Design and Implementation of a Traffic Generator using Unified Traffic Modelling

by

Björn Bylund, Nicklas Blomqvist

LIU-IDA/LITH-EX-A–15/033–SE

2015-06-12
Final thesis

Design and Implementation of a Traffic Generator using Unified Traffic Modelling

by

Björn Bylund, Nicklas Blomqvist

LIU-IDA/LITH-EX-A–15/033–SE

2015-06-12

Supervisor: Vengatanathan Krishnamoorthi
Examiner: Niklas Carlsson
Abstract

This thesis describes the design and implementation of a traffic generator that can simulate the traffic of tens of thousands of networking devices from a given traffic model. It is designed to handle traffic models created with Unified Traffic Modelling. The traffic generator is then evaluated and different solutions are compared in an effort to find the best solution for each issue. This thesis is meant to serve as a guideline for future development of traffic generators by providing insight into the problems faced during the development of one.
Acknowledgements

We would like to thank Ericsson for being so welcoming and for the opportunity to work on this project. We would like to especially thank Patrik Sandahl and Lars-Anders Cederberg at Ericsson, our examiner Niklas Carlsson and our supervisor Vengatanathan Krishnamoorthi for their guidance and support during the whole project.
Contents

1 Introduction 1
1.1 Motivation .................................................. 1
1.2 Problem formulation ................................. 2
1.3 Contribution ................................................. 3
1.4 Limitations .................................................. 4

2 Background 5
2.1 Context .......................................................... 5
2.2 Traffic generators ........................................... 5
  2.2.1 Traffic generator categories ....................... 5
  2.2.2 Evaluation of traffic generators ................. 6
2.3 Traffic modelling ............................................ 7
  2.3.1 Unified traffic modelling ............................ 7
2.4 Load balancing ............................................. 10
2.5 Erlang .......................................................... 11

3 Design 13
3.1 Generator application ................................. 13
  3.1.1 API ......................................................... 14
  3.1.2 Traffic model ........................................... 15
  3.1.3 Unit ......................................................... 16
  3.1.4 Service ................................................... 16
  3.1.5 Probe ....................................................... 17
3.2 Design choices ............................................. 17
  3.2.1 The architecture ...................................... 18
  3.2.2 Unit placement ........................................ 18
  3.2.3 Synchronisation ....................................... 20

4 Evaluation 22
4.1 Testing details ............................................ 22
  4.1.1 Testing environment ................................. 22
  4.1.2 Testing method ....................................... 23
  4.1.3 Traffic models ....................................... 24
4.2 Benchmark .................................................. 25
Chapter 1

Introduction

In the last decade, the number of users, as well as the number of different types of applications on the Internet has rapidly increased. This development has lead to network traffic with a larger variety, as well as an extreme increase in volume, speed and number of concurrent users [12] [15]. Due to these developments, network systems need better test tools and test data to be prepared for current and future demands. Traffic generators are often the most preferable way of evaluating network performance [5]. However, current traffic generators are far from perfect and there are still many open problems to be addressed. For example, traffic generators have been shown to fail to generate traffic according to the given specification as well as fail to scale well to large traffic volumes [8]. This thesis investigates how to best design and implement a scalable traffic generator with traffic based on real world traces.

1.1 Motivation

Traffic generators create network traffic based on a given specification. The most prominent application of these generators is in testing environments where systems get tested before deployment. When generating traffic for these tests, it is important that the characteristics of the traffic is representative of real world conditions, with potentially hundreds of thousands of concurrent networking devices. If it is not, the systems cannot be properly tested for their intended use and may malfunction when exposed to the real conditions. Because of this, new traffic generators have to be created that are able to handle new traffic models and new performance requirements. With traffic volumes rapidly increasing and new services continuously being added it is important to build a traffic generator that can follow this development. Ideally, a traffic generator should be implemented in such a way that it easily can adapt to changing traffic characteristics. It should also be able to handle increasingly complex models and scale with future demands,
such as to allow testing of future networks with continuously increasing data rates and higher number of concurrent users. Traffic generators can be implemented in both hardware and software and while hardware solutions are generally more precise and have better performance than software solutions, they are also more expensive and lacks the flexibility that is given by software in both deployment and modifications [8].

1.2 Problem formulation

This thesis designs and implements a scalable traffic generator which has the following requirements:

- It shall simulate the traffic from multiple devices by recreating the traffic specified in the given traffic model.

- The traffic simulated shall be between a device and a service where a service represents a web service or a video streaming service for example.

- Each device shall have a unique IP address and can have multiple concurrent connections to different services.

- The amount of devices that are simultaneously simulated shall be able to be in the order of tens of thousands, which implies that the amount of concurrent connections can be even higher.

These requirements have two main problem areas that can counteract each other and a balance must be found between them. First the traffic generator shall be able to recreate the traffic specified in the traffic model accurately and secondly the traffic generator must be able to scale well to simulate the required amount of devices. The accuracy of the generated traffic must not be lost when increasing the number of simulated devices. To satisfy the demands of the recreation of traffic there are four major issues that need addressing [11]:

- In order to be able to recreate the data in the traffic model, the traffic generator needs to continuously read data from the model. Minimising the latency of this process increases the precision of the output as well as the potential throughput.

- The traffic model specifies certain inter-arrival times of the packets which needs to be followed as closely as possible to produce a satisfiable result. If the time from the event triggering to the response to it is too long, precision is lost.

- To achieve high accuracy and throughput, the time between the send event to the packet actually being sent through the network needs to be as short as possible. A shorter send path leads to increased accuracy and throughput.
A process must be allowed to run when it needs to send or receive packets so that a high accuracy can be achieved. If the scheduling is not responsive enough, precision is lost.

In addition to these issues, the traffic generator must be able to scale well over several cores as well as several machines to allow for the amount of concurrently simulated units that are demanded.

Several problems arise from these requirements that are important to address for a system designer. Example questions that arise in the areas of data storage, parallelism, software- and network performance as well as network architecture include:

- How do we best simulate tens of thousands of concurrent devices?
- How do we best implement a generator that utilises the available resources on a machine well?
- How do we scale the generator over several machines?
- How do we store and access the traffic model in a way that is fast and efficient?

1.3 Contribution

This thesis investigates different approaches for implementing a traffic generator that is scalable and can handle complex traffic models with the goal of providing a baseline for the development of future traffic generators. It iteratively explores the problems encountered during development and makes the following contributions:

- We propose a design for a traffic generator that can simulate the traffic generated by multiple networking devices and scale over several machines.
- We investigated the performance of our chosen design with an increasing amount of simulated devices using traffic models with different characteristics.
- We investigated the performance impact of storing the model in an SQL database instead of in memory.
- We investigated the performance of different content-blind load balancing algorithms for our system.
1.4 Limitations

There are two primary limitations of the thesis:

- We only considered traffic on the IP level due to the traffic model that the system was designed for.

- The simulated device always initiates the communication in the traffic generator, not the service. This was done because the device always chooses what traffic to generate.
Chapter 2

Background

Traffic generators are widely applicable and the terminology has therefore been used differently in different contexts. In this chapter we define the context and describe the background related to the traffic generator designed and implemented in this thesis.

2.1 Context

Traffic generators together with traffic models can be used to produce realistic internet traffic. This is very useful for companies dealing with networking. They can use this traffic to test their products in an controlled environment that is similar to real conditions. This thesis is written on behalf of Ericsson and studies methods to implement a traffic generator that is scalable and that can handle large and complex traffic models. The thesis serves as a guideline for the development of future traffic generators.

2.2 Traffic generators

A traffic generator generates network traffic according to a specification in the form of packets which are injected into a network. Traffic generators can be realised through software or hardware and are widely used for producing test data for networks and other applications [8]. Due to the increasing complexity of our networks and network traffic, traffic generators are becoming more and more relevant in the networking field [15] [21].

2.2.1 Traffic generator categories

Network traffic can be generated in many different ways, from following different protocols to how the generator distributes the packets [8]. There is a myriad of different traffic generator implementations available today, each
with their own properties. In general, traffic generators can be split into five different categories [21]:

- **Replay engines**: have the purpose of taking traffic that has been previously recorded and then reproduce that recording as accurately as possible. Tcpreplay [1] is an example of a traffic generator in this category which can modify and then replay traffic captured in libpcap format.

- **Maximum throughput generators**: focus on the amount of traffic generated. They are commonly used to evaluate end-to-end network performance. Iperf [2] and BRUTE [7] are examples of traffic generators in this category. Iperf and BRUTE are used to evaluate high speed network performance.

- **Model based generators**: generate traffic based on stochastic models. Such models can define the size and inter departure time of the generated packets. Distributed Internet Traffic Generator (D-ITG) [6] is an example of a traffic generator in this category which follows different probability distributions for the packet size and inter departure time.

- **High level and auto configurable generators**: use higher-level models of network traffic and automatic configuration based on live measurements to create output statistically similar to real traffic. HARPOON [23] and SWING [12] are examples of traffic generators in this category. HARPOON is a flow-level traffic generator and SWING captures packet interactions and generates live traffic similar to the captured traffic.

- **Special scenario generators**: work with specific network scenarios such as following a certain network protocol. EAR [17] is an example of a traffic generator in this category, which generates traffic that follows the IEEE 802.11 protocol.

In contrast to existing traffic generators, ours is the first to use traffic models created with Unified Traffic Modelling.

### 2.2.2 Evaluation of traffic generators

There are many different ways to evaluate the performance of a traffic generator. In this section we present and explain categories that can be seen as the most commonly used metrics for this purpose [21]:

- **Packet-level metrics**: are based on the broad statistics of the output from the traffic generator. Examples of metrics under this category are byte throughput, packet size distribution and packet inter arrival distribution. Being on the packet level, the metrics in this category analyse output data on the lowest level.
• **Flow-level metrics:** considers information about sequences of packets that belong to the same connection instead of each packet. These sequences are also called flows. Examples of metrics under this category are flow size distribution and flow volume.

• **Scaling characteristics:** contains metrics that are related to second order characteristics, such as burstiness and long-range dependence. Examples of metrics under this category are logscale diagram and multiscale diagram. These metrics consider all of the output together.

• **Quality of Service and Quality of Experience related metrics:** are based on the quality of the traffic generated. Examples of metrics under these categories are queueing behaviour, round trip time distribution and streaming quality. They are typically examining the output of the traffic generator on a user level.

### 2.3 Traffic modelling

The ability to recreate realistic traffic is something highly sought after by researchers and companies alike. This is something that is rather complex and to help with the process, traffic models have been created. A traffic model is a stochastic process which can be used to imitate the behaviour of real traffic and can be created in a large variety of ways.

#### 2.3.1 Unified traffic modelling

The model that is provided for us to use in this traffic generator is created using Unified Traffic Modelling (UTM)\(^1\). The core idea behind it is to have one or many independent units generating traffic similar to a user in the network based on recorded traffic. The data comes from measurements done at the IP layer in the access network’s gateway to the Internet. The measurements produce a packet header log where one packet is one entry in the log. Each entry contains information about packet size, packet direction, transportation protocol, time stamps, IP-addresses and other packet related information. The data is made anonymous by hashing the IP-addresses and since only the headers are recorded any payload has been discarded. The log is then processed through four steps.

**Step 1: Separating the data into flows**

The first step is to separate the different devices in the raw packet header log so that we only process traffic from a single device. Then the packet header log is further divided into so called flows. Each flow corresponds to a connection between the device and a service and will consist of the packets

\(^1\)UTM is a result of Ericsson research that has not been published.
sent on that connection. A service can be a website or a video streaming service for example. An active connection is treated differently depending on which point of view it is looked at. From the application it is considered active until it is closed by one of the peers. But since network resources are limited, the connection might get deallocated in the network after being inactive for a certain amount of time [18]. If a connection that has had its network resources deallocated transfers data it is treated as a new connection by the network which means that it gets new resources allocated. Because of this, UTM sees these flows as independent and a flow is divided into smaller flows where it is inactive for more than a specified time. The time itself can vary depending on the network, but in our data it is three minutes but can be as high as 30 minutes. The process is illustrated in Figure 2.1.

**Step 2: Aggregating the packet headers into objects**

It is nearly impossible to exactly recreate the raw packet header log due to the randomness of the network conditions. The data is collected at the service provider and the data is generated from the client side, which means that the data will have to go through the network again which will not treat it exactly the same. This means that the level of detail in the recorded data is higher than necessary. Therefore in an effort to transform the data into what it might have looked like on the client side, each flow is compressed. This is done by creating one object out of a set of packets that have an inter-arrival time less than a specified time. This time can vary but in the data provided to us 500 ms has been used. The uplink and downlink bytes of the aggregated packets are summarized in the object, the process is illustrated in Figure 2.2.

**Step 3: Combining flows into sessions**

As explained previously, after a period of inactivity from a device in the network, the resources of the connection are deallocated. Therefore, sim-
Similarly to how objects are created in flows, sessions are created from flows that have at least three minutes inter-arrival time to each other, illustrated in Figure 2.3. This is done so that sessions can be run independently from each other as they do not have any resources allocated in the network. A session can be seen as the traffic generated when a user picks up a networking device, sends a message, browses the web for a bit and then puts down the device.

**Step 4: Combining sessions into super sessions**

To be able to generate traffic with different characteristics, sessions are selected based on certain attributes to create super sessions, illustrated in Figure 2.4. They are executed by selecting one of the sessions to run at random. This randomness can be modified by changing a weight attached to each session in the super session. Once the session has been run from beginning to end, a new session is selected. To keep the distance between the sessions, a time-out period is stored in each session.
Load balancing aims to maintain good performance for a service. To do this the service cannot be overloaded. Load balancing is used to split the load between several nodes that run the same service. This is done so that there are enough resources available to handle a new request at all times. When talking about load balancing, there are many different algorithms and policies which can be divided into two categories: content-aware and content-blind balancing.

Content-blind solutions are unaware of the information in the request. They treat every request equally and make decisions according to a policy that is chosen for the balancer. Different policies can be chosen depending on the situation but the following policies are some of the normally used [13]:

- **Random Server Selection (RSS):** The balancer assigns the requests randomly between the nodes. This randomness is uniform and non-deterministic.

- **Round Robin (RR):** requests are assigned to each node in a circular order. This order is fixed but with the possibility for nodes to join and leave the RR.

- **Least Connections (LC):** prioritises nodes based on the number of connections it currently has. The node that has the least amount of connections will be the one that is selected.

- **Least Loaded (LL):** is similar to LC but prioritises nodes based on its load. This load is based on the node’s utilisation and capacity. The node with the lowest load will be the one that is selected.
Unlike content-blind approaches, content-aware approaches are aware of the application content of the request. Different requests can affect the server’s load in different ways. Some requests may require more resources than others but this is not taken into consideration with content-blind balancing. An aware balancer can analyse the request and decide which node that is the best candidate.

2.5 Erlang

The programming language we have chosen to use for the generator application is Erlang. It is a functional language with several features that are appealing to this type of application. Since the goal is to implement a traffic generator that can simulate concurrent devices in the order of tens of thousands, each with several independent connections, a programming language that is proficient with concurrency is very important. Erlang is a language that has been designed with concurrency in mind from the very start [19]. It uses actors [10] and offers a lightweight concurrency model with processes that are independent from the operating system which makes it possible to run hundreds of thousands of processes with a small memory footprint [9].

This is possible due to Erlang running in its own virtual machine which controls all the Erlang processes [10]. The processes in Erlang can be divided into either workers or supervisors. Workers are the ones that perform the task while supervisors keep track of workers [20]. Another advantage with Erlang is the message passing system which is how processes communicate with each other in Erlang, it is asynchronous and delivers in microseconds regardless of the amount of processes currently running [9]. As an example for demonstrating these features, message passing and process creation are one to two orders of magnitude faster in Erlang than in Java or C# [4]. It is due to these characteristics that Erlang has become the de facto implementation of actor-model systems [10] [16] [24].

In the book Erlang Programming by Cesarini et al. they write: "What makes Erlang the best choice for your project? It depends on what you are looking to build. If you are looking into writing a number-crunching application, a graphics intensive system, or client software running on a mobile handset, then sorry, you bought the wrong book. But if your target system is a high-level, concurrent, robust, soft real-time system that will scale in line with demand, make full use of multicore processors, and integrate with components written in other languages, Erlang should be your choice." [9].

We felt that this description fits very well with what we wanted to do. The requirements for the traffic generator are all about having a large number of units communicating in soft real-time scaling over several cores as well as computers. This together with that the amount of calculations each worker has to do is minimal, further reinforces the argument for Erlang and other actor based languages.
There are several other actor based languages available, many of which runs on the Java virtual machine, one of these languages is Scala which with its Actors library functions very similarly to Erlang [10]. Scala however does not seem to outperform Erlang [16] and the fact that it is a hybrid functional/object oriented language using an external library made us lean towards Erlang, which was built for tasks such as ours. Another option would be to use Kilim which is a framework for Java that has similar features as Erlang and the authors claims to have 3x faster message-passing than Erlang [24]. This statement does seem to have some validity [16] but Kilim is a rather young framework, first introduced in 2008 [24], which compared to Erlangs nearly 30 years of development made us opt for the more well known solution [19].
Chapter 3
Design

In this chapter we introduce and explain our chosen design for the traffic generator. First we will discuss the general architectural design and then go deeper into each component. Finally we present the major design choices we have made during the development, the different options we considered and our reasoning behind the choices.

The architecture of the traffic generator is illustrated in Figure 3.1. It contains three major components: (i) The distributor, which acts as the mediator for the system, (ii) the database, which is where the traffic model is stored and (iii) the generator application, which generates the traffic and can be run as both client and server. One of the key attributes sought after with the system was the ability to scale over several machines, the architecture supports this by allowing multiple client and server side nodes. To be able to receive requests from other applications, the distributor acts as an interface to our solution by providing a web API. With these requests, applications can create, delete, start and stop units in the generator. To be able to forward these requests, the distributor needs to know about all active nodes and this is solved by letting the nodes connect to the distributor on startup and register itself as a client, server or both. When creating a unit the distributor will distribute the work to a node, that has registered itself as a client, following a load balancing algorithm.

3.1 Generator application

While the distributor can be seen as the manager of the system, it is the generator application that handles the traffic generation. It receives requests from the distributor to create and run units which then starts generating traffic according to the assigned traffic model. The design that we have chosen for the application, illustrated in Figure 3.2, can be split into five distinct parts:
The API, which acts as an interface for the application.

- The traffic model, which is the internal representation of the real traffic model.

- The units, which act as clients and initiate the communication.

- The services, which act as servers and listen for requests.

- The probe, which measures the application’s performance.

The application can be configured to run only the unit part, service part or both. The API, the traffic model and the probe are always included. We designed the application this way to accommodate for different setups while still only needing one application. The typical work flow of the generator is illustrated in Figure 3.3.

### 3.1.1 API

The API is the gateway between the application and the distributor. It is implemented as a worker that on startup initiates a connection with the distributor. Once the connection has been established the API registers itself so that the distributor knows that the application is ready to receive requests. These requests come in the form of JSON-objects which the API interprets and then executes.
3.1.2 Traffic model

The traffic model component has two main responsibilities, handling tasks related to the real traffic model and selecting which session each unit is going to run. Its architecture is illustrated in Figure 3.4. A worker called Data_Provider which is supervised by Traffic_Model_Sup is the internal representation of the traffic model. It fetches traffic model data and provides it to the workers. The selection of sessions is handled by workers called SS_Worker which are also supervised by Traffic_Model_Sup. At startup the supervisor queries the database for the available super sessions (Section 2.3.1) and creates a worker for each one of them. When requested, a worker selects one of its sessions at random based on weights that have been assigned to them.
3.1.3 Unit

The unit part of the application represents the client side of the traffic generator. Its architecture is illustrated in Figure 3.5. They are created under the supervisor Unit.Sup and contain a handler which is responsible for the units communication with other processes. A supervisor called Unit.Worker.Sup and its workers are responsible for the traffic generation. When a unit receives a start command, the handler asks its super session for a session to run. Once it has been received, the handler spawns a worker for each flow in the session. Each worker reads the necessary data from the traffic model and initiates a connection with its specified service. Once a connection has been established, the worker sends the identifier of the session and flow it is running to the service so that it knows what traffic to generate. Once this is done it starts generating traffic according to its flow and once all the data has been sent and received, it closes its socket and dies. As mentioned in Section 2.3.1, each session ends with a time-out period which is a flow where no data is sent. This case cannot be found anywhere except for at the end of the session. If a worker detects such event, it knows that it is the last flow in the session and reports to the handler that all the flows have completed. Once the handler receives the call, it starts the process again by asking its super session for a new session to run.

3.1.4 Service

The service part of the generator represents the server side of the traffic generator and runs the different services in the traffic model. Its architecture is illustrated in Figure 3.6. Its job is to accept incoming connections from the units and generate the appropriate traffic according to the session and flow given. This is done by having a listener for each service called Service.Listener that is supervised by Service.Listener.Sup. They listen for
connections and when they receives one they start a new service worker and gives it the socket before returning to listening for connections again. This new worker, called Service_Worker continues the communication with the connector and has Service_Worker_Sup as supervisor. The connecting unit initiates the communication by sending what flow in what session it wants to run. Once the service worker has received the flow it behaves the same way as a unit worker would. All services that needs to be started are provided by the distributor when the generator establishes a connection with it.

### 3.1.5 Probe

Inspired by the work of Mukherjee et al. [22], we added a probe in the form of a worker to the generator as a way to measure the performance of the unit/service workers. The probe has the same scheduling priority as the unit/service workers and performs similar tasks as them. The difference is that the probe also measures a variety of performance related metrics like response time and latency in the IP-stack. It calculates average values of these metrics and logs them to a file. This gives us a way to measure the performance from the inside of the application rather than looking at the traffic generated and general system measurements.

### 3.2 Design choices

When developing an application there are many design choices that have to be made. Since we are investigating implementation methods the reasoning behind these choices are important. In this section we will present the design choices that we have made, the alternatives we considered and our rationale for choosing them.
3.2. DESIGN CHOICES

3.2.1 The architecture

The fundamental problem in the thesis was to generate traffic between a client and a server. Naturally, this lead to an initial architecture which was divided into one client side and one server side. Further we wanted to be able to simulate tens of thousands of units and keep up with future demands. Therefore, the architecture had to be further developed with scalability in mind. To do this we felt that adding support for scaling over several machines would be necessary. Hence, the system needed to be able to handle several nodes which work side by side to be able to simulate more units. The client side would naturally consist of a group of all the client nodes and the server side of all server nodes.

Distributor

We wanted to have the nodes as loosely coupled as possible to minimise their inter-communication. This would lead to less overhead and there would be no dependency between them. To still keep track of the nodes we added an application that could handle the nodes. This application would act as a centralised distributor which all nodes have to connect to. This way the nodes only depend on the distributor. As described previously, it receives requests which it then forwards to one of the nodes.

Network

Since the intended use case of the generator is to analyse the traffic generated we had to ensure that it had characteristics that were observable. We decided to assign one port for each service to listen on so that all traffic to a specific service could be identified by the port. So then we had services running on the same port on different nodes, the next problem was how to decide which server that the unit would connect to. We did not want to implement it in the distributor since the time between a unit’s creation and start could be long and this would make it difficult to decide which server node to use at creation. We did not want to have the units asking the distributor every time it would connect either because that would overload the distributor very quickly. Because of these issues we decided to use a standard load balancing solution with IP-forwarding. This gave us both load balancing and an abstraction of the server side from the clients point of view. The clients would all connect to the same IP as if there was only one server but the load balancer would forward the request to a server node chosen by a load balancing algorithm.

3.2.2 Unit placement

In Section 2.3.1 we described UTM including the concept of units, super sessions, sessions and flows. It seemed natural to us that we would use
these concepts in our design as well. The question was where the generator application and the distributor would fit into that hierarchy. We identified the following three candidate approaches:

- **Flows in the generator**: by letting the distributor handle the units and their super sessions, the generator application would only be sending and receiving data. It would receive a request for each flow to be started by the distributor. The big weakness with this approach is that there would be a lot of communication between the distributor and the generator application to create all of the flows and the distributor would have to keep track of a large number of units. The advantage would be that everything is controlled from the distributor which makes load balancing easier to handle.

- **Units in the generator**: one other approach is to let units be fully implemented in the generator. The units would then run sessions and spawn flows which would send and receive data. This would make the distributor only responsible for sending commands for creating, deleting, starting and stopping units to the generator application, further minimising the communication needed between the two. The downside with this option is that the distributor would have to load balance units instead of flows which makes it more difficult as they are potentially permanent and the workload can change for a unit with time. This option also demands a more complex implementation of the generator application in order to handle the functionality of a unit.

- **Sessions in the generator**: a compromise between the two approaches is to have units in the distributor and run the unit’s sessions in the generator. The distributor would send which session to run to the generator application which would handle the creation of the flows. While it does significantly decrease the amount of communication between the distributor and generator application compared to the flow solution, it does little to solve the problem with having the distributor keep track of the large amount of units.

**Our choice**

We chose to go with implementing units in the generator due to a number of reasons. First and foremost we felt that keeping track of the units was more suited to the responsibilities of the generator rather than the distributor which makes the system parts more independent and makes the structure of the system as a whole more easily understandable. Another benefit is the decreased amount of network traffic which we felt is rather significant. Finally we felt that the issue of load balancing was not significant enough to warrant choosing another option.
3.2.3 Synchronisation

Since a service needs to communicate with a connected unit according to its current session, we needed a way to synchronise this session between the unit and the service. When a unit starts a new session, the flows in it can contain traffic to different services which means that the service needs to be informed which flow to generate. This problem is then further complicated by that these services does not have to be on the same server node, this is chosen upon each connection to a service. Because of this, we need on the fly synchronisation which we considered three options for:

- *Erlang message passing*: Erlang has a powerful message passing system built in which is well suited for synchronisation between workers. One option would be to use this as a way to send the session and flow information from the unit worker to the service worker. One of the major issues with this approach is that the unit would have to know beforehand which server node it will connect to, which means that server side load balancing through IP-forwarding would not be an option.

- *Initialisation message*: since we are already sending data between the clients and the servers, one option would be to use that channel to synchronise between them. This would be done through an initialisation message sent at the start of each connection from the client which informs the server which flow in which session it should run. This would mean that no new synchronisation system would have to be implemented but the downside is that it would create additional traffic not specified in the traffic model.

- *Synchronisation through centralization*: by using a central database, unit workers could register which flow and session it wants to run. When they later connect to a service, the service worker can look up the connectors IP in the database and know what traffic to generate. With this solution units would not have to know which server to connect to beforehand, enabling server side load balancing. However, this solution would add even more complexity to the system as we would add another step to the startup process which includes communication with an external component.

Our choice

Synchronisation is something that is often sought after in distributed systems, the problem is that it often comes at the cost of something else, be it time, complexity or limitations. The small impact of synchronising through an initialisation message, together with that it is a far less complex solution compared to the other alternatives, made it the option with the lowest cost in our eyes. The intended use case of the traffic generator is to study how
different traffic characteristics affect the network, this means that a bit of extra data at the start of each connection has very little affect on the end result. If exactly following the traffic model would have been more important however, then sending an initial message would be more complicated and the two other options may be more suitable.
Chapter 4

Evaluation

The goal of this thesis is to study methods to implement a traffic generator that is scalable and can handle complex traffic models. Due to the complexity of this task and the many problem areas, there are a wide variety of potential solutions that can be explored. Therefore, once the design had been finalized, we started looking at more implementation specific improvements regarding storage, parallelism and resource utilisation, to further increase the performance of the generator. In this chapter we present the different solutions that we explored.

4.1 Testing details

To make the results more comprehensible, we want to give a good understanding of the testing environment, the traffic models we used and the methods we used to gather the results.

4.1.1 Testing environment

The environment which we ran the tests on was two physical servers of type GEP4 which is manufactured by Ericsson. Each server used the x86-64 architecture with two Intel Xeon E5-2658, 48 GB DDR3, 500 GB SSD and two 10 Gb Ethernet controllers. The architecture specification are summarised in Table 4.1 and the CPU architecture is summarised in Table 4.2.

<table>
<thead>
<tr>
<th>CPU</th>
<th>2 x Intel Xeon E5-2658</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>48 GB DDR3</td>
</tr>
<tr>
<td>HDD</td>
<td>500 GB SSD</td>
</tr>
<tr>
<td>NIC</td>
<td>2 x 10 Gb Ethernet controllers</td>
</tr>
</tbody>
</table>

Table 4.1: System specification.
### 4.1. Testing Details

#### Table 4.2: Intel Xeon E5-2658 architecture.

<table>
<thead>
<tr>
<th>Cores</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>16</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.1 GHz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32 kB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 kB</td>
</tr>
<tr>
<td>LLC cache</td>
<td>20 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS</th>
<th>CentOS 7.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel foundation</td>
<td>Linux 3.10.0</td>
</tr>
<tr>
<td>Erlang OTP version</td>
<td>17.5</td>
</tr>
<tr>
<td>Python</td>
<td>2.7.5</td>
</tr>
<tr>
<td>Nmon version</td>
<td>14g</td>
</tr>
<tr>
<td>Nmon analyser</td>
<td>v4.2</td>
</tr>
</tbody>
</table>

Each server ran CentOS 7.1 Linux, Erlang OTP 17.5, Python 2.7.5 and Nmon 14g. The data we got from Nmon was parsed and analysed with Nmon analyser v4.2. The installed software is summarised in Table 4.3.

We chose to set up the environment as our intended architecture, illustrated in Figure 3.1, with one of the machines as the client side and one as the server side. With this set-up we ensured that there would be a physical medium that the traffic had to go through. The machines were connected to each other through a switch that had no other connections than from our machines. We ran the distributor and the database on the server side due to it being less loaded than the client side during runs.

#### 4.1.2 Testing method

Our focus with the testing was to see how the generator scaled and performed with an increasing number of units and connections. We wanted to gradually add units and take measurements when these had been created. After we had measured the performance of the system we added more units and redid the procedure. We continued this process until the generator started malfunctioning by not generating the specified traffic.

#### Nmon

To measure the performance of the system while running the application we used nmon. This tool allowed us to measure CPU, memory and network usage for the generator process into a comma separated file. This file could then be parsed by nmon analyser which created an Excel file with the data. This data is what we use for the results.

#### Probe

In addition to the system performance we wanted to get an idea of how our application performed internally. For this we used the probe which ran in our application with the same priority as the unit/service workers. However, one issue with this approach is that we would use the system to observe the system. This means that the observing could decrease the system’s
4.1. TESTING DETAILS

(a) With probe.

(b) Without probe.

Figure 4.1: CPU load with and without the probe running.

performance and affect the results. Therefore we wanted to ensure that the probe would not affect system performance in a meaningful way. We did this by running the system with and without the probe and comparing the resource usage. If the difference was small we could assume that the probe would not have a significant impact. Figure 4.1 shows the CPU-usage of the traffic generator on the client side with no units running. To the left results are with the probe running and to the results to the right is without the probe running. All results were obtained by measuring the performance of the system with a one second granularity for two minutes. The two graphs are nearly identical which shows that the probe hardly has any impact on the CPU-usage of the generator.

4.1.3 Traffic models

The traffic model determines what the generator will produce and as such plays a large part in how it will perform. It decides such things as how often a worker sends and receives data, how many workers that are created and how much data each unit sends. Therefore we used several different traffic models when testing the performance of the system. We started with a basic model that was easy to measure and then from that we derived a flow heavy model to test how the generator would react to more flows, and a communication heavy model to test how the generator would react to more intense communication in each worker. Each traffic model consists of only one super session. In the following we describe three synthetic and one realistic, UTM-based model, that we used for our evaluation.
Basic model
The basic model is defined as a single session with a single TCP flow. This flow sends 1000 bytes up and 1000 bytes down each second for 40 seconds before starting the session again.

Flow heavy model
The flow heavy model is defined as a single session with ten TCP flows. Each flow sends 100 bytes up and 100 bytes down each second for 40 seconds before starting the session again.

Communication heavy model
The communication heavy model is defined as a single session with a single TCP flow. This flow sends 100 bytes up and 100 bytes down every tenth of a second for 40 seconds before starting the session again. This is a case which is not possible in UTM due to the aggregation of packet headers into objects if they are within 500 ms of each other, described in Section 2.3.1. We included this traffic model anyway because we wanted to evaluate the performance of the traffic generator in this extreme case.

Realistic model
The realistic model is a traffic model which is based on real data gathered from a service provider. It contains multiple sessions running multiple flows with varying traffic. The multiple flows complicates testing as they are chosen at random during runtime which means that the traffic generated will be different every time a test is run. For this reason we have not used this model in the benchmarking. However, we used it when we tested the storage of the traffic model and the load balancing.

4.2 Benchmark
In this section we provide a benchmark of the system by testing it using different traffic models and number of running units. With this benchmark we can identify the strengths and the weaknesses of the system as well as confirm that it is functioning properly. To do this we used the three synthetic traffic models to identify how the generator behaved when exposed to different traffic characteristics.

4.2.1 Results
The results of the benchmarking were produced using a single instance of the generator on both the client side and the server side which were both running the standard Erlang configuration.
Throughput

By looking at the throughput we can get a sense of how the generator is performing. However the values are higher than the model specifies due to overhead caused by the TCP protocol. But despite this, the throughput should still scale linearly with the amount of connections if the generator is functioning correctly. In Figure 4.2 the average read and write data over the network is shown as a function of the number of connections. We can see that the throughput scales linearly with the amount of units. We also see that the ratio of write to read is higher with traffic models that sends more packets per unit, which increases the TCP overhead. With all models we see that the throughput stops increasing linearly when the generator reaches its limit. With the communication heavy and flow heavy models we also see a big decrease in throughput when the limit is reached. We believe that this is caused by an overload of I/O-operations in the kernel which causes it to neglect actually sending and receiving the packets. Provided that this is indeed the issue, optimizing the operating system and the generator’s send path could enable us to push the generator further.

CPU load and memory usage

The CPU and the memory are two of the most important resources of the system and going over the capacity of any of them can lead to significant performance degradations. Therefore it is very important to ensure that this does not happen. Figure 4.3 shows the average CPU usage as a function of the number of connections. In it we can see that the CPU usage increases rapidly at the start before flattening out at 50-60% after which it drastically decreases when the generator reaches its limit. With the communication heavy traffic model however the CPU usage is more unstable than the other two. We also noticed that the memory usage scaled linearly with the amount of units at around 4-6 MB per unit which we feel should not cause any problems at all. We feel that the drop in CPU usage is further evidence towards that the problem lies in the kernel being overloaded and not that the generator application is malfunctioning.

Probe response time

A key attribute of the probe is that it has a scheduling priority that is comparable to a worker in the application. We use this to gauge how well the workers are performing. One of the more important aspects of the workers performance is that they are allowed to run when they should. To measure this the probe goes to sleep at regular intervals and when it wakes up it compares the time it woke up to the time it should have woken up to measure its response time. Figure 4.4 shows the maximum, average and minimum response time of the probe as a function of the number of connections. We can see that under normal conditions the response time of the probe is
4.2. Benchmark

(a) Basic traffic model.

(b) Flow heavy traffic model.

(c) Communication heavy traffic model.

Figure 4.2: Throughput of the generator.
4.2. BENCHMARK  

CHAPTER 4. EVALUATION

(a) Basic traffic model.

(b) Flow heavy traffic model.

(c) Communication heavy traffic model.

Figure 4.3: CPU usage of the generator.
4.2. BENCHMARK CHAPTER 4. EVALUATION

(a) Basic traffic model.

(b) Flow heavy traffic model.

(c) Communication heavy traffic model.

Figure 4.4: Probe response time of the generator.
normally between 0.8 and 1 ms regardless of traffic model. The results show some irregular spikes but the average is stable regardless of the amount of connections unless the system is close to breaking down. This can be seen in all traffic models where the average probe response time drastically increases by several orders of magnitude at the breaking point. The sudden rise in response time is something that we feel can be connected to the CPU usage graph, as less resources are spent on the generator application, the worse it performs. This means that the accuracy of the output as well as the throughput of the traffic generated goes down.

4.3 Erlang tweaks

Erlang offers a lot of customization of the virtual machine through flags given at startup. With these flags the virtual machine can be optimized to better fit a specific application. We found this interesting and wanted to examine how much of a performance impact it could have.

4.3.1 Configuration

There are endless combinations in which you can set the flags for the Erlang virtual machine, for this thesis we selected a couple of what we thought were the most interesting ones and tried to find the best configuration for those flags. We then compared it to the default configuration to see how much of a difference the flags make. Table 4.4 summarises both the default configuration and the tweaked configuration.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Tweaked</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager check I/O scheduling</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>Kernel poll</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>Number of asynchronous I/O-threads</td>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>Scheduler port parallelism</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>Scheduler utilization balancing</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>Scheduler wakeup threshold</td>
<td>Very_low</td>
<td>Medium</td>
</tr>
<tr>
<td>Scheduler wake cleanup threshold</td>
<td>Very_eager</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 4.4: Summation of the default configuration and the tweaked configuration of the Erlang virtual machine.

4.3.2 Results

The results of the Erlang tweaks were produced using a single instance of the generator on both the client side and the server side which were both running the Erlang configuration mentioned above.
4.3. ERLANG TWEAKS


Figure 4.5: CPU-load while running the basic traffic model with Erlang tweaks.

Figure 4.6: Probe response time while running the basic traffic model with Erlang tweaks.

**CPU-load**

Since the configuration had mostly scheduler flags, we thought that CPU was where we most likely would see the biggest improvement. Figure 4.5 shows the average CPU usage as a function of the number of connections. In it we can see that the CPU usage increases with the amount of connections up to 65 % at 25 000 connections after which it starts to decrease to around 50 % before flattening out. This can be compared to the benchmark where the CPU usage was at 58 % at 25 000 connections.

**Probe response time**

Another interesting metric to look at for the Erlang tweaks is the probe response time as it is highly dependant on the Erlang virtual machine. Fig-
Figure 4.6 shows the maximum, average and minimum probe response time as a function of the number of connections. In it we can see that the average probe response time is between 0.9 and 1 ms until it starts rapidly increasing at 45 000 connections, reaching an average response time of 1.6 ms at 65 000 connections before the system reaches its limit. This is very comparable to the default settings which had similar performance in terms of probe response time.

4.4 Traffic model storage

The traffic model is a vital part of our traffic generator and replay engines in general, as described in Sections 2.2 and 2.3. Therefore it is important that it is stored in a way that is efficient and makes it easily accessible.

4.4.1 Solutions

There are many ways to store information and we can not test them all. Therefore we chose two very different methods to get a grasp on how much of a difference it would make.

Memory storage

One of the solutions we tested was to load the entire traffic model into the memory at startup and then read from it directly when running the generator. This was done with Erlang’s ets module which enabled us to store data in tables in memory for easy access from any worker.

MySQL-database storage

The other option we tested was to store the traffic model in a simple MySQL database which contained one table per session. This meant that when a worker wanted to access the traffic model it had to go through a driver called Emysql [3] which would forward the query to the MySQL database.

4.4.2 Results

To test the performance of the two different storage solutions we measured the time it took to retrieve data from the traffic model as well as the memory usage with an increasing number of units. For this test we used the realistic traffic model as it provides more realistic results.

Access time

Minimising the time it takes for a worker to get the data requested from the traffic model is a key part to consider when choosing how to store it. We tested this by using the probe to measure the time it took to retrieve
data like a normal worker would. What we saw was that when we stored the model in the memory the probe had an access time that was shorter than 50 µs at all times. Figure 4.7 shows the median access time as a function of the number of units when using an SQL database. We can see that the median access time was steady at 2-3 ms up to 240 units before it started to increase with the amount of units. At 400 units the median access time had increased to about 37 ms. The median access time when using an SQL database was constantly at least 40 times worse than the access time from memory. We chose to present the result as median due to spikes in access time that were as long as five seconds. While these are worth noting and should be taken seriously, it skewed the results in a way that we felt portrayed its actual performance unfairly.

Memory usage

Another aspect of choosing how to store the traffic model is the memory consumption. If it takes up too much memory then the traffic generator cannot perform to its full potential. To test this we compared the memory usage before and after loading the traffic model into memory. The traffic model we used was from realistic data and contained one super session consisting of 1905 sessions and with a total of 370 000 objects. This increased the memory usage by 130 MB which indicates that our test environment would be able to handle more and a lot bigger models.

4.5 Load Balancing

One of the core components in distributed computing is load balancing. (You can read more about load balancing in Section 2.4.) In our traffic generator
we have two separate load balancers, one for the client side and one for
the server side. The client load balancer is implemented in the distributor
and decides which node to create a unit on. The server load balancer is an
IP-forwarder as explained in Section 3.2.1. The one we decided to test was
the load balancer in the distributor. We did it because we felt that testing
the load balancing of the units would provide more worthwhile information
as they are something that we have created.

4.5.1 Solutions

For testing this we chose the following load balancing algorithms:

- Random Server Selection (RSS)
- Round Robin (RR)
- Least Units (LU)
- Least Loaded (LL)

Three of these (RSS, RR and LL) were defined in Section 2.4. The fourth
(LU) is a slight modification of the LC algorithm that was defined in Sec-
tion 2.4. Instead of prioritising nodes based on their number of connections,
LU prioritises nodes based on the number of units on them. We felt that
this would be a more appropriate algorithm for our application since each
unit is persistent and has a varying amount of connections. We also chose to
define load as the average CPU usage the last minute in the LL algorithm.

4.5.2 Results

In order to test the load balancing algorithm we would need several client
nodes, this was done by splitting our 16-core machine into eight virtual ma-
chines, each with two cores. For the load balancing algorithms, we felt that
just creating a number of units and then measuring the performance would
lead to many algorithms performing the same. Therefore we implemented
an algorithm where we would repeat the following process:

1. Create six units,
2. start five random units,
3. stop two random units, and
4. delete one random unit.

This process was repeated 160 times, resulting in 800 units of which 480
were running. With this algorithm the strengths and weaknesses of the load
balancing algorithms would be more prevalent and provide a better result.
We chose to use the realistic traffic model for this test to provide a more
dynamic environment for the load balancing algorithms to deal with.
The purpose of a load balancing algorithm is to distribute the workload so that you get the maximum performance out of the system. We chose to measure the CPU usage as we felt that it would give the best view of how loaded the system was. Figure 4.8 shows the CPU usage of the virtual machines for different load balancing algorithms. We can see that RSS, RR and LU performed comparatively with the maximum relative difference in CPU usage between the virtual machines being 1.67 times for RSS, 2.24 times for RR and 2.31 times for LU. With LL however it was 5.53 times which is significantly worse. We can see that the total load of the virtual machines in the tests is different. This is caused by the realistic traffic model as it makes each unit generate different amounts of load. We believe that the reason for the poor performance of the LL algorithm is caused by the realistic traffic model. Most sessions in it are rather long and have slow starts which means that it takes a long time before the load changes after a unit has been created. This means that it takes a while until the load is increased after a unit has been created.
Chapter 5

Discussion

For the most part, our evaluation of the system provided us with the information that we had hoped for. We chose Erlang because of its scalability properties which showed in the results, we saw that the CPU and the memory were not the limiting resources of the generator. This points towards that we have implemented a traffic generator which uses its resources well. The evaluation was not without its fair share of surprises though. For example, we did not expect how much of a difference the Erlang tweaks would make and we were surprised by the sudden and seemingly random spikes in performance that occurred during our test runs.

5.1 Benchmark

The benchmarking showed us that we had implemented a traffic generator that possessed the scalability characteristics that we were designing for. This was shown through the linear increase of memory usage and the way that the CPU usage scaled with the amount on connections. Even though the scaling was very poor at low amount of connections it quickly improved once a certain threshold was reached. The probe response time was also very promising as it was not affected by the amount of connections during normal workload and was well within our specifications.

However there were some concerning things with the results of the benchmarking as well. The main issue was the fact that even though we were nowhere near using all of our memory or CPU capacity the generator still started malfunctioning after a certain amount of connections. In fact the CPU usage actually decreased which we believe is related to an overload of I/O-operations in the kernel due to a couple of things. First of we noticed that the ssh session started being unresponsive at the same time that the generator started malfunctioning. Secondly we can see that when running the basic and the flow heavy traffic model, the generator breaks down at
around 30,000 connections. Since the two models have roughly the same amount of I/O-operations at this point it seems likely that there is a correlation there. We have also run tests with a more data heavy traffic model with which we were able to utilise the full capacity of the network interface with ease. This leads us to believe that the amount of data transferred is not the issue when running the traffic generator with the other traffic models.

Another concern is how rough the scaling of CPU usage was when running the communication heavy traffic model compared to the other two traffic models. While each unit has the same amount of connections as the basic traffic model, the generator breaks down much earlier while running it. It seems like the generator is not as adapt with fast transmitting traffic models which could be related to the response time of the probe being in the range of 8.0 to 9.0 ms. As explained previously in Section 4.1.3 this traffic model cannot be created with UTM so this is not as big of an issue.

Finally the probe response time typically showed a variance of 0.2 ms however the difference between the maximum and minimum value went up to 1 ms at one point. For our specification this result is acceptable, however in a situation where more emphasis is put on the accuracy of the output this kind of variance would likely not be acceptable. We believe that this is a result of the nature of Erlang and the fact that it is designed for soft real time systems. Especially since the average response time barely changed with the amount of workers and that the CPU was never at maximum capacity.

5.2 Erlang tweaks

The results that we got with the Erlang tweaks were far beyond our expectations. We could push the generator to 50,000 connections while keeping the average probe response time shorter than one millisecond. All of this with a decreasing CPU usage after 25,000 connections. Even though the probe response time increased, we could still push the number of connections to 65,000 with a linear increase of the throughput. This is a substantial increase compared to the default settings, for which we could only reach 27,500 connections while running the basic traffic model before the generator started malfunctioning. We believe that the main reason for this improvement is the kernel polling and the number of asynchronous I/O-threads as they affected how we handled I/O-operations which we feel was the biggest issue with the default configuration.

The tweaks further emphasised the characteristics of Erlang as a soft real time system. While it allowed us to double the amount of connections we could run, at a lower CPU usage, it was done at the cost of probe response time. We could see that the average probe response time started climbing with increasing speed as we got past 40,000 connections. We could also
see that the CPU usage was higher much earlier while using the tweaks, for example the CPU usage was doubled at 5000 connections when using the tweaks compared to without. Presumably this was due to the eager settings for the scheduler making it more active early on. However as it was early on it did not matter and it paid off in the long run. After 25 000 connections the CPU usage started dropping and had decreased with over 10 percentage points as we got to 45 000 connections.

Our results clearly show that there is much to gain from tweaking Erlang to better fit the running application. As described in Section 4.3.1 we only selected the flags that we found most interesting so there is potential for even better performance. However it is worth noting that the flags should be chosen on an application to application basis as they do not provide strictly better performance but rather a trade off.

### 5.3 Traffic model storage

As seen in the results we got the best performance with the traffic model in memory. This was what we expected but for this to work, it requires that there is enough memory on the machine running the generator. We tested a database as a different solution which had a median access time of 2-3 ms, 40 times higher than the in-memory solution. However, we feel that is more of an indication that the in-memory solution is fast, rather than that the database is slow. Another problem with the database that we thought of but did not encounter was that the amount of I/O-operations was what limited the generator. With the database solution we have to contact the database and receive the data which adds more I/O-operations. This way we may not be able to push the system as far as we could with the in-memory storage.

The database solution does have its advantages however. It is a solution which gives a central storage for the traffic model. To update, add or delete a traffic model you would only need to do this in one place. Unfortunately the database can also be the bottleneck since all generator nodes would request from it. As the generator is a soft real-time system it needs the traffic model data in a reasonably timely manner. In the results we can see that the median access time is around 2 ms, which does not include any network latency. We feel that there could be a problem here, especially with the spikes in maximum access time that we got. To compensate for the access time with a database some pre-fetching could be a way to match the timing better. The database would also need to handle a large amount of concurrent requests while keeping the response time low for this to work.

We think that a combination of the two solutions would be an good approach. What we would want is the centralisation of the data that we get from the database solution together with the access time of the memory
storage. With this we want to make as few requests from the database as possible. Our solution for this is to cache as much of the traffic model as possible in the nodes following a cache algorithm. With enough memory in a node it could store the entire traffic model and the storage method could then be seen as in-memory storage. An additional improvement here could be to use network caching between the database and the node to further minimise the amount of requests.

Another approach is that since the traffic model can consist of several super sessions and a unit only runs one of these, nodes could choose to only store some of the super sessions in memory. It would then only receive units that would run these super sessions. This would however require that the distributor knows what super sessions that the nodes can run.

5.4 Load balancing

To be able to fully utilize the option of running the traffic generator distributed over several machines, a good load balancing algorithm is crucial. For the most part, the algorithms performed similarly, with the big outlier being LL which was significantly worse. We believe that only looking at the load when choosing which node to assign a unit could be misleading as each unit provides a varying amount of load. Even if the node with the least units should have the least load on average, there is a high amount of randomness regarding the current sessions being run and which part of the session that is currently being generated. The problem is magnified due to the load not being immediately updated when a new unit is created, leading to potentially multiple units in a row being assigned to the same node. If the units would have been created at a slower speed we believe that LL would perform better.

RSS, RR and LU showed similar performance, while RSS had the best result we believe that it was caused by randomness and not by RSS being the superior load balancing algorithm. We feel that LU should be the best load balancing algorithm out of the ones we tested if all units are running the same super session. This should be the case because in long run each unit should generate the same load since they are all running the same super session. If we had used the basic traffic model instead, the load should have been equal on all nodes. A problem with it though is if there are a lot of units which are created but aren’t active, then the algorithm would not balance properly. However one could easily modify the algorithm to only consider active units in order to deal with this situation.

We feel that the load balancing of units can be compared to the load balancing of virtual machines in the cloud. These machines can have varying load that can change with time which the cloud has to handle. Mukherjee et al. [22] uses a probe that is similar to ours to check the condition of the virtual
machines. When the probe indicates a bad condition virtual machines are moved to another host. We think that a lot can be learned from this topic and that it is worth investigating further.

5.5 Traffic data privacy

In this thesis we have used both synthetic and data driven traffic models. Private information of real users is a sensitive subject due to the potential of privacy violations. Because of this we want to clarify that none of these models used here include any private information of real users. In the case of data models obtained from real network traffic, IP addresses have been anonymised using a hash function. This allowed us to differentiate the sources of the data, while ensuring that there is no way to trace it back to its user.
Chapter 6

Conclusion

In this thesis we presented and evaluated the design and implementation of a traffic generator that can simulate the traffic of tens of thousands of network devices from a given traffic model. It can simulate the traffic of over 65,000 concurrent units that have a traffic model where each unit sends and receives one kB of data each second. This only results in 1 Gb/s but if a more data heavy traffic model is chosen, the generator can maximize the bandwidth of 10 Gb/s. We have found that Erlang is well suited for simulating tens of thousands of concurrent devices, provided that it is a soft real-time system. We have also found that it can utilize the available resources of a machine well, especially when tweaking the virtual machine towards the application. We have found that by using independent units as the generators of traffic, the system can easily scale over several machines. Finally we have found that if storing the traffic model in memory is an option then it is clearly the best one.

6.1 Future work

We began this thesis with stating that it is meant to serve as a baseline for the development of future traffic generators, so naturally there is much more that can be done, not only on this traffic generator but for traffic generators in general. We are confident that if we were to continue working on our traffic generator, we would be able to push the amount of concurrently simulated devices even higher. Looking at the scaling of the CPU and memory usage we feel that if the problem with an overload of I/O-operations was solved then we would be able to push the system much further. Because of this issue one could consider another approach. Instead of trying to maximise the output of a single node, one could try to scale over several machines instead. Doing this would lead to a higher capacity of I/O-operations than compared to a single node. If this approach was taken, more work could be put into the load balancing aspect of the system as it is rather primitive in
its current form. By allowing dynamic migration of units between nodes, one could deal with the varying load that comes from running random sessions. Content aware load balancing could also be implemented by analysing the units assigned traffic model and balance with that in mind. Other interesting future work could include performance tuning of the multicore machines themselves such as to maximize their utilisation as has been done in related contexts [14].

In order to simulate the traffic of multiple concurrent devices, we represented them as independent units in our generator. A feature of this design which we feel could be used for great effect is the ability to get QoS measurements from the units. This would make it possible to gauge the user experience on a user by user basis while running the simulation which is something we feel could provide very valuable data. You would easily be able to determine the performance of the network from a user standpoint and see which users have the best and worse experience.

One of the compromises we had to make was the accuracy of the traffic generated. By sending the initialization message we do not perfectly recreate the provided traffic model but rather the characteristics of it. To achieve perfect recreation, major changes would have to be made and probably a change of programming language as well as Erlang is not suited for hard real time systems. Ultimately, you could go on forever, improving the performance and adding new features, as is the case with all software.
Bibliography


Appendix A

Generator design

Figure A.1 shows the entire design of the generator application. At a high level the Generator_App supervises the underlying components. These are either supervisors or workers which are represented by squares and circles. A square represents a supervisor and a circle represents a worker. The components of the design are further described in Section 3.1.
Figure A.1: The architecture of the generator application.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtonda ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet – or its possible replacement – for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

©Björn Bylund, Nicklas Blomqvist