview of the software architecture.

5.1 Feature Overview

The implemented flow generator, also called *FLOR*, is capable of

- Specifying a set of flows and *meta-flows*, a *session*, where:
  - User can select from all of the protocols mentioned in Section 2.3 along with their flow identifying fields
  - User can set payload length, which can be randomized (uniform distribution only)
  - User can provide up to 64 bytes of a custom payload which is placed at the start (compile-time configurable constant)
  - User can set the flow start time with a nanosecond precision
  - User can set flow's intensity with 1 pps (packets per second) or 1 spp (seconds per packet) precision
  - *Meta-flow* is a flow, which can be resent in the future, while allowing the user to
    - Specify a start of the first flow, number of repetitions and repetition period with a nanosecond precision
    - Randomize most of the flow parameters (uniform distribution only), including IP addresses, ports, payload length and intensity
    - Increment a port or an IP address
- Session loading and saving

- Running a simulation of a created or loaded session
- Setting the number of threads executing the *Packet Scheduler* and the *Packet Assembler* (see 5.3)
- Setting a NUMA node to run on (see 5.4.1)
- Setting a core for each worker thread to bind to (see 5.4)

5.2 Communication Queues

In the proposed generator, we need to pass some information between the components working in parallel. This information consists mainly of flow identification numbers; higher amount of data need to be passed only when inserting a new flow into the *Packet Assembler*.

As mentioned earlier, all the communication channels are connecting two parallel blocks (that can be considered threads), a single producer and a single consumer. For such a case it is possible to implement the communication channels using lock-less FIFO queues on the top of a circular buffer. This is a common and technique for data sharing with low overhead.

The main requirement of this approach is that pointers (or indexes) used to mark the start and the end of the data in a queue are atomic. In other words, a thread needs the ability to update those pointers in an atomic way, so that the other thread cannot read a partially changed (thus invalid) value.

Another requirement is that those pointers cannot be stored in the architectural registers as a part of optimization process because the changes need to be visible for the other thread. Both requirements can be fulfilled with the help of the C++ atomic library. On the target architecture, x86-64, this approach has just a little overhead because operations on operands with size up to 64 bits are atomic (with some alignment requirements) [2, 3.9.1][9, 8.1.1], only a compiler is limited in terms of optimizations.

Even though this is probably the fastest way possible, it is not cheap. Performance of the implemented queue was measured using 4 byte wide unsigned integers, chosen for the flow identifiers. Using our testing environment (mentioned in Section 6.1), with two threads doing nothing but polling on the queue we were only able to pass about 20 millions of elements per second between those threads. This is only about 40 % more than the number of the smallest Ethernet packets needed to saturate 10 Gbps link, and no actual work has been done yet.

As a synchronization overhead is related to the number of transactions, it is possible to minimize it by sending more information than just a single number. A pack of numbers can be created and passed all at once. Possibility to send less than a full pack and a periodic flushing (to bound the latency) adds more overhead to the design, but overall, this solution scales well. Another option is to fully avoid communication, especially for such small amount of information.

5.3 Thread Mapping

By looking at the proposed architecture, it is possible to see many blocks working in parallel – all the schedulers plus the assembler, multiplexer and demultiplexer. With just one *Packet Scheduler*, our design requires 7 running threads. Assuming we would need more threads to run the *Packet Scheduler* and maybe even the *Packet Assembler*, this solution does not scale well on a single node as we've already occupied all the available cores on a regular CPU.

Also, considering the queue overhead discussed in the previous section (5.2), it is clear we need to reduce the number of threads significantly. Otherwise, there would be many threads spending most of their time waiting for synchronization. This can be done by giving the threads as much work as possible while keeping the design modular and scalable.

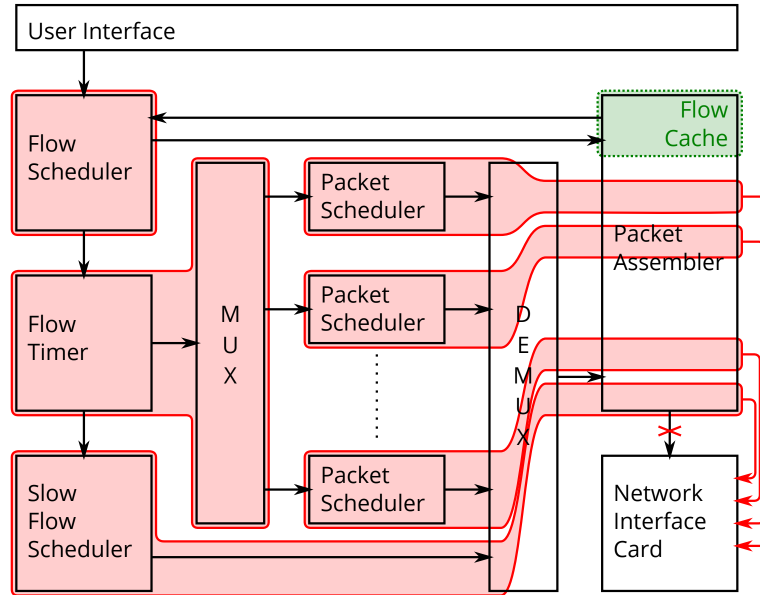

Figure 5.1: Thread mapping to the proposed parallel units

Optimized thread work allocation is illustrated in Figure 5.1. Distinct red areas correspond to the computation done by separate threads. Multiplexing is included into the *Flow Timer* and demultiplexing is eliminated by merging the *Packet Scheduler* and the *Packet Assembler*. This also allowed us to get rid of the communication overhead caused by the queues around demultiplexer. With this approach, only 4 threads are needed to run a basic simulation and a new *Packet Scheduler* and *Packet Assembler* unit is added with each additional thread.

Merging the *Packet Scheduler* and the *Packet Assembler* raises further issues. Firstly, multiple NIC interfaces are required by this design; otherwise we would need to serialize outputs of the worker threads. This should not be a problem as we can exploit the fact, that modern NICs provide multiple, configurable and independent transmit queues, also called *rings*. The number of threads executing the *Packet Scheduler* and the *Packet Assembler* is then limited by the number of available transmit rings for a specific NIC.

The second complication, highlighted with a green area and a dotted outline, is a *Flow Cache*, which is a part of the *Packet Assembler*. It stores all the details about active flows. There are two possible implementation options, a shared or a distributed *Flow Cache*.

It is safe to assume that only one thread will access an entry within the cache at a time (either the *Flow Scheduler*, when filling up the data, or one of the *Packet Assemblers*). Additionally, preliminary measurements did not show any performance advantage of one approach over the other, thus it is easier to use simpler, shared *Flow Cache*.

The only thing left to sort our is a queue containing available (free) flow IDs (mentioned in Section 4.6), which needs to be modified for a concurrent access. This is done in a lock-less manner with use of the *Compare And Swap* (CAS) atomic operation. Downside of this approach is that insertions still need to be completed in order of their start.

5.4 Core Affinity

Core affinity, sometimes called *processor affinity*, is an ability to bind a process (or a thread) to a specified core (or a set of cores). Setting up an affinity is a feature of an operating system's task scheduler.

Binding a process to a core can have a positive performance impact, minimizing task switching and taking advantage of memory locality. When a process is planned to run on a specific CPU only, there is high chance that the memory required by the process is still available in the cache, especially if it is not often preempted by other processes.

There are other considerations as well. Our processes use a polling technique when waiting on communication queues, which could result in undesired effects like spinning, starving or even deadlock (depends on the scheduler settings). This situation is possible if both consumer and producer are limited to run on the single core. The same problem can be encountered with multiple producers or consumers on the same queue, if a concurrent variant is used. This is caused by the fact that multiple reads and writes have to be completed in order.

Although on a system with sufficient amount of cores, dedicated to running a single application, this may not be an issue since the scheduler will generally avoid task switching, this is not always the case. Automatized load balancing is a difficult task and gets more complicated with features like simultaneous multi-threading. We assume the user knows his architecture and the running test better; thus accepting user's input in non-trivial cases is desired.

5.4.1 Non Uniform Memory Access

On a machine containing multiple compute nodes (processors), the problem of affinity settings is even more significant. NICs are typically connected to the system using PCIe bus (including all the NICs supported by DPDK at this time and COMBO cards). PCIe controller is integrated into the processor, which means that only one of the processors can talk to the PCIe device.

Moreover, DRAM controllers are embedded into CPUs too. That means that every processor in such NUMA machine has its local memory modules.

If the application is talking with a PCIe device or memory located on a different node, it has a significant impact on the throughput and operational latency. This is a pure consequence of the fact, that the access is provided by another CPU through processor interconnect links.

To keep the implementation simple, it is left to the user to select a correct NUMA node.

All the affinity-related settings are implemented as a part of the `AffinityManager` class. Apart from letting the user to set a NUMA node, it is possible to set a core for each of the worker threads. The class keeps track of running tasks per core and it chooses the first available node on a chosen NUMA node by default. It also tries to migrate all memory pages used by the application to the memory closest to the selected NUMA node.

5.5 Time Measurements

Timing flows and packets would not be possible without a reference clock. Several options are available. Since we are using STL with C++11, the obvious choice would be `steady_clock` from `std::chrono` library.

The problem is that we cannot be sure how it is implemented. In any case, obtaining the time using the `std::chrono::steady_clock::now()` call can produce a syscall, especially with an older Linux kernel or standard library implementation. This is obviously slow and

dependent on the clock source used by Linux kernel itself. Moreover, the C++ standard does not define a resolution of this clock and it may prove insufficient in the future [10, $30.2.4].

The lowest-overhead high-resolution clock available for users on x86-64 architecture is *Time Stamp Counter (TSC)* [9, 17.15] [3]. It uses a single instruction, `rdtsc`, to read a time stamp, which is represented by a 64-bit tick count since the system start-up.

There is a lot of confusion about using TSC for time measurements. It was broken by introduction of CPU overclocking, power-saving modes and multi-core architecture. While using TSC is definitely not portable, we can safely rely on this feature with a recent hardware. Modern CPUs from all major vendors enhanced TSC unit, making it increment at a constant rate, usually with the processor's base frequency.

Faulty BIOS, responsible for synchronization of the TSC units among processor cores and nodes, can still make the value invalid when migrating a process between cores, but we are trying to avoid this behaviour anyway (see Section 5.4). Dynamic overclocking features manipulating the bus frequency can still make TSC unusable for timing. This feature can be found on high-end gaming motherboards and needs to be disabled first.

Frequency of the TSC can be estimated quite precisely with the help of a reference clock, `std::chrono::strady_clock` in our case. It can be also provided by the user who knows the base frequency of the processor or can obtain that information from a more reliable source.

When using TSC, all time stamps need to be recalculated from seconds to TSC ticks. This may seem like a disadvantage, but by using `std::chrono` this would be done internally every time the current time is requested. This is avoided by using TSC directly, as the conversion occurs only when updating the next event's time stamp.

The only problem with converting values between nanoseconds and TSC ticks is the precision. It requires multiplication of potentially big values and possible overflow must be handled.

5.6 Pseudo-random Number Engine

Selecting a feasible pseudo-random number generator is one of the most important steps in the FLOR's implementation. A new pseudo-random value, or even multiple ones, need to be generated every time

- a new flow is created from the meta-flow description (IP addresses, port, intensity and packet count randomization),
- a flow needs to be selected when scheduling packets,
- a length of the packet is generated.

The last two processes mentioned are part of the packet scheduling and assembly, thus part of the tight sending loop.

Many algorithms for generating pseudo-random values exist; they differ in their quality (measured by several statistical tests), amount of generated data per operation, speed and purpose. Even thought in our use case we do not require cryptographically secure generation and the speed is our biggest concern, quality is still desirable. Since this topic is out of scope of this thesis, existing research was used to pick a suitable one [19].

Picked algorithm, called `SplitMix64`, generates 64-bit wide numbers, so we need to call it twice to randomize an IP address, which may not be optimal. In contrast, the sending loop only needs 32-bit number randomization, which is a potential optimization surface. The chosen algorithm was wrapped in an STL-like class and thanks to the unified interface, it can be easily replaced in the future if needed.

5.7 Stochastic Flow Planning

Stochastic flow planning was chosen as a method to determine which packet should be sent next. Using this method implies some timing considerations that need to be taken care of and are discussed in this section.

To be able to use this method in the first place, we need a proper way to compare flow's intensities. This is due to the fact that a probability of a flow being selected is determined by its intensity. A flow with higher intensity has to be planned more often to keep the timing intact.

The first step is to refine the term *flow intensity*. Generally speaking, it is a speed of the packets belonging to a particular flow. It is thus measured in packets per second and calculated by dividing the observed number of flow packets by the flow duration.

This definition does not preserve inter-packet times, which is crucial when we want to use this information for a flow timing. A flow consisting of two packets and spanning over one second has the same inter-packet time as a flow consisting of three packets that spans two seconds (transmission time is negligible), although their intensities, according to the definition, differ (2 pps versus 1.5 pps).

For this reason, we need to modify how the *flow intensity* is calculated, so that it takes inter-packet times into account. This is done by subtracting 1 from the packet count, as this is a typical case of a *fencepost error*.

$$i = \frac{n-1}{t_{\text{last}} - t_{\text{first}}} \quad \text{for } n > 1 \tag{5.1}$$

Intensity of a flow consisting of a single packet is undefined, thus such flows cannot be planned by the *Packet Scheduler*.

5.7.1 Interference of Multiple Flows

It is clear that producing required results in terms of flow timing highly depends on the quality of a used (pseudo-)random number generator. Despite the fact this generator was carefully selected, we should not be too optimistic and consider possible outcomes of this approach.

Assuming the implementation of the pseudo-random engine (with a desired general discrete distribution) is correct, we would still need a reasonable amount of generated values to observe the distribution specified. Generating deviating short-term sequence can cause two effects – start of the flow may be delayed, or the intensity of the flow can be increased.

We can illustrate those effects on a simple example. Imagine two flows with the identical intensity i and the same beginning time t_{first}. The worst case is that all the packets from one of the flows are sent first, followed by all the packets of the other flow. Intensity of the first flow would be $2i$ in this case (sum of their intensities, as the flow intensity of the system is aggregated). The second flow would have a correct intensity (after the first flow has finished, the system consists of a single flow), but would be delayed by the time it took the first flow to be sent (half of its specified duration, $\frac{t_{\text{last}} - t_{\text{first}}}{2}$).

Since the situation gets more complex when we consider different flows and dynamic flow addition, usability of this approach was proven experimentally. They can be found in Section 6.2.

5.7.2 Inserting New Flows

When inserting new flows to the *Packet Scheduler*, aggregated intensity is influenced and timing needs to be adjusted. The basic idea is illustrated in Figure 5.2. In the simple example shown,

there are two flows, A, represented by blue crosses, and B, represented by red lines. Both flows have the same intensity and their desired duration is highlighted using a coloured background.

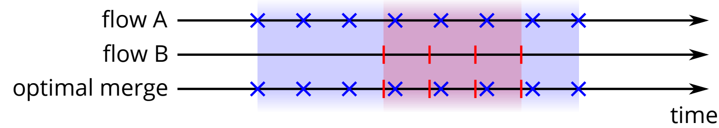

Figure 5.2: Optimal inserted flow timing

To achieve precise timing, as shown in the figure, we would need to store separate timing information for each of the flows. This is basically what a simulation calendar does, but since we are using a stochastic planning, this detailed flow information is lost. Two possible approaches to this problem can be found in Figures 5.3 and 5.4.

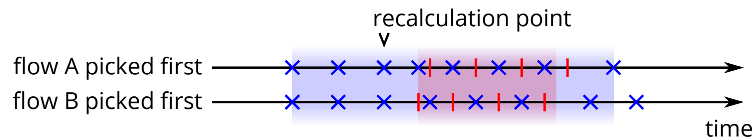

Figure 5.3: Flow inserting "go back"

In the former case, the time stamp of an upcoming send event is calculated using its previous value and new aggregated intensity. This effectively "moves" the start of the *flow B* to the transmission time of the previous packet. This may introduce a positive time slack (when packet is planned to the past) as a side effect.

If we assume a perfect random number engine interleaving the two flows, as both have the same intensity, two scenarios are possible. They differ by the first random choice and can be seen in Figure 5.3. *Recalculation point* is the point in time from which the new timing is calculated. You can see that the first packets are sent out later. As we cannot send them in the past, they are transmitted as soon as the new flow is introduced.

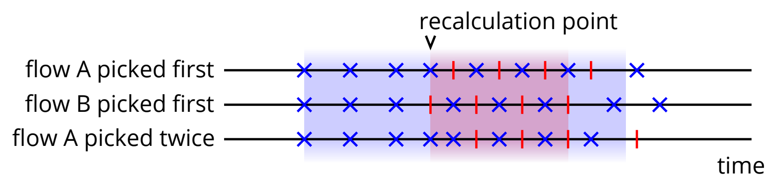

Figure 5.4: "Immediate" flow inserting

Even simpler approach (in Figure 5.4) is to just ignore the interval passed since the previous transmission time, pick a flow and send the corresponding packet immediately. Further packets are planned using the new aggregated intensity. Time of the previous packet is not needed in this case.

While it may seem that the former algorithm is better, this heavily depends on a scenario. With additional flows and different intensities, the problem gets much more complicated. Besides, the error introduced by both insertion algorithms is limited by the maximum flow period,

a second in our case, while the errors created by a bad random choice may introduce more serious timing issues. An example is shown in the third row of Figure 5.4 (the flow A is picked twice before interleaving the choices) and discussed in Subsection 5.7.1.

Taking these facts into consideration, the latter and simpler algorithm was implemented. It will be replaced if it proves unsatisfactory.

5.8 Packet Transmission

A part of the job of the *Packet Assembler* is to send crafted packets through a NIC to the network. This process differs depending on the NIC and used drivers. We discussed some of the packet sending options in Section 4.1. An object responsible for packet transmission will be further referred as a *Backend*.

We don't want to limit FLOR to use just a single packet sending method. It is convenient to make it interchangeable depending on the current needs and also have a possibility to implement a new one if necessary. For this reason, a polymorphic design is desired. Using inheritance along with virtual methods in C++ introduces a level of indirection due to virtual tables usage. Overhead introduced this way is unnecessary because the user is not expected to change the used transmission method at runtime.

C++ provides another way to achieve a common interface, static polymorphism, using template programming. All the objects dealing with the *Backend* can be a template, accepting a type of the *Backend* as a compile-time argument.

Only few common methods are required. Apart from initialization and deinitialization methods, allowing the clean resource management, a method to send a packet and a method to flush an internal buffer are necessary.

The purpose of the flush method is to be able to implement burst packet sending in the *Backend*, while avoiding the need of separate timers and threads in the design. The method is invoked to bound the latency when no packets were sent in a while. Sending packets in bursts minimizes communication overhead and allows us to achieve higher transmission rates.

All of the *Backend* objects are executing a common code – filling up an arbitrary buffer with the packet content. To avoid code duplication, this is implemented as a part of the *Flow Cache*. It takes care of all the protocols starting with Ethernet MAC addresses and it is also able to calculate checksums of higher layer protocols. A checksum offloading feature is supported too; in that case Transport layer checksums are calculated from the pseudo-headers only.

A drawback of the current implementation, which can be eliminated at some performance costs, is that the *Flow Cache* cannot tell the *Backend* the size of a packet in advance. For this reason, the *Flow Cache* expects a buffer that can hold the longest payload the *Flow Cache* will ever write.

Even if the size announcing is implemented, a lot of packet sending methods do not use a continuous storage – for example DPDK's fixed-size, linked list-like *mbufs*[1] or Sze2's cyclic buffer that can require a packet to be split up. This can end up with an unnecessary memory copying, which we are trying to avoid at all costs.

As a result, the *Flow Cache* writes down only a part of the packet. A checksum, if needed, is calculated only from this part. It tells the *Backend* how long the packet should be, but it is *Backend*'s responsibility to zero-initialize the rest of the buffer or fix the checksum.

[1] http://dpdk.org/doc/guides/prog_guide/mbuf_lib.html

Four *Backend* objects were implemented so far:

- **DPDK**, supported NICs can be found on the DPDK website[2]
- **Sze2**, to be used with COMBO NICs
- **Pcap**, for debugging purposes and capture file creation
- **Dummy**, that does not send packets, implemented solely for testing

Even though DPDK implementation exists, it was not thoroughly tested nor profiled due to the lack of compatible hardware. Offloading the checksum computation is not yet supported with this *Backend*.

The Sze2 *Backend* was used in the performance tests. Since there is no firmware that supports checksum offloading at this time, an option to simulate this behaviour was added.

Pcap interface creates a nanosecond libpcap capture file[3] for every thread sending the packets, thus avoiding synchronization. The user can merge created capture files, for example with the *mergecap*[4] tool. Performance of this *Backend* can be still limited by the storage system throughput.

5.9 User Interface

5.9.1 Sessions

As mentioned in the beginning of this chapter, session is the name used for a set of flows and meta-flows that can be simulated using FLOR. As FLOR has the ability of saving and loading sessions, a suitable data format had to be picked for the purpose.

Because of the fact we want to be able to modify sessions by hand and easily process and/or create them by scripts, a text, human-readable JSON format was picked. The session is then represented as an array of objects, each of them describing one flow or meta-flow.

Most of the values describing a meta-flow are self-explanatory. For now, it is possible to obtain the semantic of fields from the available source code, or to reverse-engineer sessions created using GUI.

Although importing of other formats is not implemented for now, conversion from other sources, such as a flow collector dump, should not be difficult.

5.9.2 Graphical User Interface

Although a graphical user interface was created especially for convenient (meta-)flow specification, it supports all the features except custom payload specification. Unfortunately, this feature is only available by manual editing of the session file right now. Several screenshots of the GUI can be found in Appendix C.

---

[2] http://dpdk.org/doc/nics
[3] https://wiki.wireshark.org/Development/LibpcapFileFormat
[4] https://www.wireshark.org/docs/man-pages/mergecap.html

5.9.3 Command Line Interface

Command line interface is simplistic and provides a way of running FLOR using automated scripts. It can only be used to run the simulation by providing an existing session file. Basic statistics are printed to the standard output in a CSV format every second, log is written to the standard error output. The simulation can be ended by signalling the process with SIGINT or SIGTERM or it will end itself after processing all the provided flow entries.

It is possible to provide TSC frequency, set a NUMA node or core binding using application parameters.

5.10 Resulting Software Architecture

The core part of the implemented application can be found in Figure 5.5. To keep it simple and readable, the user interface is not included in the figure.

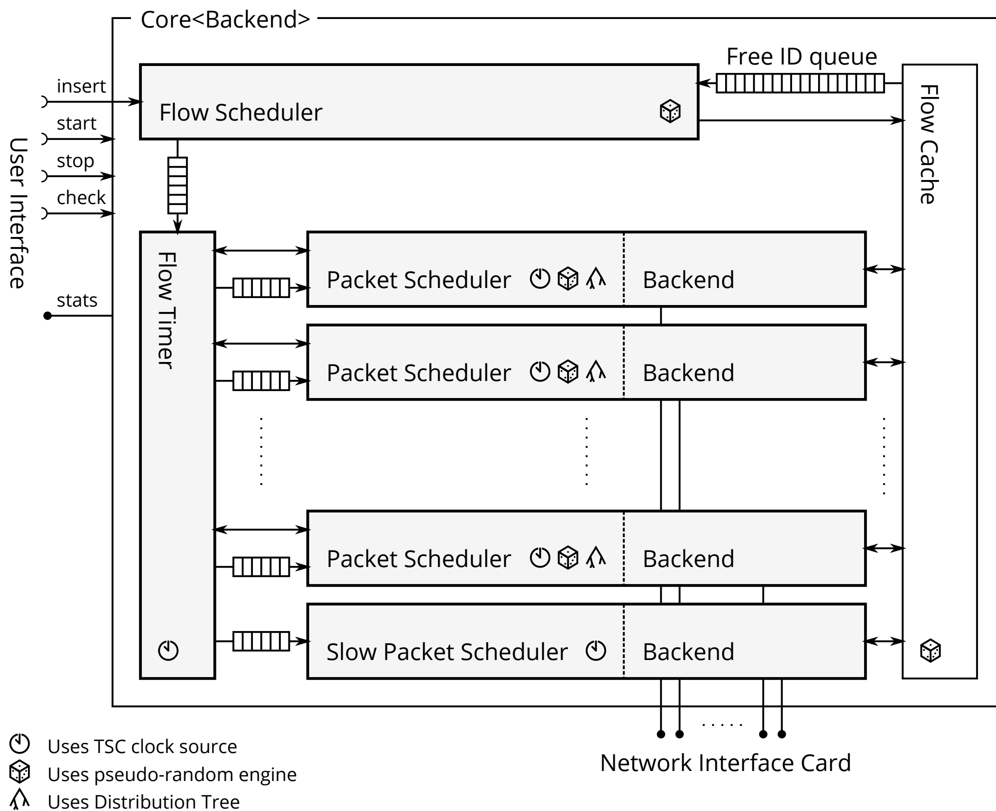

Figure 5.5: Resulting architecture of the implemented tool

The user interface is responsible for creating or reading a session and passing the (meta-)flows one by one to the *Core*. When the calendar inside *Flow Scheduler* is filled up, UI can invoke *start* or *stop* methods to control the simulation. The *Core* itself does not supervise the running threads, UI can periodically call *check* methods to do that. This will ensure that the threads are terminated when there is nothing left to do. UI can also request statistics, which are aggregated per second to avoid performance impact.

The *Core* itself has not changed much from the proposal in Section 4.6. The *Packet Assembler* was left out, packet assembly itself is done by the *Flow Cache* and the created packets are sent using the *Backend* (as mentioned in Section 5.8). All highlighted objects are running their own thread (as discussed in Section 5.3) after the *start* is invoked on the *Core*.

The *Slow Flow Scheduler* was renamed to the *Slow Packet Scheduler*, which is more descriptive, as the object is scheduling packets, not flows.

All the communication queues are illustrated with a buffer in the middle of an arrow connecting objects. The *Free ID queue* is the only queue that supports concurrent access. The *Flow Timer* has an additional connection with each of the *Packet Schedulers* that is used to check their load. This helps to pick the best candidate to plan a newly started flow. The *Slow Packet Scheduler* manages packets of all slow flows and flows composed of a single packet, so there is no need for an extra communication.

The queue between the *Flow Scheduler* and the *Flow Timer* acts as a buffer mentioned in the proposal. It is constantly filled up by the *Flow Scheduler* so that flows are planned in advance. All the other queues, filled up by the *Flow Timer*, are implemented to prevent blocking the timer thread.

Chapter 6

Testing

6.1 Testing Environment

The application was tested using the following hardware:

Motherboard	Supermicro X10DRU-i+
BIOS	SMBIOS 2.8
Processor	2x Intel Xeon CPU E5-2660 v3 @ 2.60 GHz
Memory	4x Micron 8 GB DDR4 @ 2133 MHz per CPU node (64 GB total)
NIC	COMBO-100G with NIC 7.1 firmware

System specification:

Operating system	CentOS Linux release 7.2.1511 (Core)
Kernel	4.5.0-1.el7.centos.x86_64
Compiler	g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-4)
GNU glibc	2.17 release 106.el7_2.4

6.2 Timing Tests

Purpose of these tests is to evaluate an impact of the effects caused by stochastic planning (described in 5.7.1) and flow insertion (discussed in 5.7.2). Two test cases were designed; the first one focusing on the stochastic planning and the second one for both effects.

All timing tests are done using the Pcap *Backend* and a capture file is further processed using a simple flow collector. Aggregated flow data are used to extract flow beginnings and their duration.

6.2.1 Flow Interference

The goal is to generate multiple finite flows in parallel and observe an impact to the flow's timing. Test case is repeated multiple times with different seeds for pseudo-random number generator. The following parameters were chosen for this test:

Number of flows	100 in a single meta-flow
Flow beginning	0 ns (at the beginning)
Flow intensity	10 pps
Number of packets in a flow	100 (flow duration 9.9 s)

The session consists of a single meta-flow, the remaining settings were chosen so that they have a minimum impact on the measured values. As there is no way of the getting precise simulation start from the capture file, time stamp of the first packet in the file is used. This should not introduce a significant error. Results are illustrated in Figure 6.1.

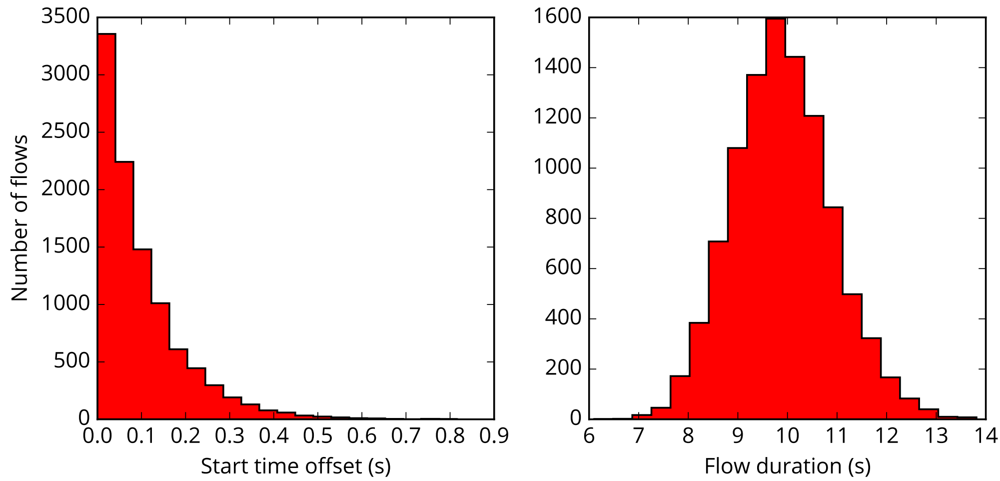

Figure 6.1: Timing accuracy with interfering flows

As expected, the error related to the flow insertion is not higher than a second, although the flow start is influenced by the PRNG sequence too. The flow duration is dependent on the PRNG quality and most of the flows deviates from the correct value only slightly. It can also be concluded that the insertion algorithm has no significant effect on flow duration, both prolonged and shortened flows are included with comparable counts.

6.2.2 Flow Insertion

This test case is designed to generate several flows over a selected time period, achieving constant flow and packet rate.

Number of flows	10000 in a single meta-flow
Flow beginning	0 ns, 0.1 s repetition period (10 flows per second)
Flow intensity	10 pps
Number of packets in a flow	100 (flow duration 9.9 s)

The simulation should stabilize at approximately 100 active flows, yielding 1000 packets per second. A transport layer port increment is used to be able to calculate flow beginning time and its deviation from the resulting capture file. As in the previous test case, time stamp of the first packet is used to determine the simulation beginning.

Results do not differ from the previous test case, as can be seen in Figure 6.2. This means that the dynamic flow insertion does not introduce any additional errors.

For the completeness, this test case was repeated with the higher flow intensity and packet count. Better timing accuracy was expected, as the number of generated numbers is higher which implies better distribution. This hypothesis was confirmed as can be seen in Figure 6.3. The number of flows and flow length were preserved, while the intensity and packet count were increased ten times.

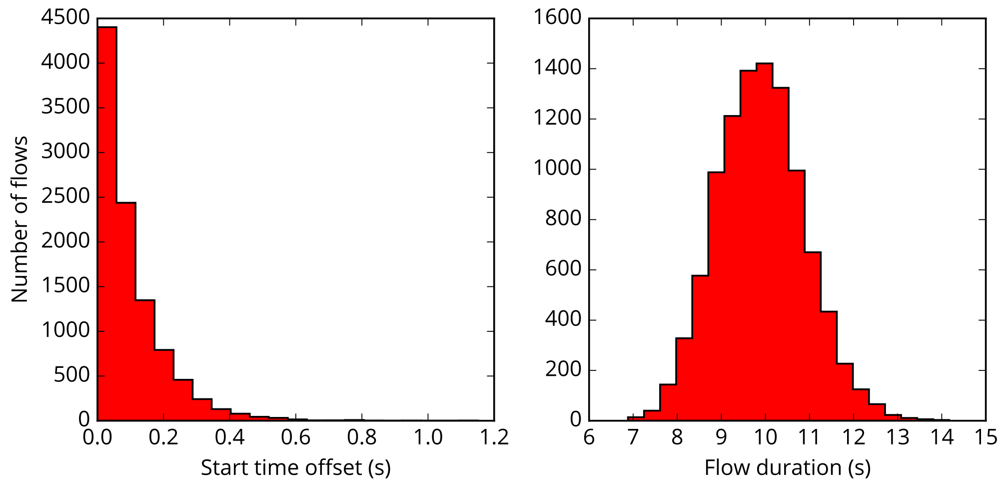

Figure 6.2: Timing accuracy with dynamic flow addition

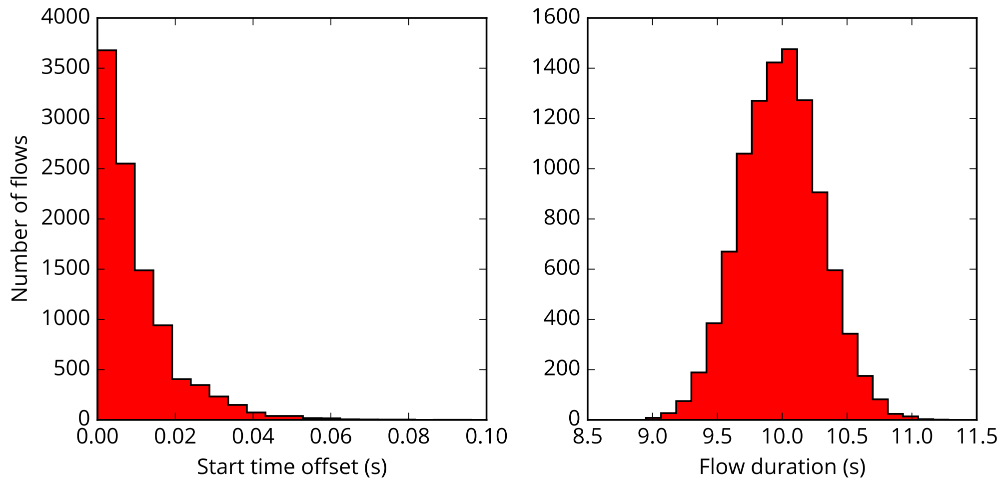

Figure 6.3: Timing accuracy with dynamic flow addition, ten times increased flow intensity

6.3 Performance Tests

The FLOR was designed and implemented to be as fast as possible. In this section we will evaluate several aspects of the generator regarding to the speed of packet and flow generation.

As the generator speed is influenced by several factors at once, test cases are designed so that those factors are isolated as much as possible. Results should give the reader a general idea how the FLOR will perform in other, more complicated use cases.

The Sze2 *Backend* is used in all performance tests and a NUMA node is set according to the NIC's physical location.

6.3.1 Flow Count

In the first test case, the number of generated packets is related to the number of flows. A single *Packet Scheduler* is used.

Number of flows	2^0 to 2^{26}, with exponential increment, single meta-flow
Flow beginning	0 ns
Flow intensity	$\frac{100 \text{ Mpps}}{\text{flow count}}$, keeping it high without the risk of overflow
Number of packets in a flow	Maximum possible value $(2^{32} - 1)$

Short, meaningless Ethernet frames are sent in this case to hide overhead of other actions such as more data copying and checksum calculation. The simulation is terminated after gathering enough data, signalling a running FLOR process. An average packet rate is used for the results.

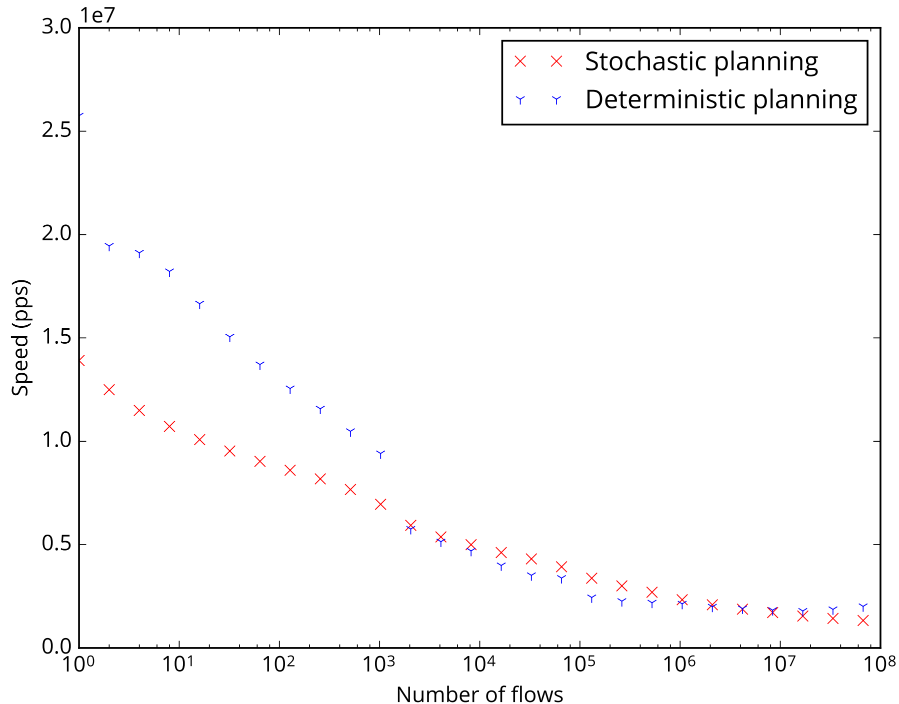

Figure 6.4: Impact of a flow count to packet generation speed

Results are compared to the implementation that uses a heap-based simulation calendar in contrast to stochastic flow planning. An architecture of such generator and feature differences can be found in Appendix D.

In Figure 6.4 it can be seen that the resulting speed decreases significantly with the number of flows planned. This effect is caused by underlying planning structures, both having the time complexity $O(\log n)$ for performed operation, where n is the number of flows. It may seem that deterministic planning performs better, but this is significant only for a small amount of flows.

6.3.2 Multi-core Scalability

With a constant flow count we are changing the number of running *Packet Schedulers* to see the scalability of this solution.

Number of flows	2^{19} in a single meta-flow
Flow beginning	0 ns
Flow intensity	190 pps (roughly 100 Mpps in total)
Number of packets in a flow	Maximum possible value ($2^{32} - 1$)
Number of *Packet Schedulers*	1 to 7

Evaluation is done the same way as in the previous case. Results can be found in Figure 6.5 and they show that FLOR scaling is linear with the number of cores running *Packet Scheduler*.

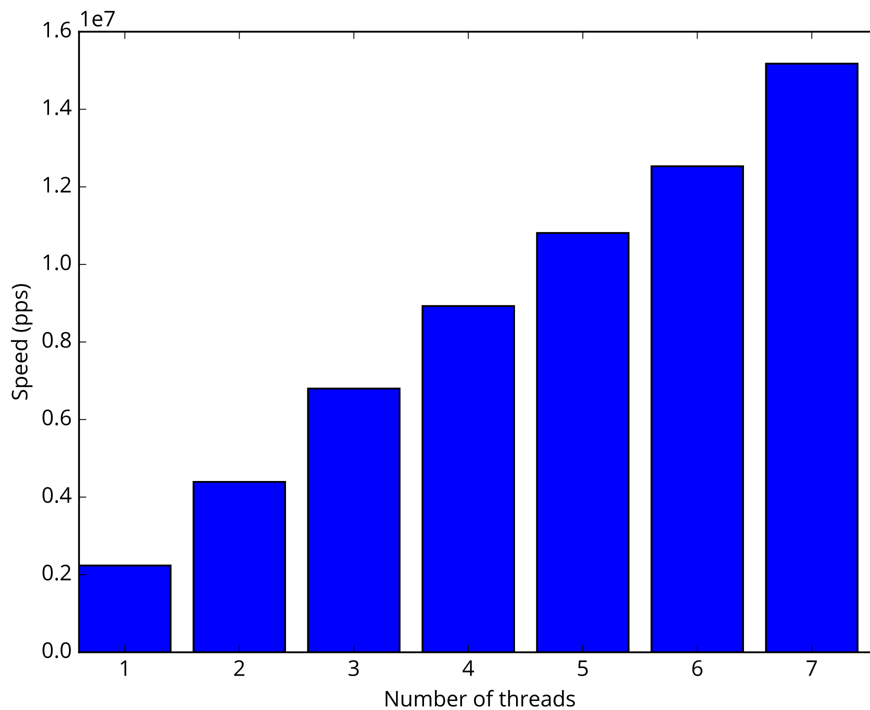

Figure 6.5: Generation speed with multiple cores

6.3.3 Payload Length

Another factor that influences the FLOR's performance is a packet length. In this test case, UDP packets encapsulated into IPv4 protocol are generated. Although speed of the *Packet Scheduler* should not depend on the packet length, this is not the case for *Flow Cache* and *Backend*.

Number of flows	1
Flow beginning	0 ns
Flow intensity	100 Mpps
Number of packets in a flow	Maximum possible value ($2^{32}-1$)
Payload length	0 to 1472 bytes with 64-byte increment

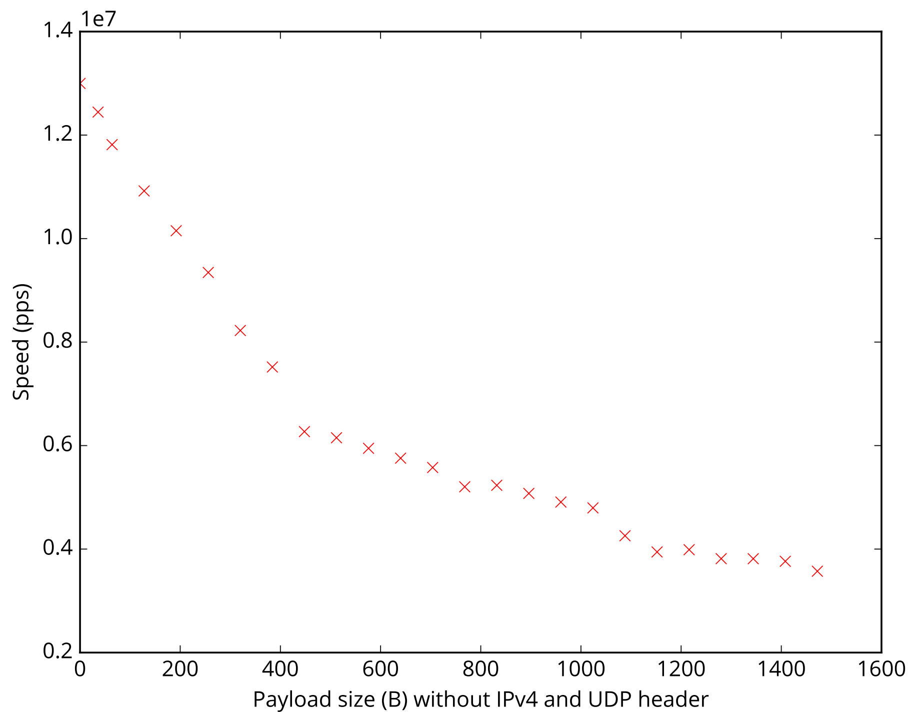

Figure 6.6: Speed depending on a packet length

Figure 6.6 shows that intensity of generated flow is decreasing non-linearly with the packet length. Figure 6.7 compares these results with the maximum packet rates achievable for given lengths at 10 Gbps and 40 Gbps links. This demonstrates two things. The actual link speed (measured in octets per second) is increasing with packet length, and the FLOR can saturate 40 Gbps link with one flow, considering larger frames and single CPU core. It is also able to saturate 10 Gbps link regardless of the packet length.

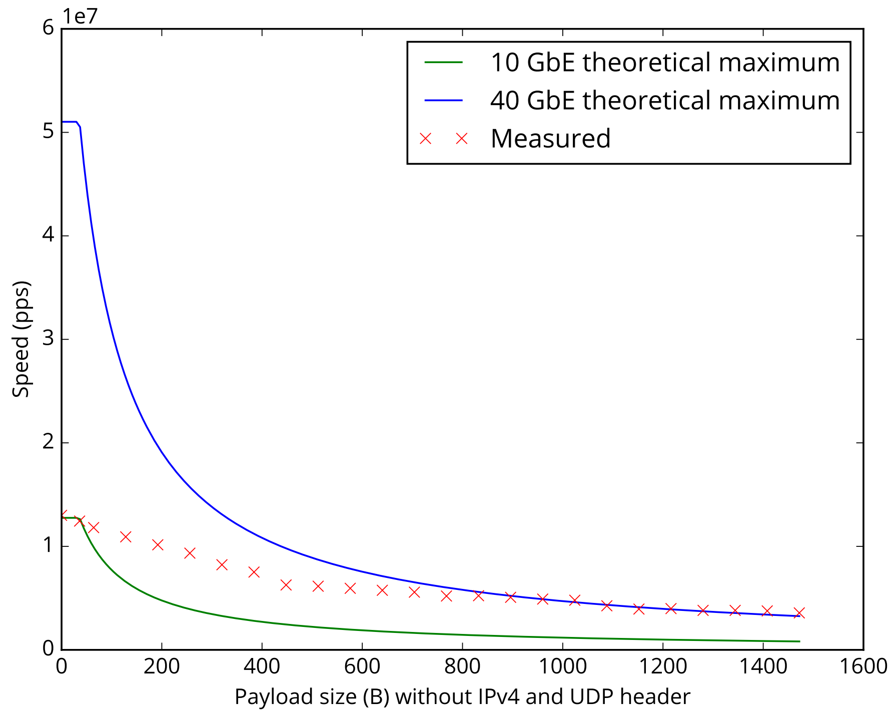

Figure 6.7: Speed compared with theoretical maximums

6.3.4 Payload Length with Checksum Offloading

The generator should perform better if the Internet layer, but mainly the Transport layer checksum offloading is implemented. Even though this is not the case, FLOR was modified to simulate this behaviour.

When the offloading simulation is turned on in the Sze2 *Backend*, *Flow Cache* is instructed to calculate the Transport layer checksum using the pseudo-header only. The Internet layer checksum is not calculated at all and a zero is substituted in the respective field. This is how checksum offloading works on several other NICs today and how it is implemented in the DPDK framework.

Parameters used are the same as in the previous test case and results can be found in Figures 6.8 and 6.9. Apart from the fact that reached speeds are higher than the ones without simulated checksum offloading, some jitter can be seen. Profiler did not reveal causes for this behaviour in FLOR's code and we believe it is caused by the NIC driver and transfer mechanisms, and it is also influenced by the burst sending strategy.

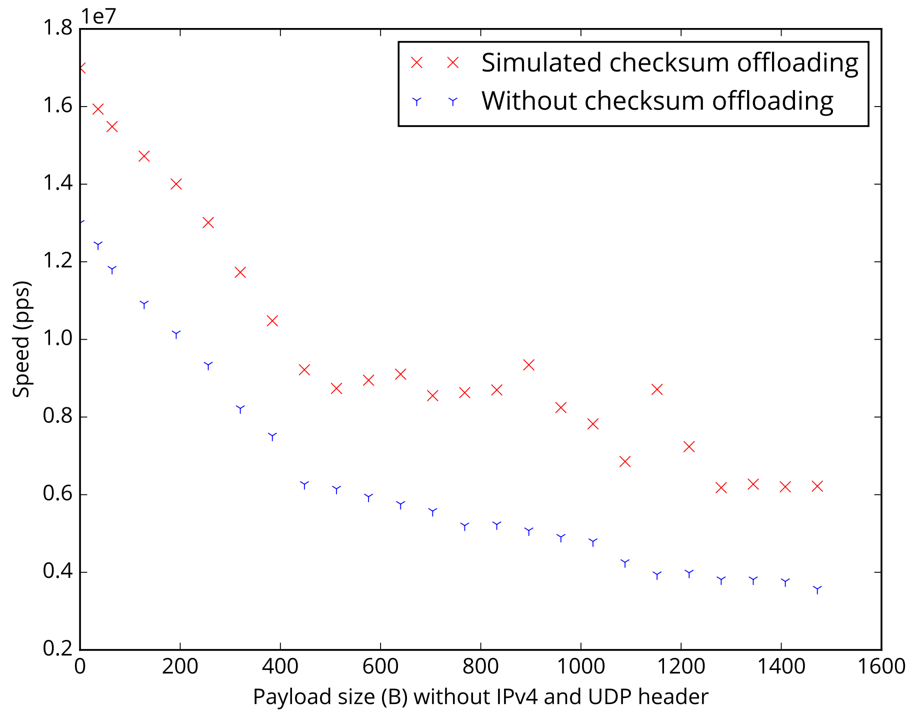

Figure 6.8: Speed depending on a packet length, simulated checksum offloading

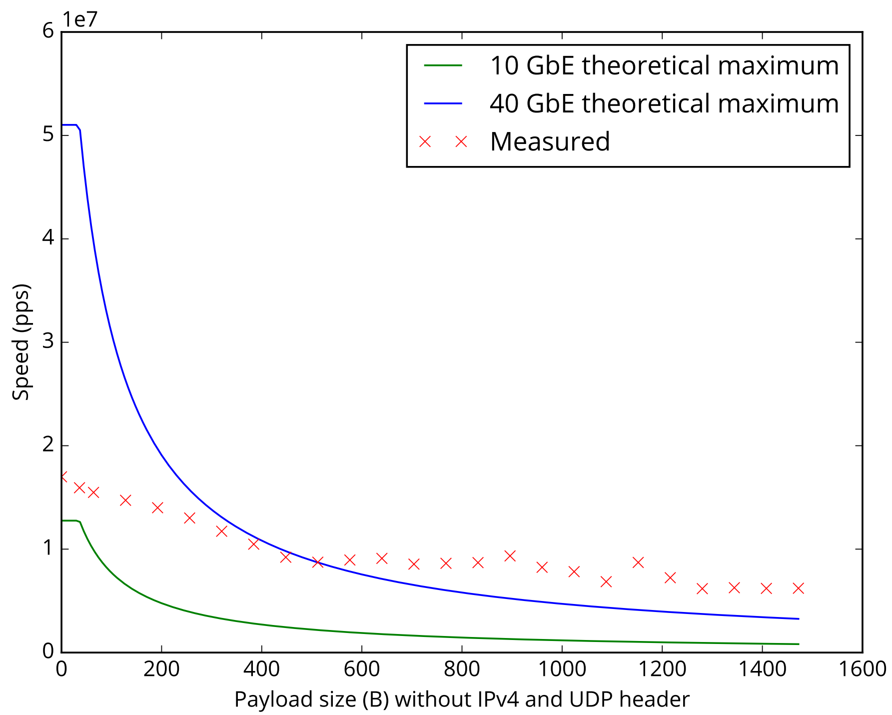

Figure 6.9: Speed compared with theoretical maximums, simulated checksum offloading

6.3.5 Flow Insertion

Following test case measures the performance in terms of flow insertion times. For this purpose, a session of 10 million flows is generated. Flows consist of a single packet and go through the *Slow Packet Scheduler*. The number of packets should correspond to the number of flows generated.

Number of flows	2^{21} to 2^{24}
Flow beginning	Starting at 0 ns, incremented by 1 ns
Number of packets in a flow	1

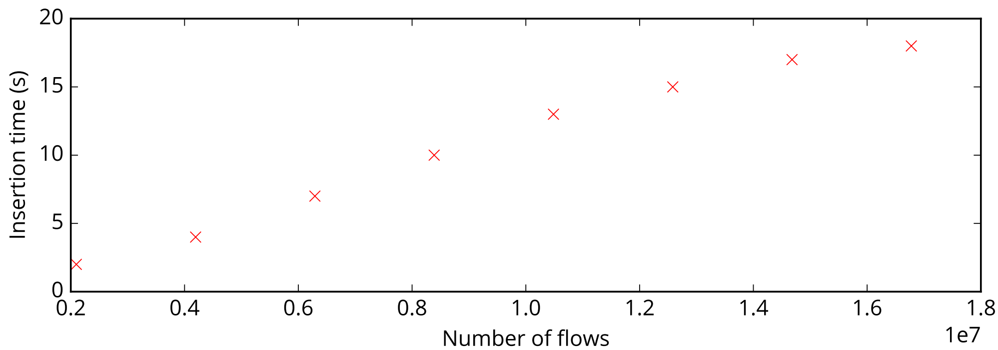

Figure 6.10: Flow insertion times

Measured values plotted in Figure 6.10 can be used to calculate insertion speeds, which are roughly 1.1×10^6 flows per seconds for all the measured values. The session loading time is also inconvenient, and can take up a few minutes to complete. There are several other problems, such as session file size (several gigabytes) and FLOR taking up more RAM.

6.3.6 Meta-flow Insertion

This test case demonstrates the advantage of using meta-flows instead of standalone flows, if possible. For this comparison to be more fair, some of the randomization features are used with an IPv6 / UDP meta-flow.

Number of flows	Maximum possible value ($2^{32} - 1$), in a single meta-flow
Flow beginning	0 ns
Number of packets in a flow	1

In this test case, 2.1×10^6 flows were inserted into *Slow Packet Scheduler* per second. This is more than twice as fast when compared to the standalone flows, even with IP addresses and ports randomization.

The cost will be higher if multiple meta-flows are used, due to the fact they need to be reinserted into the simulation calendar. Other than that, using meta-flows is recommended whenever possible as they have much lower memory overhead.

6.3.7 Slow Packet Scheduler

The last performance test evaluates the speed of the *Slow Packet Scheduler* unit, depending on the number of flows in its event calendar. Maximum speed of a flow is limited to 1 pps, thus many flows are needed to have visible effects on timing.

Number of flows	2^{16} to 2^{26}, with an exponential increment
Flow beginning	0 ns
Flow intensity	1 spp
Number of packets in a flow	Maximum possible value $(2^{32}-1)$

Low overhead, simplistic Ethernet frames are used again in this test case. In Figure 6.11, results show that *Slow Packet Scheduler* is able to schedule up to 2^{22} flows without visible slack issues. This should provide a sufficient performance for most of the FLOR's use cases.

The unit cannot keep up with higher amount of flows and its performance eventually drops to slightly over 2×10^6 packets per second.

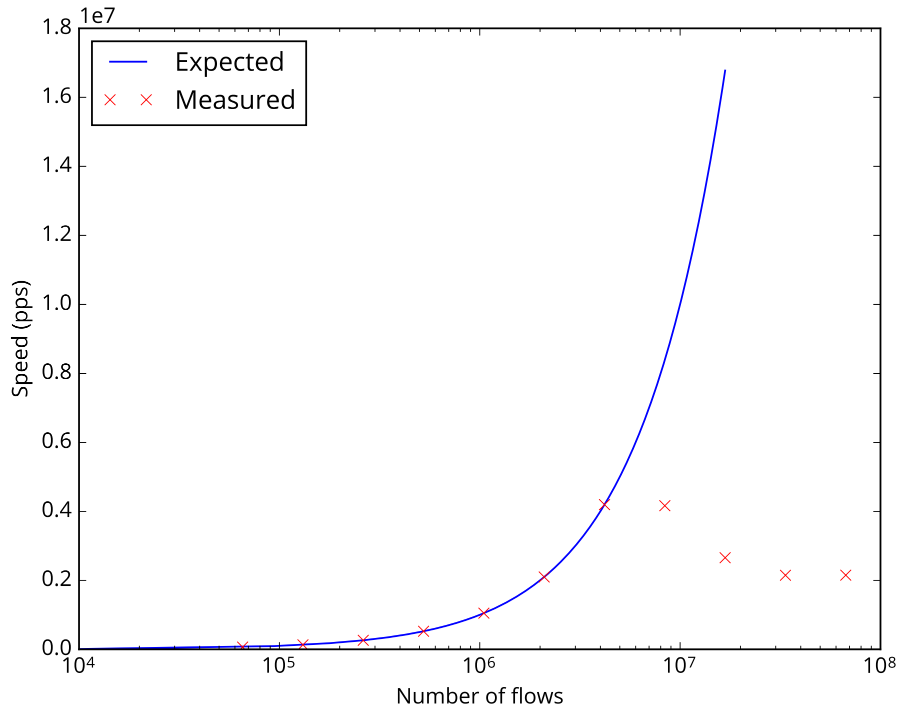

Figure 6.11: Performance of *Slow Packet Scheduler* unit

6.4 Comparison with Related Work

Comparison with related work using the same testing environment and use cases was not performed due to technical difficulties associated with such a task. It is contentious what should be considered as a same testing environment and different hardware usage is not avoidable considering requirements of the generators. It is still possible to meaningfully compare the achieved results with the information provided by the respective authors of related work.

In the current implementation, FLOR is able to compete with other open-loop generators presented only for simple use cases, generating a small amount of flows. Even in the case of generating just a single flow, the overhead associated with flow management makes FLOR a less efficient solution. Generators able to generate multiple flows by per-packet randomization, such as *Ostinato* or *MoonGen*, easily outperform FLOR also in terms of transmission rate.

However, comparison with those generators can be irrelevant due to the different level of flexibility they provide. Packets, the user is able to generate, are usually restricted to a single format and the generated traffic cannot be controlled on the flow level. FLOR, on the other hand, is designed so that packets in the flow can differ from each other, requiring checksum recalculation and resulting in a performance impact even when using a checksum offload feature.

Comparing FLOR to the closed-loop generators is a similar case. FLOR is not able to provide all the features for a fair comparison. On the other hand, closed-loop generators suffer from a bad performance and some of them were not even tested with speeds above 1 Gbps. In test cases FLOR can handle it would easily outperform such generators.

6.5 Performance Bottlenecks

Throughout the testing, FLOR's code was profiled to reveal issues affecting its performance. Although micro-optimizations are always possible, two main bottlenecks were found.

The first one is a *Distribution Tree* traversal, affecting performance with higher amount of flows, as shown in Subsection 6.3.1. The main advantage of stochastic approach to planning flows is the flexibility. Moreover, the *Distribution Tree* can be enhanced or replaced with another way of transforming probability distributions.

Profiling shows that CPI rate of the distribution tree is high, especially in the code deciding which child needs to be picked. One enhancement could be improving the tree's structure, so that the fact that some flows are more likely to be chosen is exploited. Making a tree similar to the Huffman's can make the traversal more predictable, branch prediction and memory prefetching more efficient and the whole code faster. Maintaining such a tree structure is more complex, making this a trade-off between lookup and modification time.

The second bottleneck observed is related to the checksum calculation and packet length, as measured in Subsection 6.3.3 and discussed in Section 5.8. With shorter packets, the bottleneck consist of checksum computation itself. As FLOR allows to specify only a limited payload, checksum computation time is bounded by this limit. With longer packets, the need to zero-initialize packet buffers takes over as a hotspot.

Probably the best solution to this problem is to implement the checksum offloading feature in hardware, as demonstrated in 6.3.4. With the checksum offloading, the limiting factor becomes the sending loop itself with no obvious additional hotspots.

Chapter 7

Conclusions

The aim of this project was to design and implement a software tool for realistic traffic generation. The application should be able to perform this task at high link speeds, satisfying the needs for testing today's devices. It also needs to be configurable on the IP flow level.

First, an overview of the basic protocols from the Internet Protocol Suite was given. Based on this information, the term flow was specified, as used in this thesis.

Internet traffic generator types were explained and some relevant examples provided. There is no tool which would suit the specification at the present time. Generators that are capable of producing network traffic with the required realism are slow and usually distributed across multiple network nodes. On the other hand, generators providing enough performance are lacking the required capabilities.

An architecture of the software flow generator was proposed. A stochastic process was selected as a method of flow planning. The proposed design was created with scalability in mind and overcoming all the problems encountered before implementation.

The designed software was implemented using C++ language into a flow generator called FLOR, featuring both command line and graphical user interfaces. Two sets of tests were created. The first examines FLOR's flow timing accuracy, while the second one measures its performance. Performance bottlenecks and possible ways to overcome them were discussed. Results obtained from testing were compared with the related work.

There are several improvements that can be done. First off, some of the planned features were not yet implemented. Those features were not needed for this thesis, but may be required in the future.

Section 6.5 discusses main performance problems with FLOR and also some possible improvements were mentioned. These need more attention.

DPDK implementation should be properly profiled and checksum offloading support implemented to make FLOR available for as many users as possible.

Apart from implementing checksum offloading, there are other possible ways to accelerate FLOR using hardware. FPGA-equipped COMBO cards and the NetCOPE framework can be used to port some of the functionality, like packet assembling, to the hardware. Some advanced features and a better packet timings are other improvements to be considered.

FLOR uses a lot of code that is hard to optimize in terms of program flow. It is possible to extend the used build system to execute such code in meaningful manner and use compiler-provided profile-driven optimizations. Preliminary results show that this can improve performance of some code up to 10%.

Bibliography

[1] Access to libsze2 library, *Liberouter* [online], [Accessed 20 May 2016]. Available from: https://www.liberouter.org/access-to-libsze2-library/

[2] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, 2013. [online], Rev. 3.21. Sunnyvale. Available from: http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/

[3] *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, 2015. [online], Rev. 3.22. Sunnyvale. Available from: http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/

[4] CONTA, A., S. DEERING and M. GUPTA. RFC 4443: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. *IETF* [online]. 2006 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc4443

[5] DEERING, S. and R. HINDEN. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification. *IETF* [online]. 1998 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc2460

[6] EMMA, D., A. PESCAPE and G. VENTRE. Analysis and Experimentation of an Open Distributed Platform for Synthetic Traffic Generation. *Future Trends of Distributed Computing Systems, IEEE International Workshop*. IEEE, 2004, pp. 277–283. DOI: 10.1109/FTDCS.2004.1316627.

[7] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F. and G. CARLE. MoonGen. *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 2015, pp. 275–287. DOI: 10.1145/2815675.2815692.

[8] HERNÁNDEZ-CAMPOS, F. *Generation and Validation of Empirically-Derived TCP Application Workloads*. Chapel Hill, 2006. Dissertation. University of North Carolina.

[9] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2016. [online], Santa Clara. Available from: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[10] *ISO/IEC 14882:2011*. 3rd ed. Geneva : International Organization for Standardization, 2011.

[11] QUITTEK, J., T. ZSEBY, B. CLAISE and S. ZANDER. RFC 3917: Requirements for IP Flow Information Export (IPFIX). *IETF* [online]. 2004 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc3917

[12] RFC 1122: Requirements for Internet Hosts – Communication Layers. *IETF* [online]. 1989 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc1122

[13] RFC 792: Internet Control Message Protocol. *IETF* [online]. 1981 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc792

[14] POSTEL, J. RFC 768. *IETF* [online]. 1980 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc768

[15] RFC 793: Transmission Control Protocol. *IETF* [online]. 1981 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc793

[16] RFC 791: Internet Protocol. *IETF* [online]. 1981 [cit. 2015-12-31]. Available from: http://tools.ietf.org/html/rfc791

[17] SOMMERS, J., H. KIM and P. BARFORD. Harpoon: A Flow-level Traffic Generator for Router and Network Tests. In: *Proceedings of the joint international conference on Measurement and modeling of computer systems*. ACM, 2004, pp. 392–392. DOI: 10.1145/1012888.1005733.

[18] VARET, A. and N. LARRIEU. Realistic Network Traffic Profile Generation: Theory and Practice. *Computer and Information Science*. 2014, vol. 7, no. 2. DOI: 10.5539/cis.v7n2p1.

[19] VIGNA, S. Xoroshiro+ / xorshift* / xorshift+ generators and the PRNG shootout, *Sebastiano Vigna* [online], [Accessed 20 May 2016]. Available from: http://xoroshiro.di.unimi.it/

[20] VISHWANATH, K. V. and A. VAHDAT. Swing: Realistic and Responsive Network Traffic Generation. *IEEE/ACM Transactions on Networking*. 2009, vol. 17, no. 3, pp. 712–725. DOI: 10.1109/TNET.2009.2020830.

[21] VOSE, M. D. A Linear Algorithm for Generating Random Numbers With a Given Distribution. *IEEE Transactions on Software Engineering*. IEEE, 1991, vol. 17, no. 9, pp. 972–975. DOI: 10.1109/32.92917.

Appendices

List of Appendices

A Distribution Tree **47**

A.1 Tree Traversal 47

A.2 Node Insertion 48

A.3 Node Removal 49

B Build System and Dependencies **51**

C Graphical User Interface **52**

D Deterministic Flow Planning **54**

Appendix A

Distribution Tree

Distribution Tree is a rooted binary well-balanced tree used to transform random numbers with uniform distribution into numbers representing events with discrete probabilities. It is aimed towards space efficiency and fast add/removal operation.

Algorithms with constant time complexity for number conversion exists, but their space complexity and the need of preprocessing makes them unusable. Distribution Tree provides $O(n)$ space complexity and $O(\log_2(n))$ time complexity for all mentioned operations.

Leaves in the tree represent probabilistic events. Other nodes contain a sum of probabilities of all the events located in it's left subtree. Every non-leaf node has exactly 2 children.

The leaf nodes also contain event identification they represent. In our use-case, the event and the corresponding leaf are deleted when they were selected the given number of times. It's convenient to store this information with the leaf too. In the following illustrations this extra information is hidden for the sake of readability.

A.1 Tree Traversal

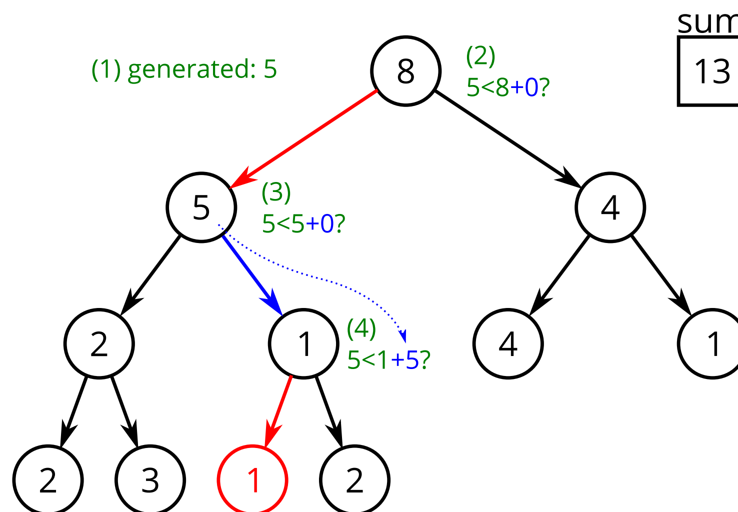

Figure A.1: Distribution Tree traversal

Tree traversal is used to transform a random number and it is illustrated in Figure A.1. It can be interpreted as a binary search on cumulative distribution function (effectively calculating it's inverse). The difference is that the nodes does not contain accumulated values, so we need to calculate them ourselves. This design choice was necessary to keep the tree updates simple. The algorithm consists of the following steps:

1. Generate a random number between zero (inclusive) and the sum of event probabilities in the tree (exclusive). (**(1)** in the figure)
2. Set the root as a *currently processed node* and an *accumulated value* to 0.
3. If the *currently processed node* is a leaf node, return the event identification associated with it. Otherwise continue.
4. If the generated value is less than the sum of node probability and the *accumulated value*, set the left child as the new *currently processed node* (**(2)**, **(4)**). Otherwise add the node probability to the *accumulated value* and set the right child as the new *currently processed node* (**(3)**).
5. Return to step 3.

In the presented algorithm we expect the tree to be non-empty. If this condition is not met the operation is meaningless.

A.2 Node Insertion

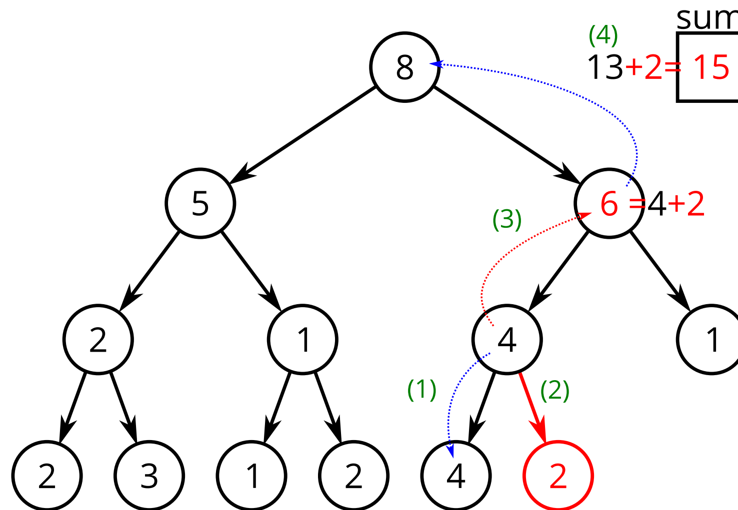

Figure A.2: Inserting a node to the Distribution Tree

When we want to include a new event, we need to insert a node representing it to the tree. The existing leaf node is transformed to a non-leaf node by attaching a copy of itself as a left child and the new node as a right child. Algorithm follows (illustrated in Figure A.2):

1. If the tree is empty, set the *newly added node* as the root, it's probability value as the sum of event probabilities and end the algorithm. Otherwise continue.
2. Pick the first leaf in breadth-first order and set it as *currently processed node*.
3. Make a copy of *currently processed node* and append it as a left child (**(1)**).
4. Connect the *newly added node* as a right child of the *currently processed node* (**(2)**).
5. Starting from the *currently processed node*, update the tree as follows (**(3)**):
   (a) If the *currently processed node* is the root, go to step 6.
   (b) If the *currently processed node* is a left child, add the probability of the *newly added node* to the parent's probability value.
   (c) Set the parent as the new *currently processed node*.
   (d) Return to step 5a.
6. Update the sum of event probabilities (**(4)**).

A.3 Node Removal

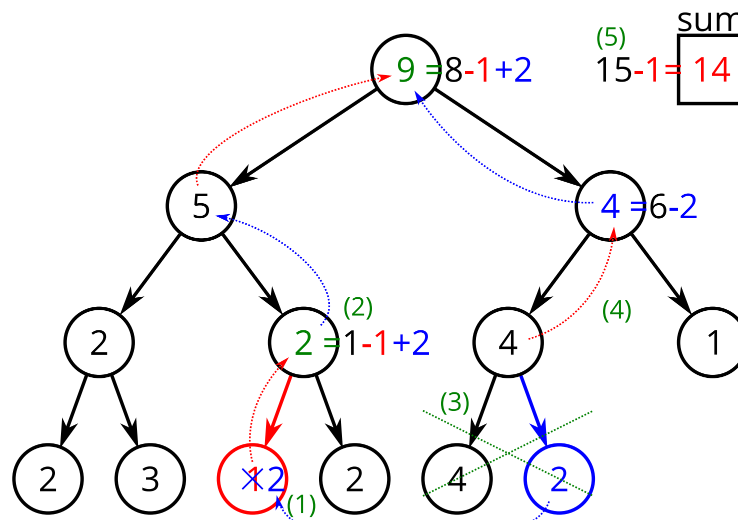

Figure A.3: Removing a node from the Distribution Tree

The idea behind node removal is to remove the last 2 leaves in the breadth-first order instead. Left one will be collapsed into it's parent, making it a new leaf (the first one in the breadth-first order). The other one is placed instead of the removed leaf. This requires two update operations on the non-leaf nodes. Steps of the following algorithm are illustrated in Figure A.3.

1. If the *node to be removed* is the root, delete it and end the processing. Otherwise continue.
2. If the *node to be removed* is the last node in the breadth-first order, skip to step 4.

3. Replace the *node to be removed* with the last node in the breadth-first order and update the tree using the following steps ((**1**), (**2**)):

   (a) Save the difference between probability value of *node to be removed* and probability value of the last node in the breadth-first order as a *difference value*.

   (b) Set the *node to be removed* as a *currently processed node*. Replace the content of the *currently processed node* with a data from the last node in the breadth-first order.

   (c) If the *currently processed node* is the root, continue with the step 4.

   (d) If the *currently processed node* is a left child, add the *difference value* to the parent's probability value.

   (e) Set the parent as the new *currently processed node*.

   (f) Return to step 3c.

4. Remove the last node in the breadth-first order and update the tree using the following steps ((**3**), (**4**)):

   (a) Save the probability value of the last node in the breadth-first order as a *subtrahend value*.

   (b) Copy the content of a left child of the last non-leaf node in the breadth-first order into it's parent.

   (c) Delete the last two leaf nodes in the breadth-first order, making their parent the first leaf node in the breadth-first order.

   (d) Set the first leaf node in the breadth-first order as a *currently processed node*.

   (e) If the *currently processed node* is the root, go to step 5.

   (f) If the *currently processed node* is a left child, subtract the *subtrahend value* from the parent's probability value.

   (g) Set the parent as the new *currently processed node*.

   (h) Return to step 4e.

5. Update the sum of event probabilities ((**5**)).

Appendix B

Build System and Dependencies

CMake[1] was chosen as a build system, with required version 3.0.2 or higher. It can be obtained from the homepage or installed using a distribution's package system. All major GNU/Linux distributions provide CMake package and most of the dependencies.

Building requirements:

- GNU/Linux OS with pthread support
- C++11 compliant compiler and a standard library implementation (`GNU GCC 4.8.5` and `clang 3.4.2` were tested)
- Qt5 Widgets
- NUMA policy library (called `libnuma` on most distributions)
- RapidJSON[2] SAX library, a recent version with `std::iostream` wrappers support

To build the FLOR, a CMake variable `BACKEND` needs to be set, either by using `cmake-gui` or by specifying `-DBACKEND=<backend class>` within the CMake command line arguments. Implemented *Backends* can be found in the `source_tree/Backends` directory. It is also recommended to set `CMAKE_BUILD_TYPE` variable to `Release`. Then run `make` command.

Various *Backends* can have their own dependencies and require more build steps. Notes on how to implement a *Backend* can be found in the `source_tree/Backends/README` text file.

As an example, if you want to build FLOR with the Pcap *Backend* off-tree, with source directory called `FLOR` and destination `FLOR-build`, you could run the following commands:

```
~$ mkdir FLOR-build
~$ cd FLOR-build
~$ cmake ../FLOR -DBACKEND=Pcap -DCMAKE_BUILD_TYPE=Release
~$ make
```

Resulting binaries can be found in the build directory under the `flor-gui` and `flor-cli` names.

[1] https://cmake.org/
[2] http://rapidjson.org/

Appendix C

Graphical User Interface

This appendix contains a few screenshots of the FLOR's GUI. Figure C.1 shows the main interface with a loaded session. Meta-flows are listed in the middle part, NUMA and affinity settings are located above. In the lower part of the window, application log messages are written.

In Figure C.2, an implemented meta-flow general setup dialog is shown. Figure C.3 contains a view containing statistics of a running simulation, replacing the meta-flow view when the simulation is started.

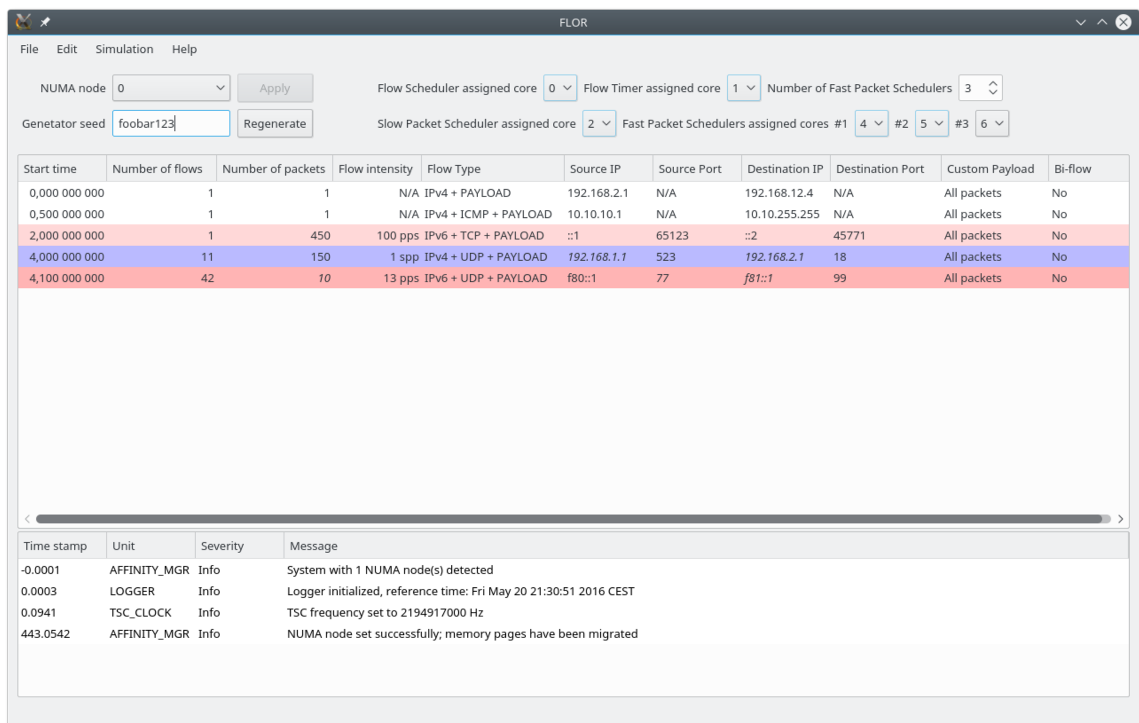

Figure C.1: Main window with meta-flow overview

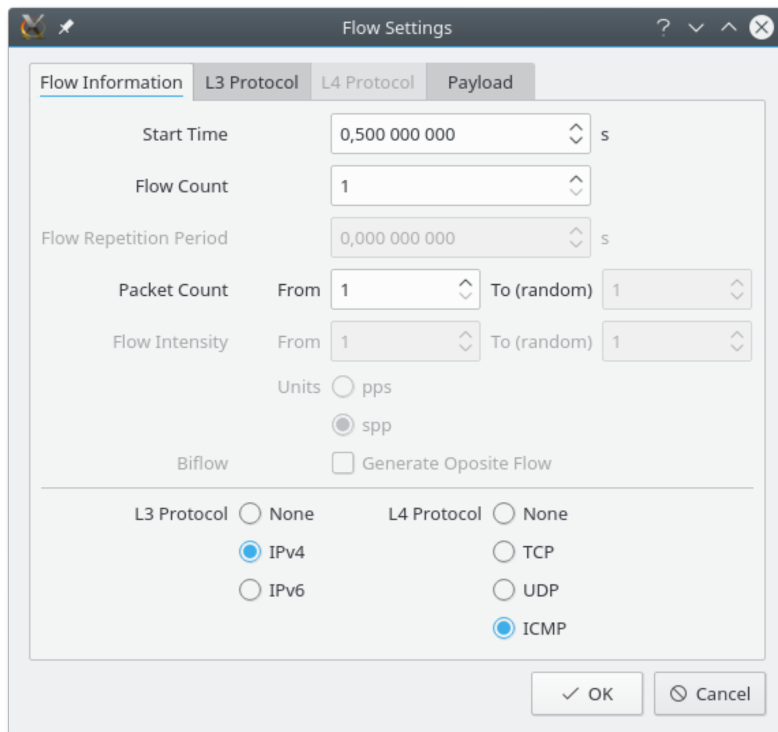

Figure C.2: Meta-flow editor

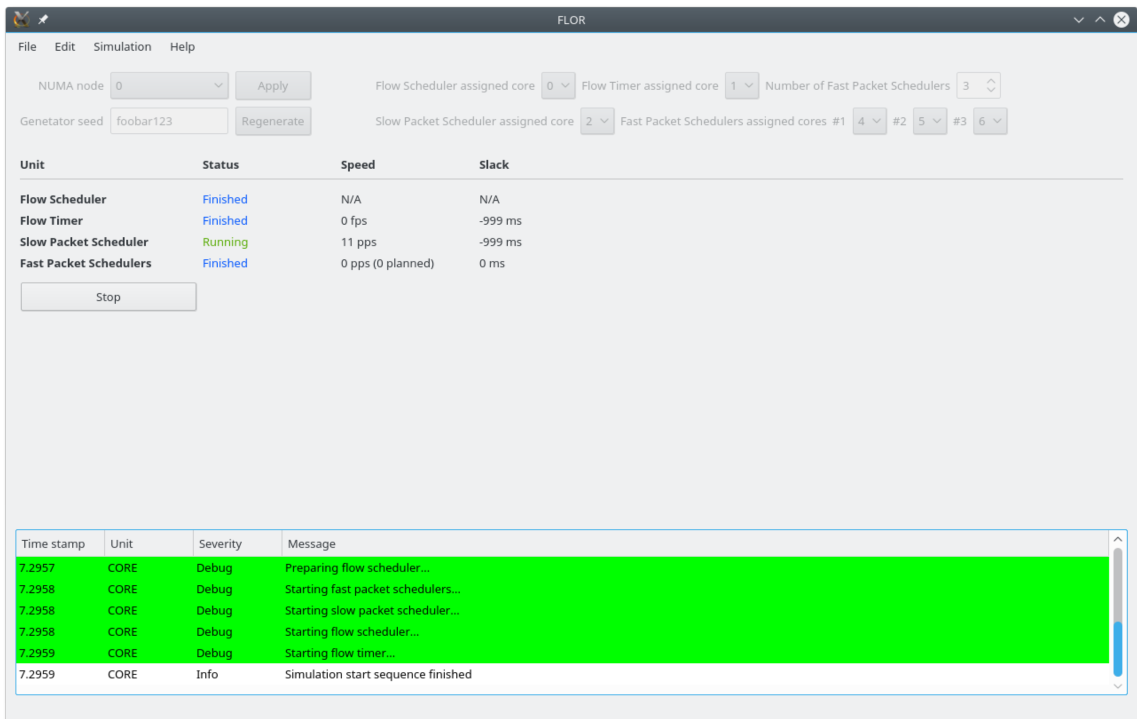

Figure C.3: Simulation overview

Appendix D

Deterministic Flow Planning

Another version of the FLOR was created to be used as a reference implementation in some of the tests. Instead of a stochastic flow planning, it uses heap-based simulation calendars, similar to the one used for *Slow Packet Scheduler*. Architecture of a modified *Core* is illustrated in Figure D.1.

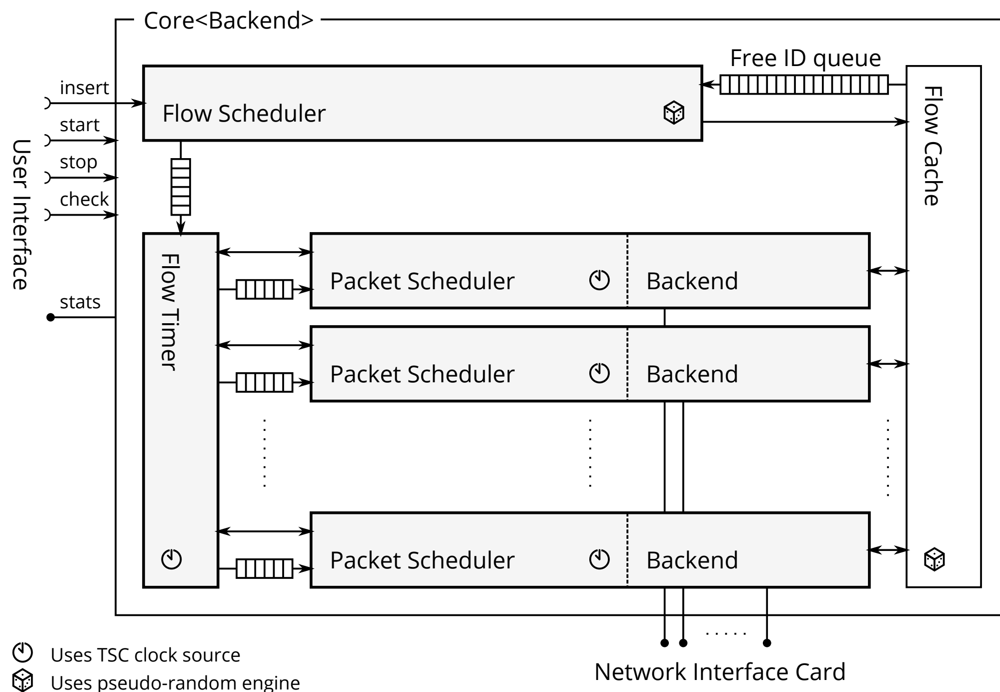

Figure D.1: Alternative FLOR architecture with deterministic planning

Distinct *Slow Packet Scheduler* is no longer necessary and load balancing can be simplified. In order to make it work we need to store the aggregated intensity, and to be able to keep it up-dated, calendar entries have to contain both period and flow intensity (avoiding recomputation of one or another). FLOR with such *Packet Scheduler* is slightly less space efficient compared to the one using the *Distribution Tree*.

Another advantage of the fact that there are no distinct schedulers is that all the available cores and transmission rings can be used by the units planning fast flows. This may lead to the better resource management and higher overall performance in several specific use cases.

Features differ only slightly. User is able to specify intensities with the mpps (milli-packets per second) precision. The slowest flow that can be inserted this way is a flow with 1000 s period. Flows with a lower intensity have to be broken down to multiple flows.

Even though deterministic flow planning can be better in some situations, it lacks the flexibility of a stochastic one, mentioned in Section 6.5.