A look at how a network map is affecting resources of the nodes it is mapping

Using the Emerald programming language

Thomas Kristiansen

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

A look at how a network map is affecting resources of the nodes it is mapping

Using the Emerald programming language

Thomas Kristiansen

Abstract

In this thesis, I have written a program in the Emerald programming language, that maps out the internet nodes of the network where the Emerald program is running. The program replicates itself onto all available nodes and then monitors the status of these nodes. The monitoring includes information about which of nodes that are up, and what the round trip delays are between the nodes. This information is periodically updated so that the map is as accurate as possible at all times.

This thesis is an empirical thesis. The program will be tested on a network of nodes called PlanetLab. PlanetLab lets me run the program on nodes all over the world, and evaluate the program while it is running. This gives me a look at how the program is running on a large network, with nodes in a realistic environment, and this will be the basis of the thesis. I will, among other things, look at how the program affects the network, and how the program affects the performance of the nodes them selves.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, I have written a program in the Emerald programming language, that maps out the internet nodes of the network where the Emerald program is running. The program replicates itself onto all available nodes and then monitors the status of these nodes. The monitoring includes information about which of nodes that are up, and what the round trip delays are between the nodes. This information is periodically updated so that the map is as accurate as possible at all times.

This thesis is an empirical thesis. The program will be tested on a network of nodes called PlanetLab. PlanetLab lets me run the program on nodes all over the world, and evaluate the program while it is running. This gives me a look at how the program is running on a large network, with nodes in a realistic environment, and this will be the basis of the thesis. I will, among other things, look at how the program affects the network, and how the program affects the performance of the nodes them selves.

1.1 Research questions

In this thesis I am looking at the following to research questions, regarding the network I have created:

1. How does it affect the bandwidth of the nodes?
2. How does it affect the CPU and memory usage of the nodes?

1.2 Research methodology

To test the program in a realistic environment, I am running and testing the program on the PlanetLab network. I will run the program on different nodes in the network, and pick some of the nodes to do network, CPU and memory monitoring on. I will then evaluate and discuss this data.

Chapter 2

Background

2.1 The Emerald programming language

The Emerald programming language is an object-oriented programming language, developed with goal of simplifying the construction of distributed applications. [25]

An example program, written in Emerald, called "Kilroy was here" can be seen in Listing 2.1.

Listing 2.1: Emerald program - 'Kilroy was here' [25]

```
const Kilroy <- object Kilroy
  process
    const origin <- locate self
    const up <- origin.getActiveNodes
    for e in up
       const there <- e.getTheNode
       move self to there
    end for
    move self to origin
  end process
end Kilroy
```

2.2 PlanetLab

PlanetLab is a global network of Nodes spread out all over the world. The network is meant for research, and supports development of network services. The network started back in 2003, and since its beginning more than 1,000 researchers at different academic institutions and industrial research labs, from all over the world, has used its services to develop new technologies. These technologies has been developed for distributed storage, network mapping, peer-to-peer systems, distributed hash tables and query processing. [29]

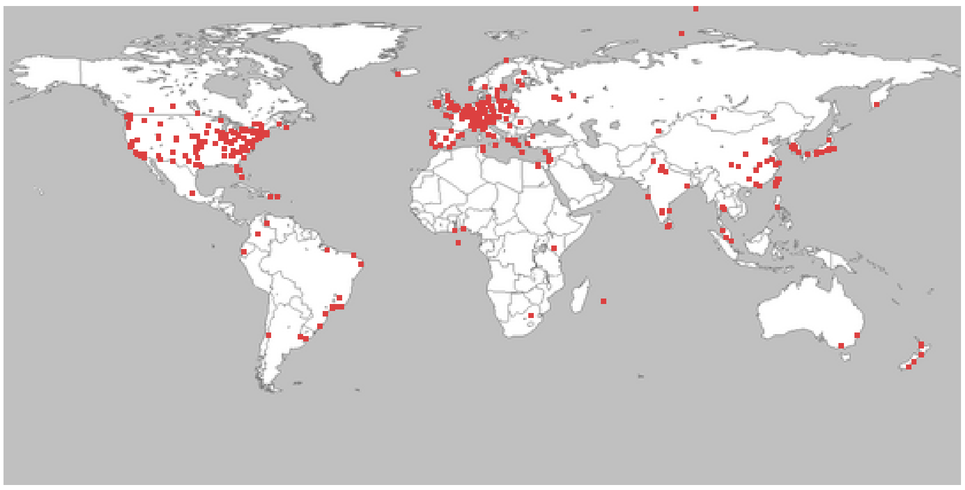


Figure 2.1: A map of active PlanetLab Nodes [29]

2.2.1 The community

PlanetLab is kind of like a community, where academic institutions and research labs all over the world share resources to keep a live a network that they all can use for research of new technologies. As of today, PlanetLab exists of 1353 nodes [29] placed on computers all around the planet. Figure 2.1, on page 4, shows a map of the world with active PlanetLab Nodes marked with red dots. This might not be the current map of Nodes, as PlanetLabs homepage [29] doesn't state if it is up to date or not, but it gives you a picture of how this network is put together.

For institutions or research labs to join the PlanetLab community, and take use of its network, they have to apply for a membership. I am not going to go in to detail about how the membership agreements work, except for one part of it. That part is the fact that you have to maintain at least one site to be able to be a member (as long as you don't have a special agreement). On this site, you have to host at least two PlanetLab Nodes. [28] This means that if someone wants to join the community, they also have to contribute to it. That is one of the great aspects of PlanetLab, because as the number of members grow, the number of available Nodes grows as well, meaning everyone benefits from more institutions and research labs joining.

2.3 Distance Metrics

When we are talking about distance in the Internet, we look at distances between two or more hosts in a network. If we were to measure the distance between two hosts, the metric we choose to use is dependent on what we are going to use the result for. There are many different metrics that could be used, and I have listed, explained and discussed the uses of some of them in this chapter [15].

2.3.1 The Metrics

Round-trip time

The round-trip time (hereafter referred to as RTT) is the time a packet uses from its source host to the destination host, and then back to the source again. When measuring the RTT between to hosts in a network, the result will never be exactly the same. Therefore, measuring the RTT only once is almost never sufficient. In regards to this, there exists a few methods of measuring, that helps getting a result that is as accurate as possible. For example, calculating the average of a set of RTT measurements, using the median of these measurements, or just using the last measured RTT.

There will always be some factors influencing the measurements that are hard to pick up on. These factors are often random and short-lived, like network congestion, meaning they can show up at any time. Because of that, some of the methods of measuring have to be adjusted accordingly. If we, as an example, use the average of a set of RTT measurements, there could be some trips that takes much longer time than other ones. If we calculate the average with these longer trips as part of the set, the results will be misleading. It is therefore important to exclude measurements that are varying hugely from the rest.

IP path length

The IP path length is the number of hops, or routers, a packet has to traverse in its path from the source host to the destination host [15]. It does not take into account other sources of delay like network congestion, but if you are measuring the RTT on a network where the biggest source of delay is how long it takes routers to process packets, then this metric is useful.

Autonomous System path length

Autonomous Systems (hereafter referred to as AS) are groups of one or more networks where all the networks share a common routing protocol. These are called the internal routing protocols for the group. Each AS then interact with each other using a common external routing protocol called BGP (Border Gateway Protocol), which is the standardized external routing protocol used in the Internet. In the Internet, these ASs could for example be Internet Service Providers (ISPs). Each AS also has its own unique number that identifies it. [7, 15]

The AS path length metric works the same as the IP path length, just that instead of counting the number of routers a packet has visited, we now count the number of ASs it has visited. To determine the number of ASs visited by a packet, we could use the BGP routing tables. This metric does of course not take in to count the delays happening inside each AS, so it is normally used when the biggest source of delay is assumed to be because of the congestion caused by public traffic in the external network (the Internet). That is also why BGP uses AS path length as its primary metric when it comes to route selection. [7]

Geographical distance

When we are talking about geographical distance in the Internet, we mean the length of the great circle arc connecting the source host and destination host on the Earths surface. Of course, the structure of the Internet does not entirely match the layout of the earths surface, but if it did match and all the routers a packet had to traverse was on the great circle arc connecting the source and destination, then the packets would approximately follow the shortest geological path between the source and destination hosts. [15] Even though the route a packet would follow doesn't match the great circle arc, geographical distance is still an important and well used metric when it comes to distance in the Internet.

2.4 Web Crawlers

2.4.1 The Concept

The Internet is a huge and ever growing web of information. This information takes form in web pages, pictures, videos, audio, advertisements, books, and every other kind of data you could imagine. Navigating all of this data is, as you can imagine, a big and complex job. This is where the web crawlers come in. Web crawlers are programs used by websites, like Google or Yahoo, to identify and index web pages all over the Internet and storing the information in a database. By doing so, they make it easier and faster to search for web pages that contain certain words or pages that are of a specific theme. People can then query these databases and, to a certain extent, navigate the Internet. These queries do of course return thousands upon thousands of web pages per search, so the order these pages are returned in to the user, is also important. This means that the different search engines have ways to implement how people see and experience the Internet, which could be both a good and a bad thing.

2.4.2 How they work

As mentioned in the previous section, web crawlers are programs used by websites, mostly search engines, to make it possible to search the Internet in a fast and easy way. These websites periodically sends these programs out to all the web pages they manage to identify, and the crawlers download these web pages. The programs then scans these pages for information that later can be used as indexes to describe them. The algorithm the programs use for this process varies from program to program, but it is usually based on finding key words and phrases in the web pages that are useful for identifying the current website they are searching through. The data collected from the crawlers are then combined with the URL of the web page they are indexing, and then stored in the search engines database. It is this database that is queried when people do searches in the search engine. The data returned to the user is a list of the URLs that are connected to the indexes that matches the search. [20]

2.4.3 Challenges

In the early days of web crawlers, they just searched the Internet as I described in the last section. There where no requirements or guidelines to how they should do the indexing, and the design of the crawler where pretty basic compared to the web crawlers that exists today. As every invention, the web crawlers have evolved and adapted to challenges throughout the years. One of the earliest issues did not affect the crawlers directly, but instead influenced the web pages visited by them. As mentioned earlier, web crawlers crawl by requesting URL links for web pages. This could mean that a crawler could end up sending a lot of request to one domain holding a lot of web pages. If a domain receive to many requests in a short amount of time, it can affect the performance of the server holding all the web pages. A worst case scenario would be the server crashing, or at least the server experiencing reduced available processing power. To solve this problem, a concept called politeness was introduced. Politeness is to basically restrict the number of request a web crawler sends to a server per a specific unit of time.

There has also been a couple of issues that has affected the web crawlers directly. One of them is so called traps that are directed specifically at web crawlers. These traps are websites with huge amount of pages full of nonsense data, that are meant to basically waist the time of crawlers. The solution to this problem was to introduce black-lists in to the implementation of the web crawlers. These black-lists are filled with URLs to those types of traps, and the crawlers then skip the web pages it encounters if they exists in this black-list. The lists are of course also updated as new trap pages are discovered.

Another challenge that has been affecting the crawlers directly is the exponential growth of websites on the Internet. As you can see in Figure 2.2, on page 8, the number of registered websites has increased from about 17 Million to about 1,6 Billion over the last 18 years. This might be the biggest challenge the developers of web crawlers has had to deal with over the years, because there is no exact solution to the problem. The Internet is always going to grow, and the crawlers just has to adapt, finding new and more efficient ways to crawl through the increasing amount of web pages. [21]

2.4.4 The evolution of crawlers

As the Internet has expanded and evolved over the years, the web crawlers has had to evolve with it. In the beginning, there existed

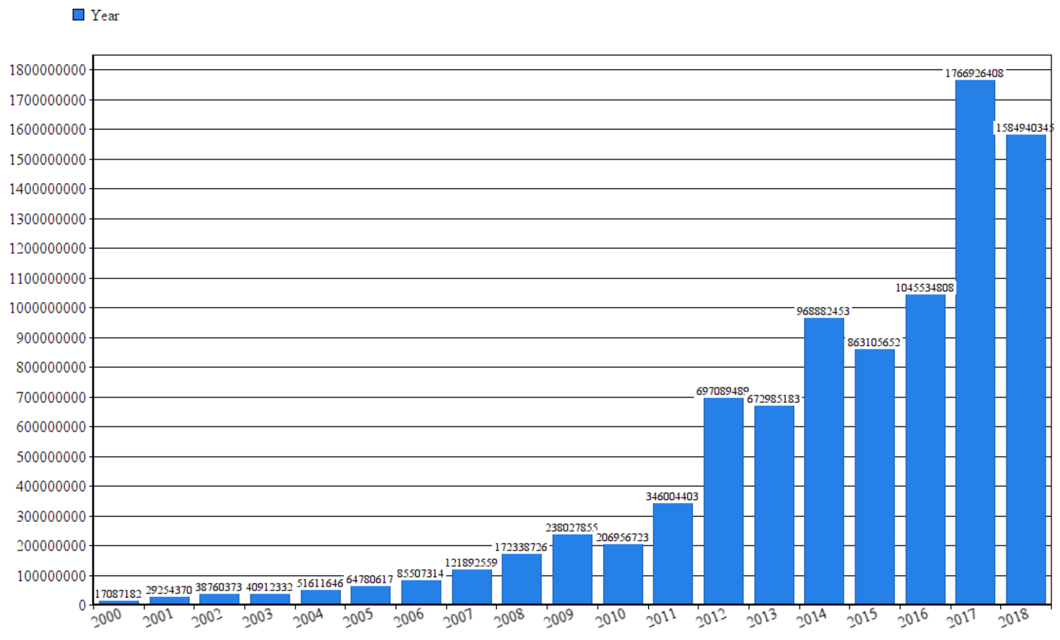


Figure 2.2: Number of websites on the Internet from 2000 to April 2018. Source: NetCraft Web Server Survey [18]

mostly static web pages, meaning the web crawlers could download the pages and basically index them as they where. After a while, more complex web pages began to arise, meaning the crawlers had to be evolved and adapted to these changes. In this section I am going to briefly explain the three main stages of the evolution, and how the crawlers had to adapt to them.

Traditional web crawlers

The traditional web crawlers are the once that was first created. The first generation of web crawlers, back in 1993, basically worked as explained in section 2.4.2. They where sent out to all websites they managed to identify, downloaded the web pages, analyzed and indexed important and relevant information before they stored that information in a database together with the websites URL. And that was pretty much all they did. About one year later, a more advances version of web crawlers came to life. What made them different from the first crawlers was that they addressed two of the challenges I talked about in section 2.4.3, namely the constant request to domains and the web crawler traps. The introduction of politeness and black-lists was already a huge improvement to the web crawlers, even though it happened only one year into to its existence. The first commercial web crawlers where also of this new type of crawlers. [20, 21]

A few years later, in 1998, the web crawler called Google came in to life. This web crawler was created by Sergey Brin and Lawrence Page [6], and was created to address the problems regarding scalability. The first thing they did to address this problem was to reduce the time used on disc access. To accomplish this, they, amongst other things, introduced indexing and compression of the repository. This means that they compressed each web page before storing it in the repository. Doing this, the repository used a lot less disk space per web page, meaning it could store a lot more pages. The indexer then read the repository, uncompressed the documents and finally parsed them. I am not going to get into how the indexer parses the data, but you can read more about that in the article "The Anatomy of a Large-Scale Hypertextual Web Search Engine" [6] written by Brin and Page. Another thing they did to handle scalability was the introduction of an algorithm they called "PageRank". The PageRank is created to visualize an average user on the Internet, who are given a web page at random, that keeps clicking on links until he gets bored and starts on another random web page. A page then gets a PageRank based on the probability that this average user has visited the page. To calculate the actions of this average user, the algorithm takes in to account the number of links pointing to the web page, as well as the style of those links. The crawler then uses the PageRank to decide how often it visits a site. By doing so, it uses less resources on unattractive sites, meaning the more popular sites would be visited more often. [6, 21]

One year later, in 1999, Allan Haydon and Marc Najork [14] introduced a web crawler called Mercator. This crawler was originally created to handle the problem of extendability, meaning it was created to take future growth into consideration. To handle this future growth, the crawler was created to support outside plug-ins, using a Java-based framework. This made it possible for people to add additional functionality to the crawler, as well as it made it possible to create different versions of most of the crawlers major components and then configure the crawler to use these instead of the original ones. Mercator was also created with the scalability problem in mind, by trying to solve the "URL-Seen" problem. The URL-Seen problem is basically the problem regarding that a crawler will encounter links multiple times. It might seem like a pretty easy problem to fix, but when the list of visited URLs gets big, the action to check whether or not at URL has been visited before becomes very time consuming. To handle the URL-Seen problem, Mercator stored hashes of discovered URLs in memory. They then put a limit to how big this list could be, and when the list reached the limit, they compared it to the URLs stored on disk. Finally, the list on the disk was updated. [14, 21]

In 2001, IBM introduced a web crawler called "WebFountain" [10]. The special thing about this web crawler was that it wasn't created just to index the Internet, but also to create a local copy of it. The local copy of a page was kept on local storage, and was updated every time WebFountain visited the web page. The web crawler was also fully distributed, meaning that the responsibility for scheduling, fetching, parsing and storing was all distributed between a cluster of machines. This, and some other futures of the crawler, made the web crawler scalable, and also kept the freshness of its stored web pages at a high level. [10, 21]

In the later days of the traditional web crawlers, there was a few crawlers that focused on the URL-Seen problem. In 2002, the web crawler "Polybot" was introduced by Vladislav Shkapenyuk and Torsten Suel [31]. This crawler further developed the technique used by the Mercator to handle the URL-Seen problem. Instead of storing hashes of discovered URLs, the Polybot used a Red-Black tree [24] to store the URLs in memory. When the tree had grown above a certain limit, the tree would be merged with a sorted list of URLs that was stored on local storage. Another web crawler that dealt with the URL-seen problem was the "UbiCrawler", created by Boldi, Codenotti, Santini and Vigna [5], which came out the same year as the Polybot. This crawler had a bit of a different approach to the URL-Seen problem. UbiCrawler used a peer-to-peer method, where it distributed the different URLs to all the available web crawler Nodes. Each URL was always assigned to a Node, but to that Node only, to avoid unnecessary data replication. There was no central place that calculated whether or not a URL had been seen before, like in PolyBot and Mercator. Instead, a URL was passed to a Node that was responsible to check if the URL had been seen before or not. This Node was decided by taking the hash of the URL, and then mapping it to the list of Nodes. By doing it like this, there was no communication needed between the Nodes to find the Node that should be responsible for a URL, as every Node was capable of finding it out by them selves. This again increased the efficiency of the process. Several other web crawlers that came to life in this time period also used the peer-to-peer architecture, and then added factors, for example the servers geographical position, to make it more efficient. [5, 21, 31]

In 2008, another web crawler came to life. This crawler was called "IRLbot", and was created by Lee, Leonard, Wang and Loguinov [16]. The IRLbot also addressed the URL-Seen problem, but in a bit of a different way than the other ones I have talked about in this section. The way the IRLbot did it was that it used a framework called "Disk Repository with Update Management", or DRUM for short. The purpose of this framework was to be able to store large collections of <key, value> pairs. The key in these pairs where a hash of some data, that worked as a unique identifier, and the value in these pairs where some information that was connected to the key. These pairs had three supported operations: 'check', 'update' and 'check + update'. The 'check' operation was used when the incoming data-set had a key that needed to be checked against the ones that was stored on the disk cache, to determine if the key was unique or a duplicate. The 'update' operation was used when the incoming data-set contained <key, value> pairs that needed to be merged with the disk cache. The last operation, 'check + update', did both the 'check' and 'update' operation in one go. DRUM worked by segmenting the disk into so called "disk buckets". For each of these buckets, the framework also allocated a bucket on the RAM that was corresponding to one of the buckets on the disk. URLs where then mapped to a bucket on the RAM. When a bucket on the RAM was filled, the bucket on the disk that was connected to the bucket on the RAM was accessed. This is where the <key, value> pair and its operations come in. The data was transfered from the RAM bucket to the disk bucket using these <key, value> pair operations. This process allowed DRUM to store a lot of URLs on disk, which meant that the performance would stay the same when the number of URLs increased. [16, 21]

Deep web crawlers

Server-side programming and scripting languages, like PHP and ASP, made life harder for web crawlers. As these languages got more popular, online databases got more accessible as well. This evolution lead to web applications often storing a lot of its data in databases, and then using HTML forms and executable files to generate content using this data. Because of this new way of generating content, web crawlers could no longer just follow links and download pages as they used to before, since the pages didn't include the data they where after. Basically, the contents are hidden from the crawler, which is also why this type of content is referred to as the deep web. To solve this problem, a second generation of web crawlers started to emerge. These web crawlers was made to interact with HTML forms to retrieve data stored on databases.

The way they did it was to just submit HTML forms multiple times, filing in different data in the form fields each time. This did not fix the entire problem of crawling the deep web though. There was still the problem of determining what the crawlers should fill the form fields with. Radio buttons, drop-down lists and other similar fields where not hard to fill out as they are predefined, and the crawler just has to choose all the different possibilities. The big problem comes when the crawler has to fill out text fields. There has been some proposals to how this could be done. [21]

Sriram Raghavan and Hector Garcia-Molina [30] posted an article in 2001, where they suggest a method for crawlers to fill out forms that they call the "task-specific, human-assisted approach". The task-specific part of this approach is that the crawler actually visits the sites it are crawling, and then submits forms and queries to retrieve the hidden pages. The human-assisted part is based on the fact that humans feed the crawler information about what it want it to search for. This means that the crawler can use this data to fill out forms on the pages it finds relevant according to the search. [21, 30]

A year later, in 2002, another method was published by Liddle, Embley, Scott and Yau [17]. They presented a method, that is kind of similar to Raghavan and Garcia-Molinas solution [30], but instead of actually filling out the forms, this method uses HTTP POST and GET requests to the action path of a form it discovers. Firstly they search for forms on the sites they visit. If they find one, they extract all the information they can get from the form, like base URL, the action path, fields and their names, standard field values and so on. The crawler then use the information it finds to do a HTTP GET or POST call to the action path of the form, with every field set to its default value. The crawler also generates several more requests by choosing different combinations of the selected value of the drop-down lists, check-boxes and radio buttons. It also supports the user of the crawler to pass values that should be used as value for text boxes, but the crawler leaves them empty if the user doesn't supply any values. [17, 21]

In 2004, Luciano Barbosa and Juliana Freire [2] came out with another solution to the problem, with an algorithm that was split up in to two parts. The first part of the algorithm is basically to collect data from the website, and then use that data to rate different keywords after how frequently they are used. The second part of the algorithm is to uses a greedy strategy on the list of keywords from the first part of the algorithm, to find the query with the highest coverage. The process to finding this query could be time consuming, but when a query is found, it could be reused later on, to for example update the indexes of a search engine that searches the deep web, meaning the query only has to be calculated one time per web page. [2, 21]

Another year later, a more advanced process was published by Ntoulas, Zerfos and Cho [32]. This solution introduced three policies to how a crawler selects the queries it is going to use. The first one is a random policy. This option takes random keywords from some sort of collection of words, for example a dictionary, and then use these random keywords as values in the query. The hope here is that a random query will return a decent amount of matches. The second policy is a policy based on generic-frequency. This option analyzes some generic document collected from somewhere else, and then sorts the keywords in that document by how frequent they appear. The most frequent keyword is then used in a query to a database. After that query is done, the next keyword is used in the query, and so on until all the keywords are used. The third, and last, policy is an adaptive policy. This option uses the responses it gets from the queries to the database, and then estimates which keywords that are more likely to give the most documents in return. It then updates the query with these new keywords, and keeps on adapting for each query sent. [21, 32]

A few years later, in 2008, yet another solution was introduced by Lu, Wang, Liang and Chen [19]. This approach crawled the web using sampling. They start by creating a sample-database that randomly selects some documents from the total-database (or the original data source). They then use this sample-database to create a pool of queries, and then test these queries on the sample-database. The queries that returned the best result from the tests on the sample-database is then used to collect data from the total-database. So this entire approach is based on the assumption that queries learned from the sample-database, also works well on the total-database. [19, 21]

Rich Internet Application crawlers

As the Internet has evolved, client side web browsers has become more and more powerful, which also has lead to a bigger availability of client-side technologies. These two things combined has meant that computation of data on websites has moved more and more from server-side over to client-side. This means that a lot of content now are hidden on clients, which traditional web crawlers can't access. This is called the "Client-side hidden-web". This is where the Rich Internet Application, or RIA, crawlers are getting in to the picture. There are many different strategies to RIA crawling and I will mention a few in this subsection. [21]

In the beginning, most of the RIA crawlers used either a type of Breadth-First or a type of Depth-First strategy. In 2009, a Breadth-First crawling strategy was used by Duda, Frey, Kossmann, Matter and Zhou [9], in an paper where they focused on AJAX crawling. What makes an AJAX application different from traditional web applications is the fact that the application no longer is just a simple web page identified by a URL. These applications also exists of a series of states, events and transitions. Their version of the Breadth-First AJAX crawling algorithm starts by reading the initial DOM of the website document. Then the algorithm calls the "onLoad" event of the HTML documents body tag. This is AJAX specific, and is something every Javascript-enabled web browser does to construct the initial state. After this initial state is constructed, the Depth-first crawling starts. This crawling is done by triggering all events in the web page it is on, and then also invoking all the Javascript functions corresponding to these events. Every time the DOM changes, a new states is created, and the transition connected to this change is stored. This goes around and around until all states has been triggered. [9, 21]

In 2008, a RIA crawler called "Crawljax" was introduced by Mesbah, Bozdag and van Deursen [23]. This crawler also focused on AJAX crawling, but it used a variant of the Depth-first strategy. The default strategy of the crawler didn't explore all the events in each state, but instead only explored an event from the state where it was first encountered. This means that an event wouldn't be explored on all the different states, leading to some states not being discovered (since an event triggered on one state could produce another state than the event would have done if it was triggered on another state). Because of this, the Crawljax could also be configured to trigger events on all the states that is was enabled, meaning the Crawljax then would use a standard Depth-first strategy instead of the modified one it was originally configured to use. [21, 23]

From 2008 to 2012, there where a couple of crawlers introduced [1, 3, 8], but I am not going to address them in this paper. Instead I am going to skip ahead to 2012, when Peng, He, Jiang, Li, Xu, Li and Ren [27] introduced a greedy crawling strategy. In this strategy, any un-executed events in the current state will be triggered. When the current state runs out of un-executed events, the crawler moves to the state closest to itself that has one or more un-executed events. Then this process is executed again and again. They also introduced a couple of variants to this strategy, where instead of moving to the closest state to itself, the crawler would move to the most recently discovered state or to the state that was closest to the initial state of the web site. They also did a couple of tests on these different strategies, and concluded that all three strategies executed about the same amount of events to end the crawling. This means that all three variants of the strategy had similar performance. [21, 27]

One year later, in 2013, a crawler called "FeedEx" was introduced by Amin Milani Fard and Ali Mesbah [11]. FeedEx is also a greedy algorithm, but not in the same way as the one mentioned above [27]. Instead of finding the closes state with executable events, the FeedEx uses an algorithm that has a matrix that is used to measure how big of an impact a certain event will have on the corresponding states, if it is executed. The matrix then sorts these events by the impact they would make, and the one with the biggest impact is then the first to be executed. There are four factors that the matrix uses to determine the order of the choices. The first one is code coverage. The code coverage factor is based on how much of the application code that is being executed. The second factor is the overall average path diversity. This factor looks at how diverse the exploration of the crawler is. The third factor is the DOM diversity. This factor looks at how much the newly discovered DOMs differ from the ones that has already been discovered. The last, and fourth, factor is the size of of the derived test model. This one kind of explains it self, the factor is how big the derived test model is. [11, 21]

Chapter 3

Research Methodology

3.1 Entire project

This section explains and discusses the different methods used to evaluate both of the research questions.

3.1.1 Choice of programming language

When it comes to the choice of programming language I am going to use to write my test program, it was kind of decided for me when I chose the master thesis suggestion from my supervisor. One of his wishes or suggestions for the thesis was that I wrote the testing program in the Emerald programming language.

3.1.2 Choice of Operating System

The Emerald programming language was made to be compiled and executed on both Linux and Windows computers. This means that the choice laid between either one of the Windows Operating Systems, or one of the many Linux distributions. For the purpose of this thesis, I needed to run my program on a network of computers that communicated with each other over realistic conditions. A perfect solution for this was PlanetLab.

PlanetLab is a network of nodes, running on computers all over the world. Since I was able to get access to this network through the school, and also because Emerald programs already had been tested to work on many of these computers before, the choice fell on using this network. These computers are running on the Linux Operating System, but not all of them are using the same Linux distribution. A list of the different distributions can be seen in Table 3.1, on page 18.

3.1.3 Performance metrics

In addition to the performance metrics specified in section 3.2.1 and 3.3.1, I am also going to use a few metrics to measure the distance between the computers I am using for the testing. These metrics will be

Distribution	Release version
Fedora	8
Fedora	25
CentOS	6.4

Table 3.1: List of Linux distributions on PlanetLab computers.

used to add additional data to the other metrics listed in the sections mentioned above. There are mainly four different metrics that are commonly used to measure distances on the internet, and I am using the two metrics listed in Table 3.2, at page 18. The reason I didn't choose the IP path length or the Autonomous System path length as metrics is because I don't see them having any important impact on the testing done for this thesis. How many routers or Autonomous Systems a packet visits will not have any direct effect on how any of the computers perform. The round-trip time on the other hand will have an effect as this will be a factor in how often a node is able to ping another node. The geographical distance is also important, as it will be an indicator of how long the round-trip time will be.

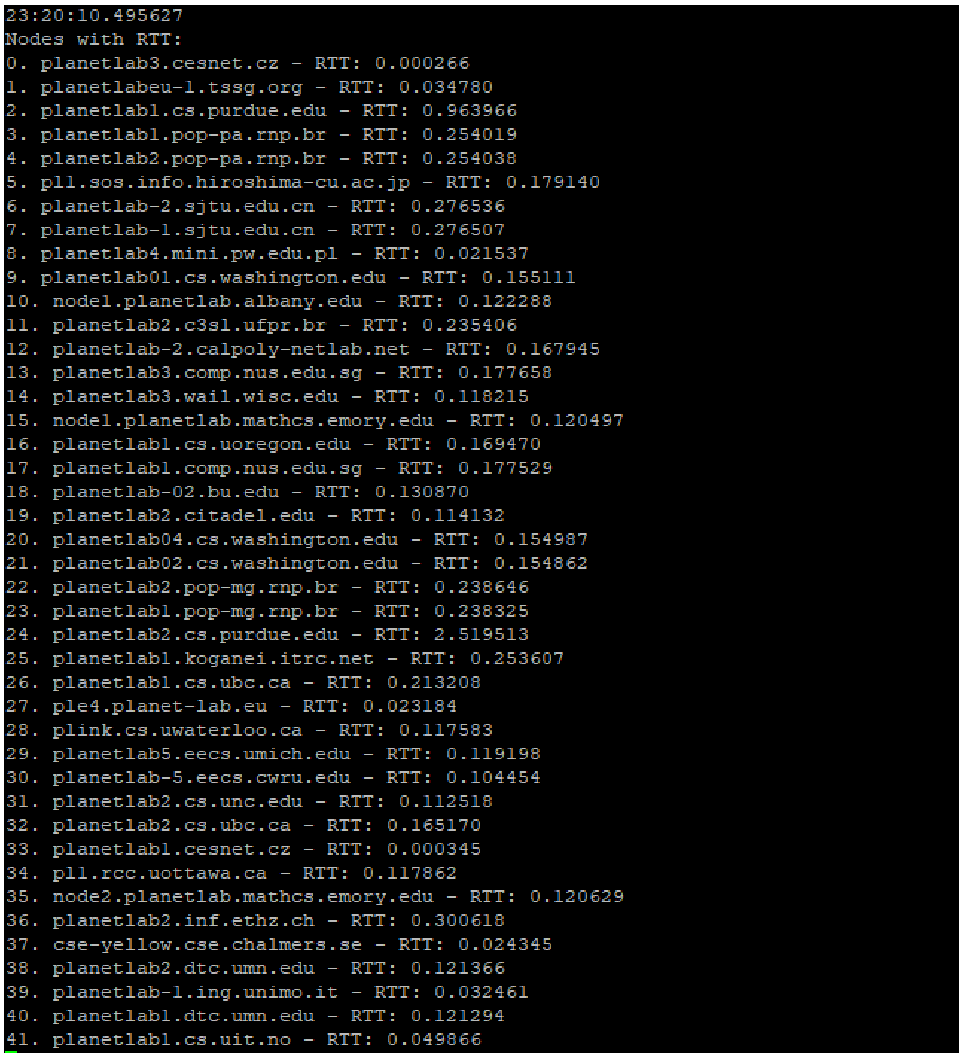
Resarch metric	Description
Round-trip time (RTT)	The round-trip time is the time a packet uses from its source host to the destination host, and then back to the source again. You can read more about this metric in section 2.3.1
Geographical distance	The geographical distance is the length of the great circle arc connecting the source host and destination host on the Earths surface. You can read more about this metric in section 2.3.1

Table 3.2: List of research metrics used to measure distance.

3.1.4 Methods used for data sampling

Round-trip time

To measure the Round-trip time between the different computers I am using the Emerald program this entire thesis is based on. The Emerald program is distributed out to all the different nodes, and all the nodes are then pinging each other through the Emerald software. Each instance of the program is then storing the latest round-trip time between it self and all the other nodes the program is distributed to, meaning I can take use of this stored data to



```
23:20:10.495627
Nodes with RTT:
0. planetlab3.cesnet.cz - RTT: 0.000266
1. planetlabeu-1.tssg.org - RTT: 0.034780
2. planetlab1.cs.purdue.edu - RTT: 0.963966
3. planetlab1.pop-pa.rnp.br - RTT: 0.254019
4. planetlab2.pop-pa.rnp.br - RTT: 0.254038
5. pl1.sos.info.hiroshima-cu.ac.jp - RTT: 0.179140
6. planetlab-2.sjtu.edu.cn - RTT: 0.276536
7. planetlab-1.sjtu.edu.cn - RTT: 0.276507
8. planetlab4.mini.pw.edu.pl - RTT: 0.021537
9. planetlab01.cs.washington.edu - RTT: 0.155111
10. node1.planetlab.albany.edu - RTT: 0.122288
11. planetlab2.c3sl.ufpr.br - RTT: 0.235406
12. planetlab-2.calpoly-netlab.net - RTT: 0.167945
13. planetlab3.comp.nus.edu.sg - RTT: 0.177658
14. planetlab3.wail.wisc.edu - RTT: 0.118215
15. node1.planetlab.mathcs.emory.edu - RTT: 0.120497
16. planetlab1.cs.uoregon.edu - RTT: 0.169470
17. planetlab1.comp.nus.edu.sg - RTT: 0.177529
18. planetlab-02.bu.edu - RTT: 0.130870
19. planetlab2.citadel.edu - RTT: 0.114132
20. planetlab04.cs.washington.edu - RTT: 0.154987
21. planetlab02.cs.washington.edu - RTT: 0.154862
22. planetlab2.pop-mg.rnp.br - RTT: 0.238646
23. planetlab1.pop-mg.rnp.br - RTT: 0.238325
24. planetlab2.cs.purdue.edu - RTT: 2.519513
25. planetlab1.koganei.itrc.net - RTT: 0.253607
26. planetlab1.cs.ubc.ca - RTT: 0.213208
27. ple4.planet-lab.eu - RTT: 0.023184
28. plink.cs.uwaterloo.ca - RTT: 0.117583
29. planetlab5.eecs.umich.edu - RTT: 0.119198
30. planetlab-5.eecs.cwru.edu - RTT: 0.104454
31. planetlab2.cs.unc.edu - RTT: 0.112518
32. planetlab2.cs.ubc.ca - RTT: 0.165170
33. planetlab1.cesnet.cz - RTT: 0.000345
34. pl1.rcc.uottawa.ca - RTT: 0.117862
35. node2.planetlab.mathcs.emory.edu - RTT: 0.120629
36. planetlab2.inf.ethz.ch - RTT: 0.300618
37. cse-yellow.cse.chalmers.se - RTT: 0.024345
38. planetlab2.dtc.umn.edu - RTT: 0.121366
39. planetlab-1.ing.unimo.it - RTT: 0.032461
40. planetlab1.dtc.umn.edu - RTT: 0.121294
41. planetlab1.cs.uit.no - RTT: 0.049866
```

Figure 3.1: An example output of my Emerald program

tell the round-trip time between all the computers. To get as much data as possible, the Emerald program can be set to output this information every second. I can then run the program for a certain amount of time on all the computers, and then pull that data of the nodes to compare all the data. An example of how to output looks, can be seen in Figure 3.1, at page 19.

Geographical distance

3.2 Bandwidth usage

This section explains and discusses the different methods used specifically to evaluate the first research question: How does it affect the bandwidth usage of the nodes?

3.2.1 Performance metrics

When it comes to the Bandwidth of a computer, there are a few different metrics to consider. Since the purpose of this thesis is to look at how a computer is affected, I have chosen the metrics listed in Table 3.3, at page 20.

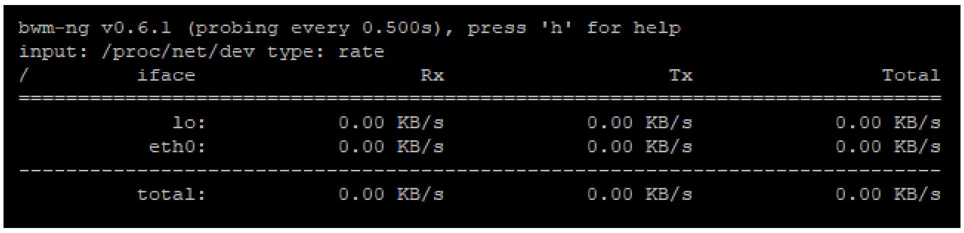
I chose to collect data about both bytes and packets transferred, because both of those metrics could affect the performance and experience of a computer.

Resarch metric	**Description**
Bytes in/s	The number of bytes coming in to the computer every second. This will be recorded over a period of one minute.
Bytes out/s	The number of bytes going out of the computer every second. This will be recorded over a period of one minute.
Packets in/s	The number of packets coming in to the computer every second. This will be recorded over a period of one minute. recorded over a period of one minute.
Packets out/s	The number of packets going out of the computer every second. This will be recorded over a period of one minute. recorded over a period of one minute.

Table 3.3: List of bandwidth research metrics.

3.2.2 Methods used for data sampling

To sample the data on the bandwidth side of things, I used a console-based bandwidth monitor tool called 'bwm-ng' (or Bandwidth Monitor NG) [13]. This is an open-source and free to use tool, under the GNU General Public License v2.0 [12]. With this tool I am able to monitor and record both the bytes and packets transferred, in a given period of time. An example of the standard output from the 'bwm-ng' command is shown in Figure 3.2, at page 21. The different commands I used for this project are listed in Listing 3.1, at page 21. These commands starts an instance of 'bwm-ng' that monitors and records the average, maximum and live bandwidth data. When the program is exited, it prints the information to a CSV file with timestamps of when the data was recorded. This way I can compare data across different nodes.



```
bwm-ng v0.6.1 (probing every 0.500s), press 'h' for help
input: /proc/net/dev type: rate
/         iface                   Rx                   Tx                Total
==============================================================================
            lo:           0.00 KB/s            0.00 KB/s            0.00 KB/s
          eth0:           0.00 KB/s            0.00 KB/s            0.00 KB/s
------------------------------------------------------------------------------
         total:           0.00 KB/s            0.00 KB/s            0.00 KB/s
```

Figure 3.2: An example output of the ’bwm-ng’ command

Listing 3.1: Commands used with ’bwm-ng’

```
bwm-ng -o csv -T avg -c 0 -F outputAvg.csv -I eth0
bwm-ng -o csv -T max -c 0 -F outputMax.csv -I eth0
bwm-ng -o csv -T rate -c 0 -F outputRate.csv -I eth0
```

There are a lot of different tools I could have used to do the same job as ’bwm-ng’. The reason I chose exactly this tool, is first and foremost because of its compatibility with Linux. Since all the machines I am using for testing is running Linux distributions, I needed a tool I could use with those distributions. The second reason is that my access to the computers on the PlanetLab network are somewhat limited. I have sudo access, but since these computers are going to be used by a lot of other people besides my self, I needed a program that didn’t require me changing a lot of shared files. Because I am doing this testing on different machines that are running different Linux distributions and versions, I had to find a tool that was working on all of the computers. The third reason is that I also needed a program that was free to use, and that I could use for writing my own thesis. Since ’bwm-ng’ met all of these requirements, I ended up using that one.

The downfall with ’bwm-ng’ is that you can’t specify a process or program to monitor. It just monitors an entire interface. For the computers used in this thesis, there is only one interface used for all traffic in and out. This interface is called ’eth0’. ’eth0’ is an ethernet cable connected interface, which is positive for our research since an ehternet cable connected interface usually is more stable than a wireless connected interface. But because ’bwm-ng’ monitors all network traffic in to the computers, I need to monitor and record the bandwidth at three different times. Firstly I need to monitor it before I have started any of my programs on the computer. This is to create a baseline for the normal traffic in and out of the computer. Secondly I need to monitor the traffic after all of the computers are connected to each other through the Emerald software. This is before I have started my program, but there will be some traffic going between the computers. Lastly I need to monitor the traffic after the program is started up and running on all of the nodes. This is the data that is going to be most valuable to this thesis, but the monitoring done before the program has started up is also important for comparison to how the computers normally run.

3.3 CPU and memory usage

This section explains and discusses the different methods used specifically to evaluate the second research question: How does it affect the CPU and memory usage of the nodes?

3.3.1 Performance metrics

When it comes to the CPU and memory usage of a computer, the metrics to consider is kind of straight forward. Especially when it comes to looking at how a computer, and the experience of it, is directly affected. The metrics chosen for this thesis is listed in Table 3.4, at page 22.

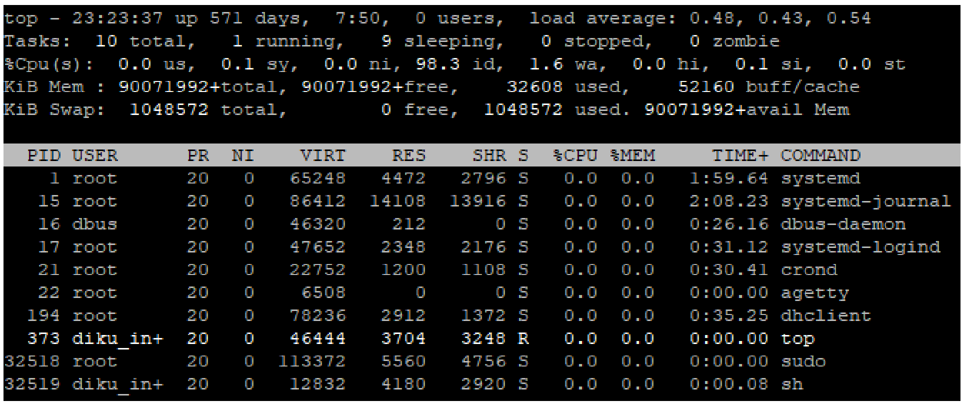
Resarch metric	**Description**
CPU usage in %	The amount of CPU power used, in %, of the maximum CPU capacity. The CPU usage will be recorded every second over a period of one minute.
Memory usage in %	The amount of memory used, in %, of the total memory available to the computer. The memory usage will be recorded every second over a period of one minute.
Memory usage in byte	The amount of memory used in byte. The difference from this metric to the one above, is that this one is directly comparable to data from other computers. This metric will also be recorded every second over a period of one minute.

Table 3.4: List of CPU and memory usage research metrics.

3.3.2 Methods used for data sampling

As I was going to choose the methods to sample data for the CPU and memory usage, I had the same challenges as I had when deciding the methods to use for the sampling of bandwidth usage. I needed to find a tool that I could use across all the different Linux distributions used by the computers in the PlanetLab network, as well as a tool that didn't require big changes to shared configuration files. I also needed a tool that would help me calculate the load average. Luckily for me, there already exists a tool built in to Linux, for CPU and memory monitoring, called 'top' [4]. This tool also calculates and shows the load average for me, which is exactly what I needed.

The command line tool 'top' shows an overview of the current processes and, amongst other things, information about the CPU and



```
top - 23:23:37 up 571 days,  7:50,  0 users,  load average: 0.48, 0.43, 0.54
Tasks:  10 total,   1 running,   9 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 98.3 id,  1.6 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem : 90071992+total, 90071992+free,    32608 used,    52160 buff/cache
KiB Swap:  1048572 total,        0 free,  1048572 used. 90071992+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0   65248   4472   2796 S   0.0  0.0   1:59.64 systemd
   15 root      20   0   86412  14108  13916 S   0.0  0.0   2:08.23 systemd-journal
   16 dbus      20   0   46320    212      0 S   0.0  0.0   0:26.16 dbus-daemon
   17 root      20   0   47652   2348   2176 S   0.0  0.0   0:31.12 systemd-logind
   21 root      20   0   22752   1200   1108 S   0.0  0.0   0:30.41 crond
   22 root      20   0    6508      0      0 S   0.0  0.0   0:00.00 agetty
  194 root      20   0   78236   2912   1372 S   0.0  0.0   0:35.25 dhclient
  373 diku_in+  20   0   46444   3704   3248 R   0.0  0.0   0:00.00 top
32518 root      20   0  113372   5560   4756 S   0.0  0.0   0:00.00 sudo
32519 diku_in+  20   0   12832   4180   2920 S   0.0  0.0   0:00.08 sh
```

Figure 3.3: An example output of the 'top' command

memory usage of each process, as well as information about the load average. An example of the output from the 'top' command is shown in Figure 3.3, at page 23. Since this tool shows output for each process, I can monitor the exact process used by my Emerald program. This gives me more exact values than what I could get from the bandwidth monitoring.

To make the sampling of data precise, I couldn't just read the data of the 'top' output while it was running. I needed to automate the process, so that the program ran for a given time and outputted the data to a file at a specific rate. To do this, I wrote a small Bash script that loops 400 times, and for each loop sleeps one second before running the 'top' command. This means that the script will run for 400 seconds. I did it this way because I then have time to start the script on all of the computers I am testing on, and still get the top 'command' running for at least one minute simultaneously on all of them. The script can be seen in Listing 3.2, at page 24. As you can see in the script, I have added a few options to the 'top' command:

- '-b'. The '-b' option starts 'top' in a so called 'Batch mode'. This 'Batch mode' makes it possible to send the output of the program to, for example, a file. I take use of this option because I want to get all the output in to one single file, that I then can pull of the computer and compare to the data i get from other computers. To do this, I append the output of the bash script to an output file, using the '»' syntax.

- '-p'. The '-p' option is used to specify a specific PID the program should monitor. Parameter 'xxx' in the Bash script is where I insert the specified PID for the process of the emerald program I am running on the machine. I specify the PID because I only care about the data from the process used by my program.

- '-n'. The '-n' option tells the 'top' program the number of iterations it should produce before ending. As I only want one iteration per second, I add parameter '1' to this option.

Listing 3.2: Script used to run 'top' (xxx is process PID)

```
#!/bin/bash
for i in {1..400}
do
   sleep 1
   top -b -p xxx -n1
  echo -e "\n\r"
done
```

Chapter 4

Experimental setup

4.1 PlanetLab Architecture

As mentioned in section 3.1.2, at page 17, I have chosen PlanetLab as the platform for my testing. You can read more about exactly what PlanetLab is in section 2.2, on page 3, but in this section I am going deeper in to how the Architecture of PlanetLab works.

4.1.1 Terminologies

Before explaining how the Architecture of PlanetLab works, there are a few terminologies that are needed to understand it. These terminologies are listed and explained in this section.

Principal Investigator (PI)

The Principal Investigator, or PI, is the one that is responsible for managing users and slices at the site they are connected to. The PIs are also legally responsible for everything that happens from the slices they create. A PI is often a member of the faculty at whatever institution the site they are PI for is connected to. Usually just one PI per site. [26]

Technical Contact

The Technical Contact is the one who is responsible for installation, maintenance, and monitoring of the nodes at the site they are connected to. Each site is required to have a Technical Contact. When a node goes down, or there are any other problems with one of the sites nodes, the Technical Contact is the one who is contacted. [26]

User

A user is anyone who uses the PlanetLab network. This means anyone who either develops or deploys applications on PlanetLab. [26]

Authorized Official

An Authorized Official is someone who contractually or legally can bind an institution. For example the president of an institution. A signature from an Authorized Official is required for an institution to join the PlanetLab network. [26]

Site

A site is a location where a physical PlanetLab node are located. So if an institution has a physical PlanetLab node located at their institution, they are per definition a site. [26]

Node

A node is a server that is dedicated to run components of PlanetLab services. [26]

Slice

A slice is a set of resources that are allocated and distributed across the PlanetLab network. This means, for most users, access to private virtual servers on a given number of PlanetLab nodes, through a UNIX shell. The PI of the site a user is connected to, are responsible of creating the slice and assign users to it. Users, included PIs, can then assign PlanetLab nodes to it, and after node has been assigned to the slice, virtual servers for that slice are created on all of the selected nodes. A slice doesn't last forever, and to keep it alive, users has to renew the slice for it to remain valid. When a slice expires, all data associated to it is deleted. [26]

Sliver

A sliver is the definition for a slice running on a node. The sliver is what you connect to when you ssh in to a specific node that is part of your slice. [26]

4.1.2 The Architecture

In this section I am going to list an explain some of the key features of the PlanetLab Architecture. In Figure 4.1, you can see an overview of the Architecture of a PlanetLab node. Parts of this image will be explained in this section.

Distributed Virtualization

When you, as a user, are connected to nodes through your slice, you want to isolate your activities from other activities that are being executed from other slices on the same nodes as your are. To solve this, PlanetLab

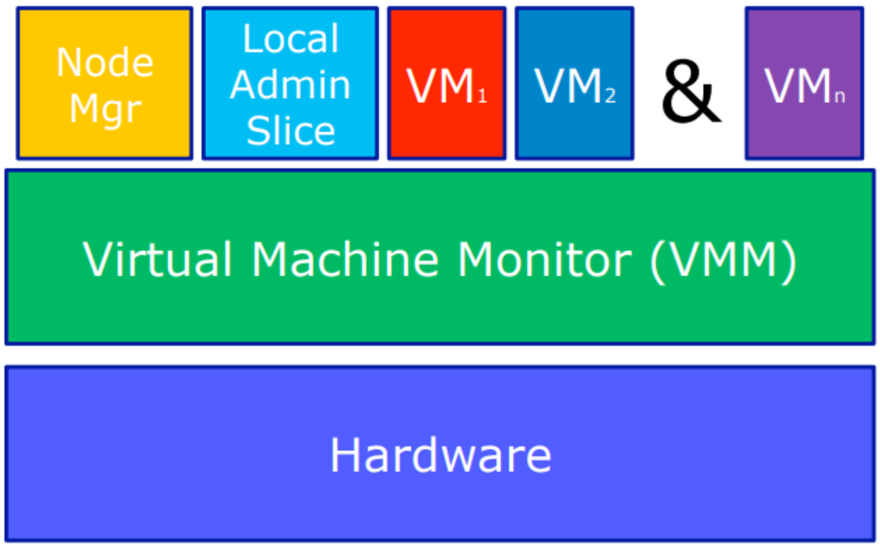


Figure 4.1: Architecture of a PlanetLab node. Taken from [26]

gives you your own file system and process control, that are isolated from other slices. The CPU cycles and network bandwidth is still shared with other slices that are connected to the node, but those slices wont get access to your file system. [26]

Trust Relationships

To create a trustworthy relationship between a node owner and a user, PlanetLab has an api called PlanetLab Central (PLC), that works as a "Trusted Intermediary" between the two parties. An illustration of this can be seen in Figure 4.2, at page 4.2. There are basically four steps to this interaction between node owner, PLC and user: [26]

1. The PLC shows that it trust the user by giving them credentials to get access to a slice. [26]

2. The user trust the PLC to create a slice for them, and the user also trust the PLC to inspect its credentials. [26]

3. The Node owner trust the PLC to give user access to the node, and also trust that the PLC will map the network activity to the correct user. [26]

4. The PLC trust that the node owner will keep the node physically secure. [26]

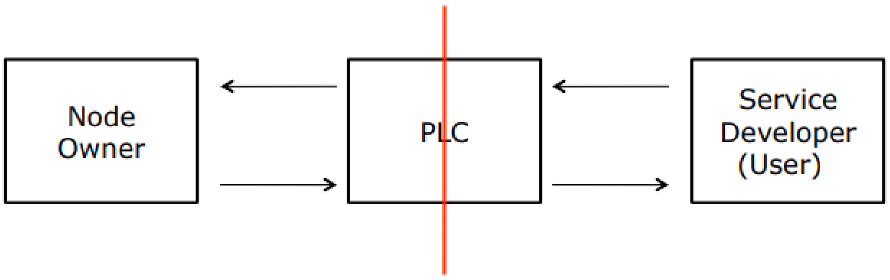


Figure 4.2: Trust relationship between Node Owner, PLC and Service Developer. Taken from [26]

4.2 Node setup

In this section I am explaining my process for how I chose the PlanetLab nodes used for this testing, and how I set the nodes up and ready for testing.

4.2.1 Access to PlanetLab, and choice of nodes

Access to a PlanetLab slice was given to me by my supervisor. Trough that slice, I had access to 401 different nodes, theoretically. My plan was to run the program on 100 nodes and do my testing while the program was running on all of them. Unfortunately, that didn't go as planned. When I started to try and connect to the nodes, I quickly learned that I wasn't able to connect to near as many nodes as I hoped to. Most of the nodes I tried to connect to timed out before I was even able to try and log in to them. Out of the 401 nodes I should have had access to through my slice, I was able to connect to 44 of them. Luckily, these nodes where kind of spread out all over the world, so I didn't just get access to nodes that where inn the same area. A list of all the Nodes, and where they are located, are listed in Table 4.1, at page 4.1. The fact that these was the only nodes I could connect to, the choice of which nodes to use became pretty easy.

4.2.2 Configuring nodes for testing

To set the nodes up for testing, I first needed to install Emerald on all of the nodes. This had to be done manually for every single node, and the way to do it depended a little bit on the node it self. The easy nodes where the nodes that worked with 32-bit binaries, as de Emerald compiler is a 32-bit software.

Nodes working with 32-bit binaries

For the nodes working with 32-bit binaries, I first needed to run the commands listed in Listing 4.1. After those 3 commands was executed, I needed to edit the '.bashrc' file, located in the home folder, and add the lines listed in Listing 4.2. Finally I needed to logout of the remote connection to the node, and then log in again. After that, the Emerald compiler was installed en fully working.

Listing 4.1: Commands used to install Emerald for nodes working with 32-bit binaries

```
$ wget http://www.uio.no/studier/emner/matnat/ifi/
    INF5510/v15/emerald-0.99-linux.tar.gz
$ tar xvf emerald-0.99-linux.tar.gz
$ mv emerald-0.99-linux emerald
```

Listing 4.2: Lines added to '.bashrc' file

```
export EMERALDROOT=/home/diku_inf5510/emerald/
export EMERALDARCH='i686mt'
export PATH="$EMERALDROOT/bin:$PATH"

export TERM=xterm-256color
```

Nodes not working with 32-bit binaries

For the nodes not working with 32-bit binaries, I first needed to run the commands listed in Listing 4.3. After those 5 commands was executed, I needed to edit the '.profile' file, located in the home folder, and replace the lines in the file with the lines listed in Listing 4.4. Finally I needed to logout of the remote connection to the node, and then log in again. After that, the Emerald compiler was installed en fully working. Only difference from the other Emerald install is that I neede to use the aliases 'emx32' and 'ec32', instead of normal 'emx' and 'ec', to compile and run the Emerald software.

Listing 4.3: Commands used to install Emerald for nodes not working with 32-bit binaries

```
$ wget https://www.uio.no/studier/emner/matnat/ifi/
    INF5510/v18/bind_public.so
$ wget http://www.uio.no/studier/emner/matnat/ifi/
    INF5510/v15/emerald-0.99-linux.tar.gz
$ tar xvf emerald-0.99-linux.tar.gz
$ mv emerald-0.99-linux emerald
$ sudo dnf install glibc.i686
```

Listing 4.4: Lines added to '.profile' file

```
[ -f /etc/planetlab.profile ] && source /etc/planetlab
    .profile

export EMERALDROOT=/home/diku_inf5510/emerald/
export EMERALDARCH='i686mt'
export PATH="$EMERALDROOT/bin:$PATH"

export TERM=xterm-256color

alias emx32="LD_PRELOAD=~/bind_public.so emx"
alias ec32="LD_PRELOAD=~/bind_public.so ec"
```

Hostname	**Location**
ple1.cesnet.cz	Prague, Czech Republic
planetlab3.cesnet.cz	Prague, Czech Republic
planetlab1.cs.uit.no	Tromsø, Norway
cse-yellow.cse.chalmers.se	Gothenburg, Sweden
planetlab1.cesnet.cz	Prague, Czech Republic
planetlab1.dtc.umn.edu	Minnesota, USA
planetlab-1.ing.unimo.it	Modena and Reggio, Italy
planetlab2.dtc.umn.edu	Minnesota, USA
planetlab2.inf.ethz.ch	Zürich, Switzerland
planetlab4.mini.pw.edu.pl	Warsaw, Poland
node2.planetlab.mathcs.emory.edu	Georgia, USA
pl1.rcc.uottawa.ca	Ottawa, Canada
planetlab2.cs.ubc.ca	Vancouver, Canada
planetlab2.cs.unc.edu	North Carolina, USA
planetlab-5.eecs.cwru.edu	Ohio, USA
planetlab5.eecs.umich.edu	Michigan, USA
plink.cs.uwaterloo.ca	Waterloo, Canada
ple4.planet-lab.eu	Paris, France
planetlab1.cs.ubc.ca	Vancouver, Canada
planetlab1.koganei.itrc.net	Tokyo, Japan
planetlab2.cs.purdue.edu	Indiana, USA
planetlab1.pop-mg.rnp.br	Minas Gerais, Brazil
planetlab2.pop-mg.rnp.br	Minas Gerais, Brazil
planetlab02.cs.washington.edu	Washington, USA
planetlab04.cs.washington.edu	Washington, USA
planetlab2.citadel.edu	Carolina, USA
planetlab-02.bu.edu	Massachusetts, USA
planetlab1.comp.nus.edu.sg	Singapore
planetlab1.cs.uoregon.edu	Oregon, USA
node1.planetlab.mathcs.emory.edu	Georgia, USA
planetlab3.wail.wisc.edu	Wisconsin, USA
planetlab3.comp.nus.edu.sg	Singapore
planetlab-2.calpoly-netlab.net	California, USA
planetlab2.c3sl.ufpr.br	Paraná, Brazil
node1.planetlab.albany.edu	New York, USA
planetlab01.cs.washington.edu	Washington, USA
salt.planetlab.cs.umd.edu	Maryland, USA
planetlab-1.sjtu.edu.cn	Shanghai
planetlab-2.sjtu.edu.cn	Shanghai
pl1.sos.info.hiroshima-cu.ac.jp	Hiroshima, Japan
planetlab2.pop-pa.rnp.br	Pará, Brazil
planetlab1.pop-pa.rnp.br	Pará, Brazil
planetlab1.cs.purdue.edu	Indiana, USA
planetlabeu-1.tssg.org	Waterford, Ireland

Table 4.1: List of nodes with location.

Chapter 5

Testing

Node nr.	Hostname	Linux Distribution
1	planetlab1.cs.uit.no	Fedora 25
2	cse-yellow.cse.chalmers.se	Fedora 25
3	planetlab3.cesnet.cz	Fedora 8
4	planetlab2.inf.ethz.ch	Fedora 8
5	planetlab4.mini.pw.edu.pl	Fedora 8
6	planetlab2.cs.ubc.ca	Fedora 8
7	ple4.planet-lab.eu	Fedora 25
8	planetlab1.pop-mg.rnp.br	Fedora 8
9	planetlab1.comp.nus.edu.sg	Fedora 8
10	planetlab-2.calpoly-netlab.net	Fedora 8
11	planetlabeu-1.tssg.org	Fedora 25
12	planetlab2.pop-pa.rnp.br	Fedora 8
13	node2.planetlab.mathcs.emory.edu	Fedora 8
14	pl1.rcc.uottawa.ca	Fedora 8

Table 5.1: List of test nodes with Linux distribution.

5.1 Bandwidth usage

This section presents and comments on the results of the testing done for the bandwidth on the nodes listed in Table 5.1, at page 33. The research metrics used for this testing is listed in section 3.2.1, in Table 3.3 at page 20.

5.1.1 Baseline monitoring

Before any testing is done with either the Emerald software or my program running on the computers, I needed to do some baseline monitoring to see how the bandwidth usually behaves on each of the nodes. The results of this testing is shown in Tables 5.1, 5.2, 5.3 and 5.4.

The reason we have to do a baseline testing for the bandwidth usage is that the tool chosen for the bandwidth testing, 'bwm-ng', doesn't let the user specify a specific process to monitor. You have to monitor the traffic for the entire computer. As the testing for this thesis is not done in a fully controlled environment, with computers out on the open internet that are controllable by other people in the PlanetLab network, the bandwidth testing wont be precise. This of course has a few negative sides to it, but also a few positive. Unfortunately, we have to deal with the negative parts in the testing, to get value from the positive sides later on.

If we take a look at Figure 5.1, we can see that the average incoming bytes per second is pretty low on all of the computers. The highest average is approximately 0.358 megabytes/s on node 4, which really isn't that much of traffic. A few of the nodes has some bigger spikes, as we can see from the column showing the maximum Bytes/s second recorded, but this again is normal. These spikes lasts, for the most part, just for short periods of time, where a bigger amount of data is transferred to the computer.

In Figure 5.2, the chart showing outgoing bytes per second, we can see that the story is pretty much the same as for the incoming bytes. The highest average here is only about 0.09 megabytes/s, which is even lower than for the incoming bytes. As for Figure 5.1, there are a few spikes in the maximum outgoing bytes/s, but this is also normal as these most likely are just periods of time where a few bigger chunks of data is sent out from the computer.

I have also monitored the outgoing and incoming packets for each of the nodes selected for testing. A chart for the incoming packets per second, can be seen in Figure 5.3. If we compare this chart to the one in Figure 5.1, we can see that the spikes are pretty much exactly the same. This is of course because all data is transferred in packets, meaning that for the nodes where there are transferred a bigger number of packets per second, the bytes transferred per second will also be bigger. The fact that these charts are pretty similar means that the monitoring done with 'bwm-ng' is working as it should be. The values in the y-axis is of course smaller for the chart in Figure 5.3 than for the one in Figure 5.1, because one packet always is bigger than one byte.

The last values to look at for the baseline monitoring, is the ones shown in Figure 5.4. If we compare this chart to the chart in Figure 5.2, we can see that the spikes also match between these charts. This is because of the same principle I explained for the incoming bytes and packets. The y-axis does also have smaller values in Figure 5.4 than in Figure 5.2, because incoming packets are of course also bigger than one byte.

5.1.2 Monitoring of Emerald software

As explained in section 4.2.2, each of the nodes listed in Table 4.1 has to be connected through the Emerald software before I can start testing

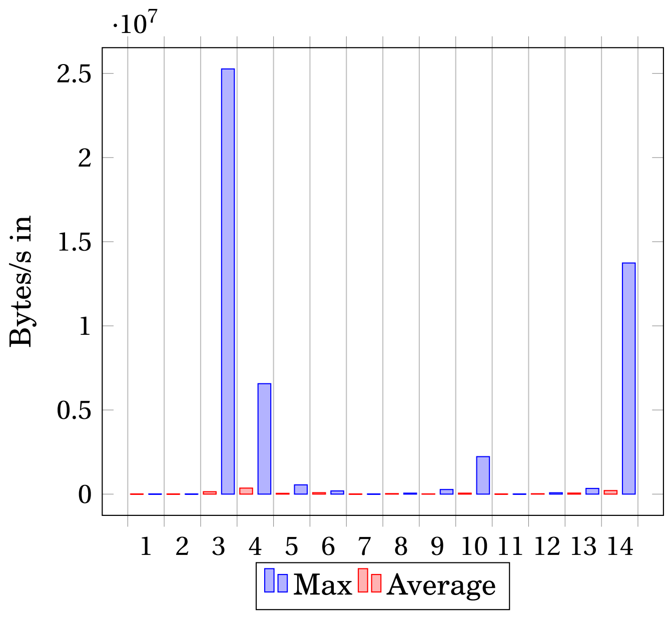


Figure 5.1: Bandwidth graph Bytes/s in, Baseline

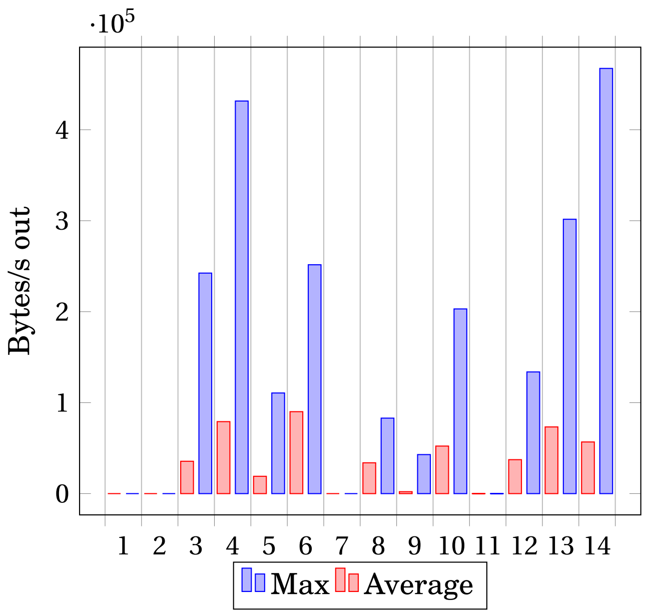


Figure 5.2: Bandwidth graph Bytes/s out, Baseline

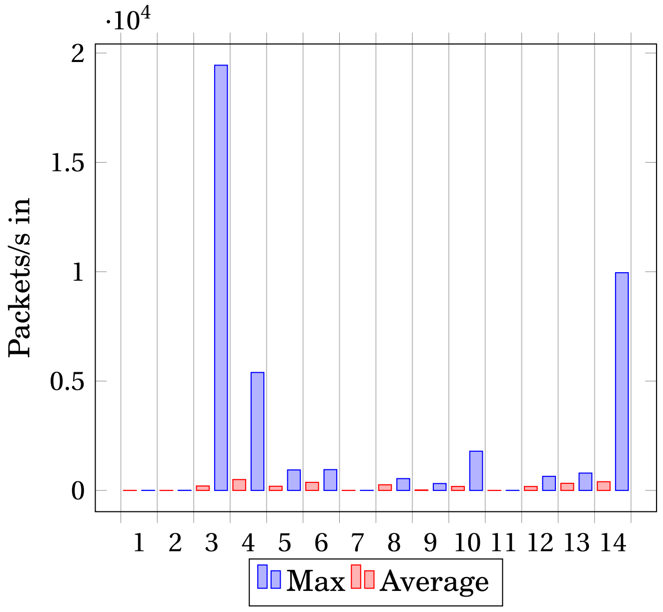


Figure 5.3: Bandwidth graph Packets/s in, Baseline

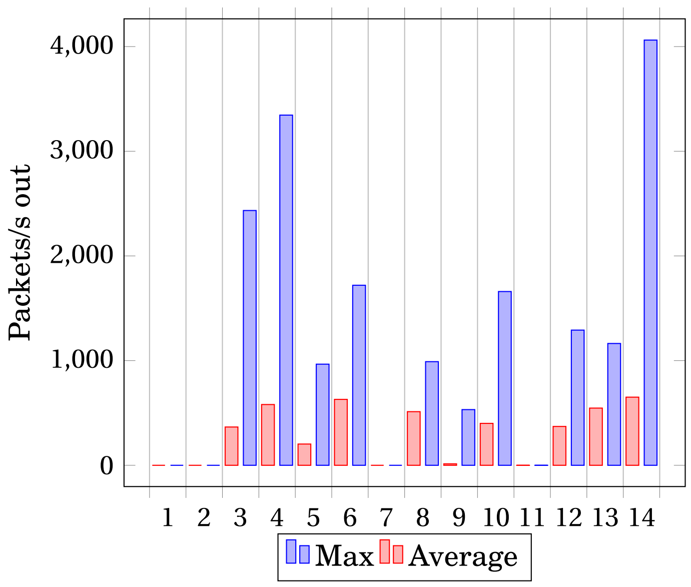


Figure 5.4: Bandwidth graph Packets/s out, Baseline

with the Emerald program I have created. This also includes the nodes I have chosen for testing, listed in table 5.1. Since this connection most likely will produce some data transfer between the nodes, in addition to the data that will be transferred from my program, I have monitored the bandwidth of the 14 testing nodes while they are connected through the Emerald software. The results of this monitoring can be seen in Table 5.5, 5.6, 5.7 and 5.8.

Let us first take a look at the chart in Figure 5.5. If the testing for this thesis had been done in a controlled environment, I would have expected the incoming bytes per second to be at least the same amount as for the baseline monitoring, or higher. Since I am not doing the testing in a controlled environment however, this is not the case. There are some similarities though, for example do nodes 4 and 14 have kind of the same spike in the maximum bytes/s transferred, but there are also some differences like node 5 also having a pretty big spike in the maximum bytes/s transferred. The maximum bytes/s transferred is, on the other hand, not the important part to look at here. The important part is the average bytes/s transferred. If we compare the average between the values in Figure 5.5 and Figure 5.1, we can see that some of the nodes has a higher average, but some of the nodes actually has a lower average as well. The biggest difference between the two charts, is actually on the negative side. Node 14 has a lower average of 0.14 megabytes/s than in the baseline. 0.14 megabytes/s is not a huge difference however, and it is most likely just a coincidence that this occurred, and if I would have done the monitoring with a few minutes of difference, this could have just as much been on the positive side of the spectrum. The conclusion here is that there doesn't seem to be any noticeable difference between the baseline monitoring and the monitoring while the Emerald software is running, when it comes to the incoming bytes/s.

For the outgoing bytes per second, shown in Figure 5.6, I would expect a similar result in relation to the baseline monitoring, as we got for the incoming bytes per second. If we take a look at the chart, and compare it to the chart in Figure 5.2, we can see that that prediction is pretty correct. The most noticeable difference here is the maximum bytes/s from node 3. This is a bit lower in Figure 5.6, but other than that the charts look pretty similar. The difference in the average outgoing bytes/s is actually even smaller than for the incoming bytes/s, with the biggest difference being just 177,32 bytes/s (0,00018 megabytes/s). This again confirms my conclusion in the last paragraph, stating that the Emerald software has very little impact on the bytes transferred per second, if any at all.

As for the incoming packets/s, shown in Figure 5.7, we can see that the chart, as expected, matches the chart in Figure 5.5. The spikes in the maximum column of the charts are on the same nodes, and the columns showing the average values also look pretty much alike. As before, the y-axis does of course have lower values on the chart showing packets/s, than on the chart showing bytes/s. Other than that, the

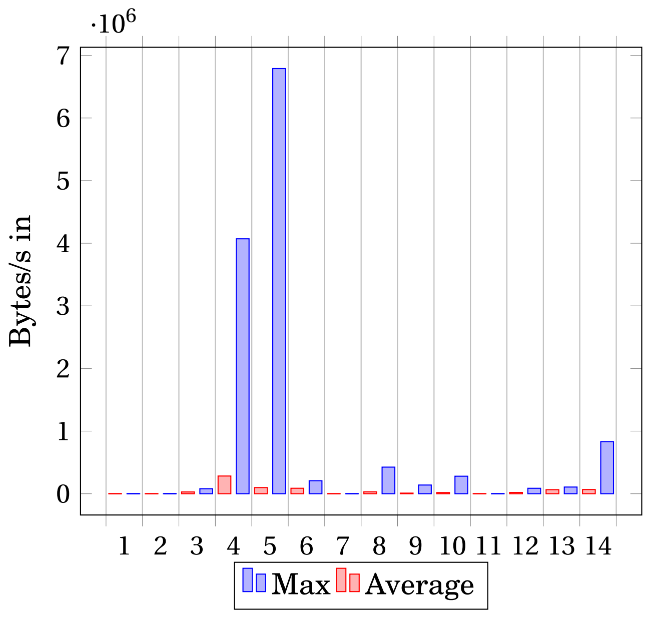


Figure 5.5: Bandwidth graph Bytes/s in, Emerald software running

charts matching each other as they should be.

When it comes to the chart showing the outgoing packets/s, shown in Figure 5.8, I am expecting the same result as I did above. The chart should match the chart for outgoing bytes/s. If we compare the chart with the one in Figure 5.6, we can see that that expectation is correct. The spikes on both the bars showing maximum packets/s, as well as the bars showing the average packets/s, matches the corresponding bars in the chart showing outgoing bytes/s. The only difference is, again, only that the values in the y-axis of the chart showing packets/s are lower than the values in the chart showing bytes/s.

5.1.3 Monitoring of the Program

After the baseline monitoring and monitoring of the Emerald software is finished, the only thing left to monitor is the actual Program created for this thesis. The data monitored in this section is the data that will actually tell us something, and is the data that is going to be most important when it comes to looking at how the bandwidth is affected. All the data from this monitoring can be seen in Figures 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19 and 5.20.

As we have done for the earlier sections of this chapter, let us start by looking at the incoming bytes per second. At first eyesight, the chart in Figure 5.9 looks pretty much identical to the chart from the last section, Figure 5.5. We see the same spikes on the same nodes, and the differences are pretty minimal. If we compare it to the chart in Figure 5.1, from the baseline, the differences are a bit bigger, but still not that noticeable. To make it a bit easier to see the difference, I have

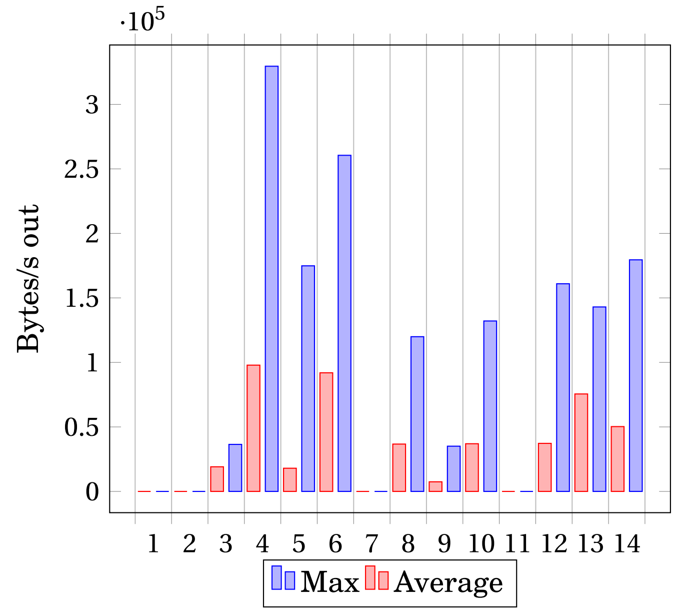


Figure 5.6: Bandwidth graph Bytes/s out, Emerald software running

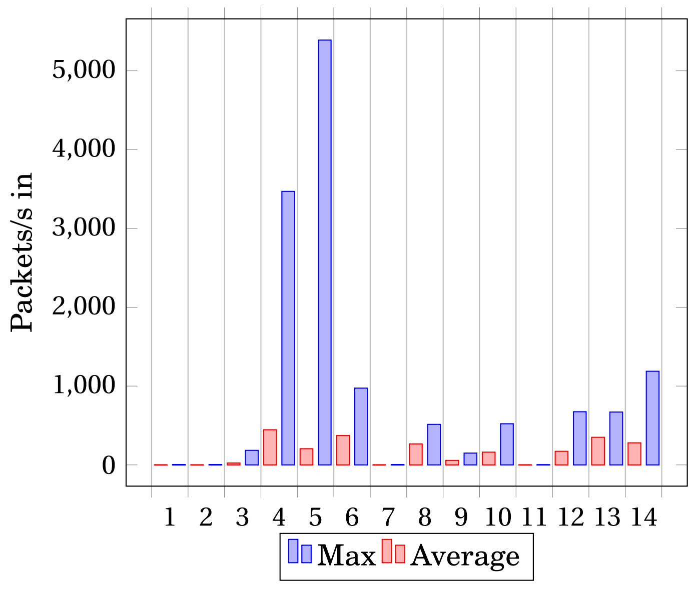


Figure 5.7: Bandwidth graph Packets/s in, Emerald software running

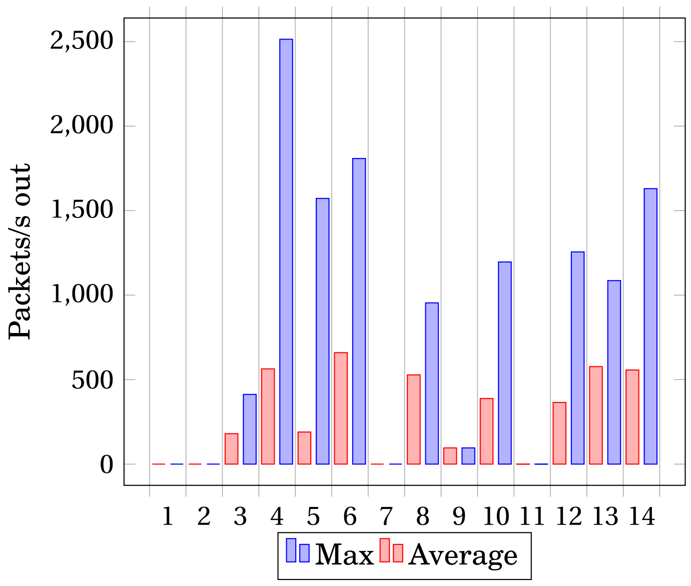


Figure 5.8: Bandwidth graph Packets/s out, Emerald software running

created a chart that shows the difference from the chart in Figure 5.9 to the baseline chart for incoming bytes/s. This chart can be seen in Figure 5.13. As we can see in that chart, the biggest differences here is the bars showing the maximum bytes/s recorded. As I have mentioned earlier in this chapter, the maximum bytes/s doesn't directly tell us anything about how the data traffic actually has been for most of time, as this can be just one transfer of a file that is bigger than the usual data transferred to this computer. Therefore, I have created another chart, that shows the exact same data as in Figure 5.13, just that it only shows the bars showing the average values. This chart can be seen in Figure 5.17. Here we can see that there are some differences in the average incoming bytes/s, but it isn't that big of a difference from the baseline. The biggest difference is on the positive side though, but it is just about 0,386 megabytes/s, which really isn't that much. I would say that that is way inside the margin of error and that it could quickly have changed based on the time this monitoring had been done. So it doesn't seem like the incoming bytes/s is affected that much, at least not enough for us to notice with the tools used in this thesis.

For the outgoing bytes per second, the data from the monitoring is shown in the chart in Figure 5.10. If we compare that chart to the chart in Figure 5.6, we can actually see a bit of a difference. At least more than we did for the charts from the incoming bytes per second. We can also see a difference if we compare it to the chart from the baseline, in Figure 5.2. Even though the biggest differences seem to be in the bars showing the maximum outgoing bytes/s, there are also some differences in the bars showing the average outgoing bytes/s. To get a better look at the difference, I have, like for the chart showing the incoming traffic, created a chart showing the difference between the chart showing the outgoing bytes/s, in Figure 5.10, and the baseline in Figure 5.2. This chart can be seen in Figure 5.14. Again, the bars showing the maximum bytes/s are the ones with the biggest difference, but the bars showing the average bytes/s are actually also showing a bit of a difference, and mostly on the positive side. To get an even closer look at the average bytes/s bars, I have created a dedicated chart just for showing the average outgoing bytes/s. This chart can be seen in Figure 5.18. As we can see in that chart, almost everyone of the nodes has either no difference from the baseline in average bytes/s or they have a higher average. The only two nodes whit a lower average in outgoing bytes/s are nodes 10 and 11. And then we are just talking about 695,68 bytes/s on the lowest. On the other hand, there are multiple nodes with a higher average in outgoing bytes/s, with around, or over, 0,02 megabytes/s. The highest one is node 13 with a difference of 0,045 megabytes/s. Again, this is really not that much of traffic, but we might be seeing an effect from the program, since it's not just a couple of the nodes that are showing an increase in average outgoing bytes per second.

When it comes to the incoming packets per second, I have also created a chart for that data. The chart can be seen in Figure 5.11. If we compare that to the chart showing the incoming bytes/s, in Figure 5.9, we can see that the charts look pretty similar, as they should. I also created a chart for the difference in the incoming packets/s from this chart to the one created for the baseline. This chart can be seen in Figure 5.15. Again, if we compare that chart to the one in Figure 5.13, we can see that those charts look pretty similar as well. This again confirms the fact that the 'bwm-ng' tool has collected the data correctly. Finally, I also created a chart for the difference in incoming packets/s from the baseline, just showing the average incoming packets/s. This can be seen in Figure 5.19. And if we compare that to the chart in Figure 5.17, they should be pretty similar as well. Even though the charts actually look a bit different, the difference in y-axis values are so big, that it isn't that off. We can see kind of the same trends in the bars.

The last metric to look at is the outgoing packets per second. The chart showing this data can be seen in chart 5.12. If we compare that chart to the one showing outgoing bytes/s, in Figure 5.10, we can see that this to charts are pretty much alike, as expected. Again, there are some differences, but the trend in the bars are pretty much the same. I also created a chart for the difference in the outgoing packets/s from this chart to the one created for the baseline. See Figure 5.16. If we take a look at that chart, and compare it to the one in Figure 5.14, showing the difference between the baseline and the outgoing bytes/s, we can see that these charts also look the same. Lastly, to compare the outgoing packets/s to the chart in Figure 5.18, I created a chart just showing the average packets/s. This can be seen in Figure 5.20. If we compare those charts as well, we can see that they also are pretty much the same, and that the bars are showing the same trends.

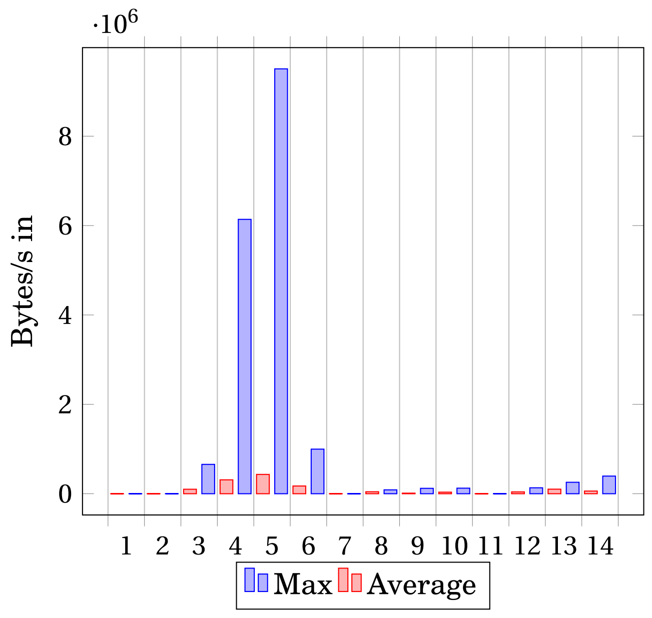


Figure 5.9: Bandwidth graph Bytes/s in, Program running

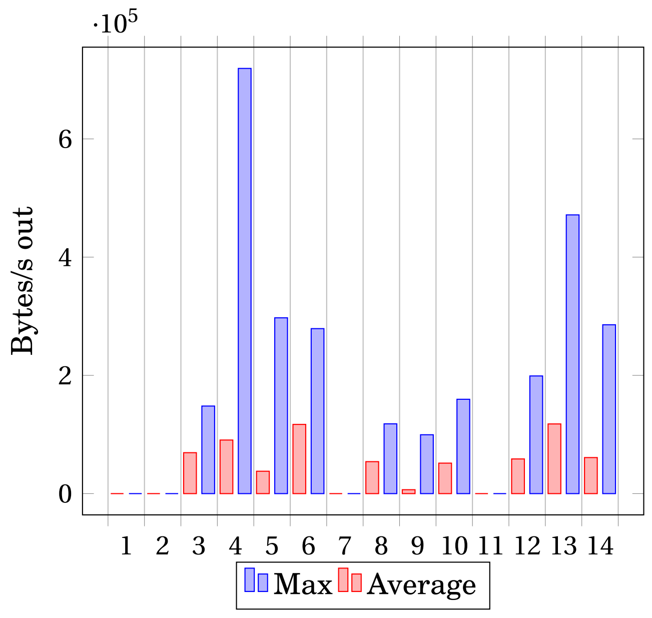


Figure 5.10: Bandwidth graph Bytes/s out, Program running

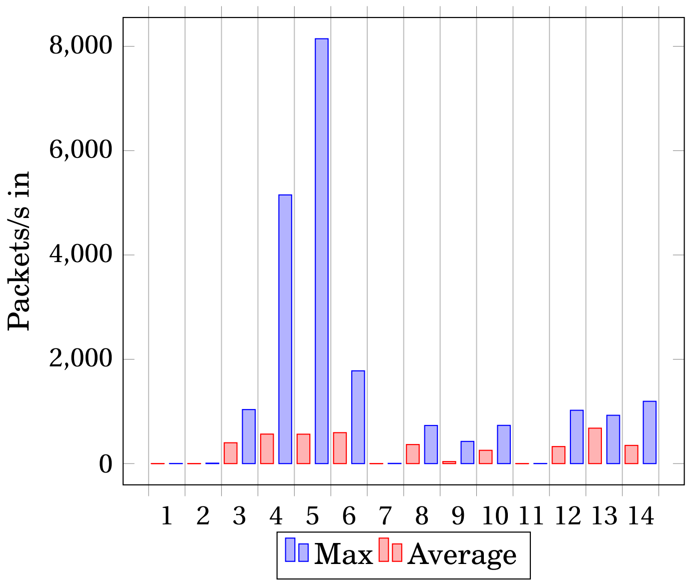


Figure 5.11: Bandwidth graph Packets/s in, Program running

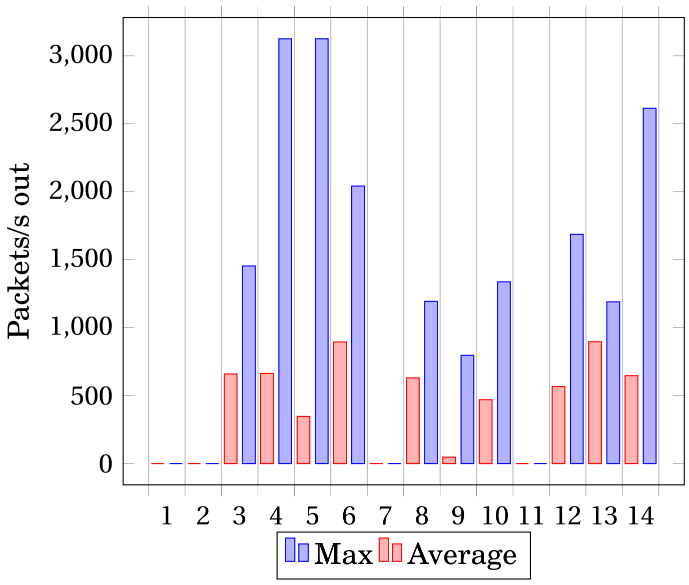


Figure 5.12: Bandwidth graph Packets/s out, Program running

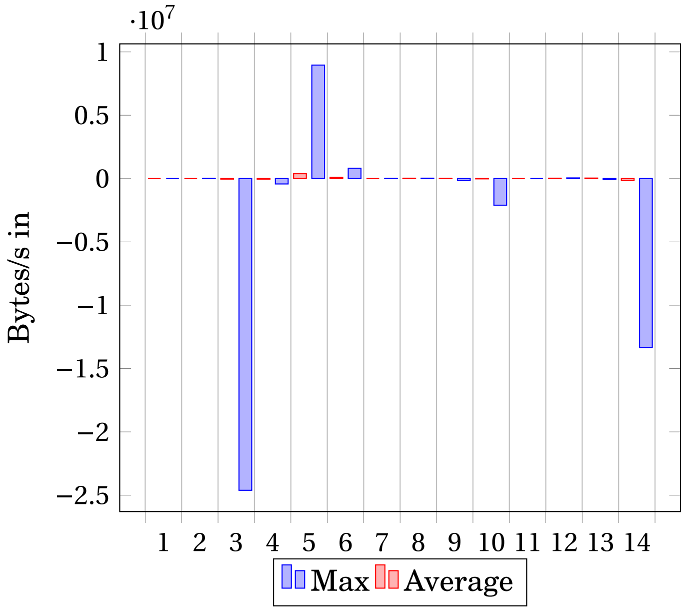


Figure 5.13: Bandwidth graph Bytes/s in, Program running (baseline subtracted)

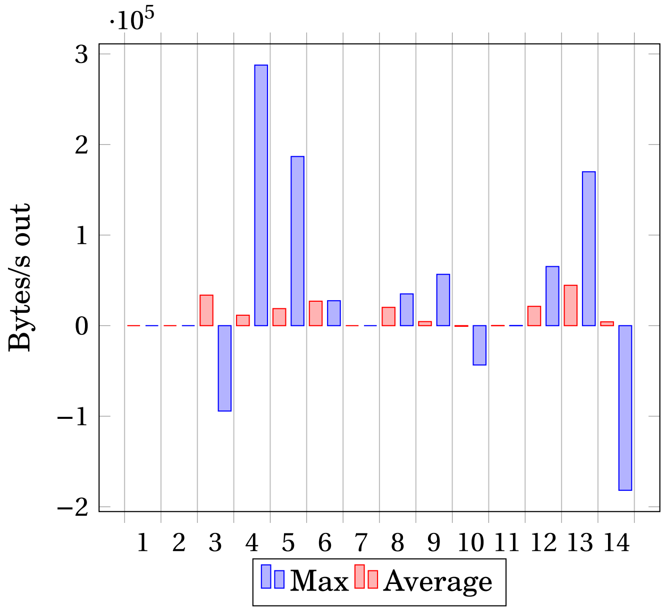


Figure 5.14: Bandwidth graph Bytes/s out, Program running (baseline subtracted)

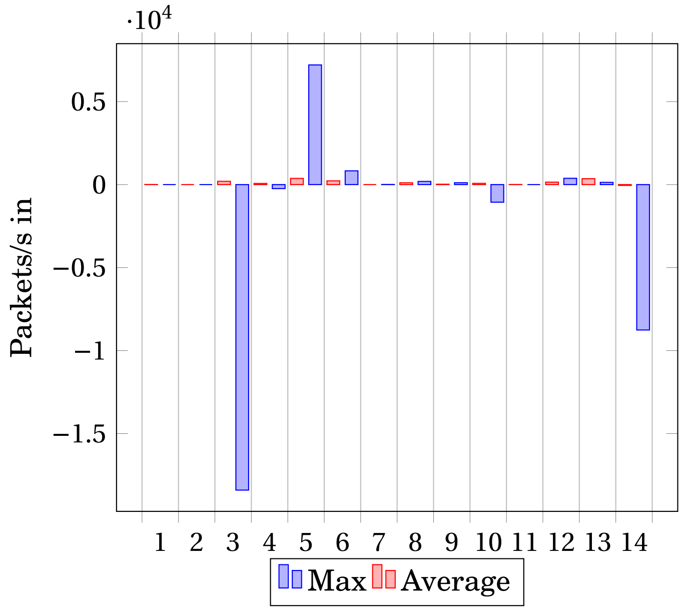


Figure 5.15: Bandwidth graph Packets/s in, Program running (baseline subtracted)

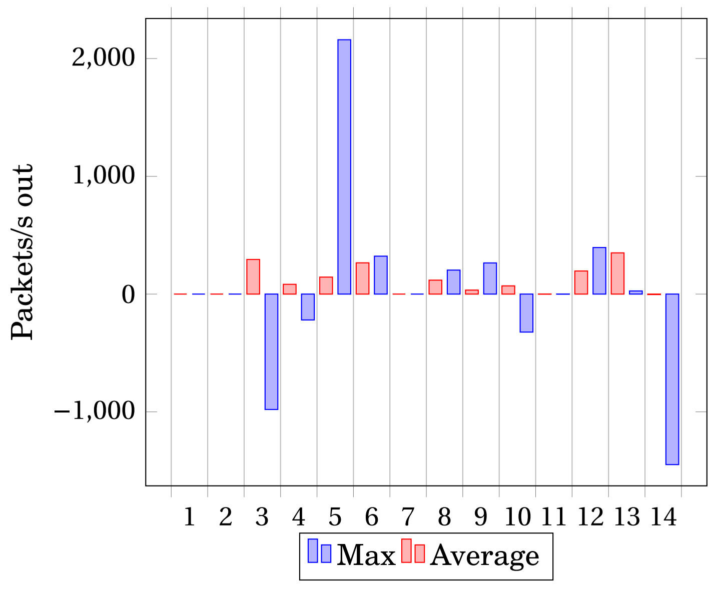


Figure 5.16: Bandwidth graph Packets/s out, Program running (baseline subtracted)

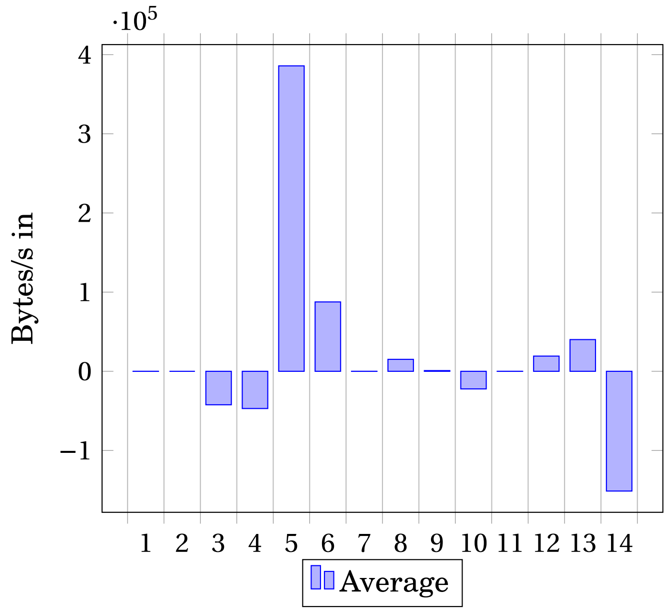


Figure 5.17: Bandwidth graph Bytes/s in, Program running (baseline subtracted, only showing average)

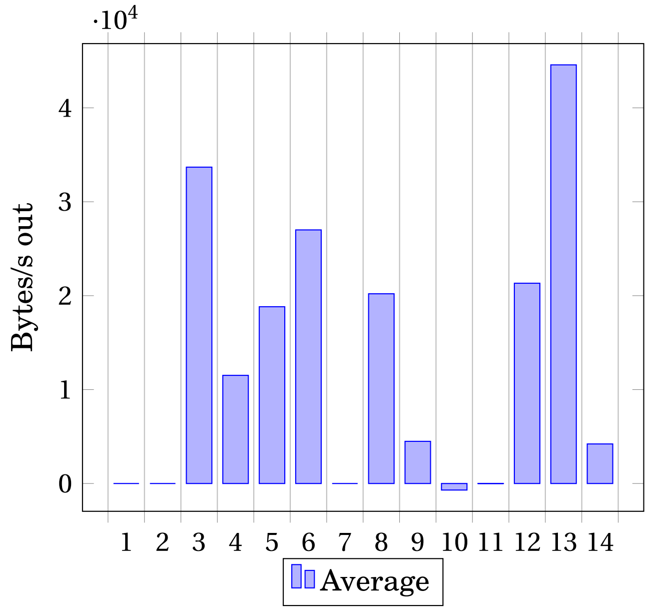


Figure 5.18: Bandwidth graph Bytes/s out, Program running (baseline subtracted, only showing average)

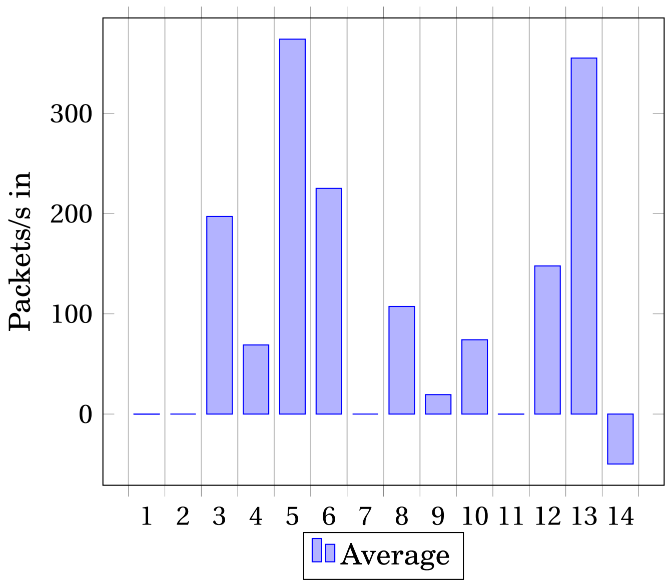


Figure 5.19: Bandwidth graph Packets/s in, Program running (baseline subtracted, only showing average)

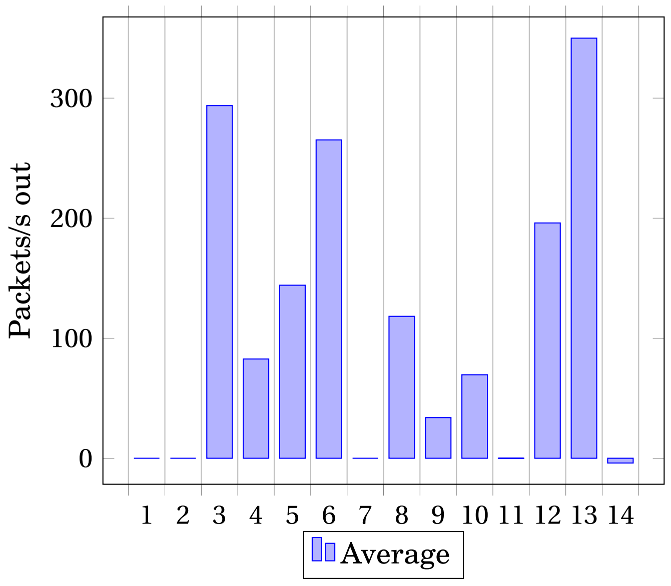


Figure 5.20: Bandwidth graph Packets/s out, Program running (baseline subtracted, only showing average)

5.2 CPU and Memory usage

This section presents and comments on the results of the testing done for the CPU and Memory on the nodes listed in Table 5.1, at page 33. The research metrics used for this testing is listed in section 3.3.1, in Table 3.4 at page 22.

5.2.1 CPU

When it comes to the monitoring of the CPU usage, I was kind of dependent on getting a baseline for the CPU usage, so I had something to compare the data I'd get from monitoring the CPU while the Program was running on the nodes. Unfortunately, all the CPU's where at idle when I was doing the testing an monitoring, so this section will just be to take a look at how the CPU behaved after the program was running, and look at the differences between the nodes and discuss them.

Let us start with the chart shown in Figure 5.21, at page 49. As we can see, most of the nodes peaked at between 2% and 7% CPU usage. There are two exceptions however. Node 2 peaks at 13.3% CPU usage, and node 6 peaks at a whole 36% CPU usage. The one going up to 13.3% usage isn't that bad, but the one going up to 36% usage is a pretty significant difference from the other nodes. Keep in mind that all of these nodes run the exact same program, and are pinging the exact same nodes as everyone else. I thought first that it might be because of geographical distance or maybe it had unusually long round-trip times to the other nodes, but I didn't see anything unusual when I checked those to metrics. So there are basically two things I can think of that could make this happen:

1. This one spike is just an anomaly, and the CPU usage was much lower for the rest of the monitoring.

2. The CPU at this node is just that much slower than the CPU at the other nodes.

Lets look at the first one of those possibilities. If we take a look at the chart in Figure 5.22, at page 49, we can take a look at the average CPU usage over the period that i monitored for. Here we see that all of the nodes are averaging from 2% to 8% CPU usage, except from node 6 that is averaging 18% CPU usage. So it seems that node 6 still is using a lot of the CPU. I took a look at the individual CPU usages of node 6, and I saw that the CPU usage jumped a lot up and down between 8% and 36% CPU usage over the time that i monitored it. This does indicate that the CPU of that node is just that much slower than for the other nodes, because the other nodes did also vary in CPU usage, only that they varied inside a much smaller area.

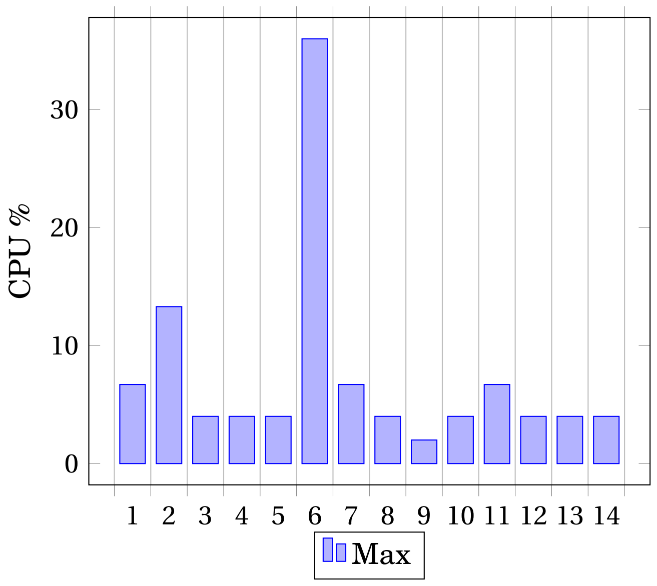


Figure 5.21: Max CPU % graph with Program running

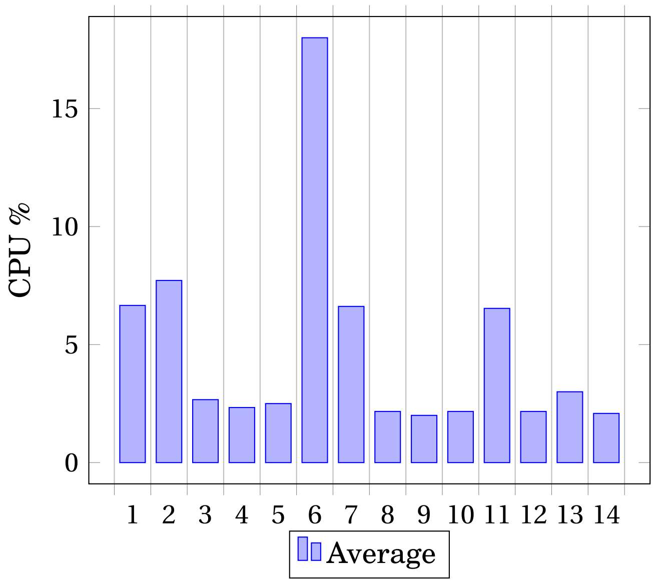


Figure 5.22: Average CPU % graph with Program running

5.2.2 Memory

For the monitoring of the Memory usage, I was able to do a baseline test of the Memory usage when just the Emerald software was running on all of the nodes. The data collected while monitoring the baseline can be seen in the chart in Figure 5.23, at page 51. As we can see that the nodes kept a pretty low memory usage during the monitoring of just the Emerald software running. The nodes with the lowest usage had a 0% average memory usage, while the two nodes with the highest usage had a 0.7% average usage. The 5 nodes that are not using 0% or 0.7% are using 0.1% of memory in average. If we take a look at Table 5.2, we can see the total amount of memory each of the 14 test nodes has. If we then use the % of memory usage we have in Figure 5.23 and multiply that with the values we have in Table 5.2, we can calculate the approximate KB of memory each node uses in average. This calculation has been done, and are listed in Table 5.3. As we can see from that table, all the nodes that didn't have the average memory to 0%, are using around the same amount of memory which makes sense, since they are running the same software. The nodes that are listed with 0 KB are also probably using around the same amount of memory in average as the other nodes, its just that they have so much total memory that the percentage was rounded down to 0% instead of up to 0.1%.

Now lets take a look at the chart in Figure 5.24, at page 51. This chart shows the average memory usage after the test Program has been started. As you can see, one of the nodes that was up in 0.7% has now fallen down to 0.1%, while the other node that was at 0.7% has gone up to 1.1%. We can also see that node 14 has gone up from 0% to 0.1%. This kind of confirms my conclusion in the last paragraph, as it seems that for the baseline testing the average memory usage laid just under 4000 KB, while it now has peaked over 4000 KB, meaning the average amount of memory used is now closer to 0.1 %, of the 8312160 KB of memory node 14 has, than 0%.

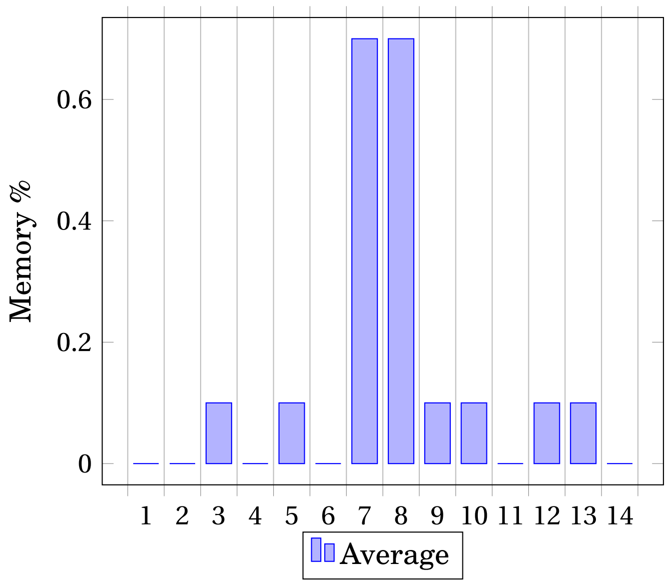


Figure 5.23: Memory % graph baseline

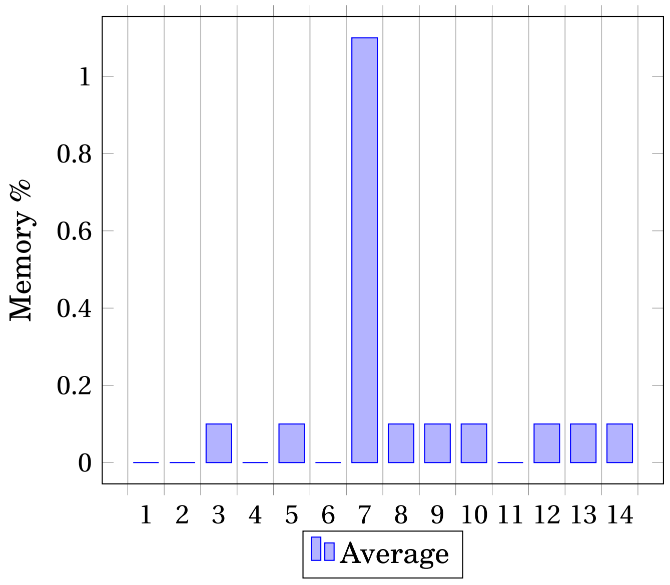


Figure 5.24: Memory % graph with Program running

Node nr.	Hostname	KB Memory
1	planetlab1.cs.uit.no	90071992
2	cse-yellow.cse.chalmers.se	90071992
3	planetlab3.cesnet.cz	4151704
4	planetlab2.inf.ethz.ch	15839832
5	planetlab4.mini.pw.edu.pl	4147476
6	planetlab2.cs.ubc.ca	12456540
7	ple4.planet-lab.eu	524288
8	planetlab1.pop-mg.rnp.br	4064976
9	planetlab1.comp.nus.edu.sg	4151704
10	planetlab-2.calpoly-netlab.net	4146528
11	planetlabeu-1.tssg.org	90071992
12	planetlab2.pop-pa.rnp.br	4142356
13	node2.planetlab.mathcs.emory.edu	4069716
14	pl1.rcc.uottawa.ca	8312160

Table 5.2: List of test nodes with total amount of memory in KB while Program is running.

Node nr.	Hostname	KB Memory
1	planetlab1.cs.uit.no	0
2	cse-yellow.cse.chalmers.se	0
3	planetlab3.cesnet.cz	4151.704
4	planetlab2.inf.ethz.ch	0
5	planetlab4.mini.pw.edu.pl	4147.476
6	planetlab2.cs.ubc.ca	0
7	ple4.planet-lab.eu	3670.016
8	planetlab1.pop-mg.rnp.br	28454.832
9	planetlab1.comp.nus.edu.sg	4151.704
10	planetlab-2.calpoly-netlab.net	4146.528
11	planetlabeu-1.tssg.org	0
12	planetlab2.pop-pa.rnp.br	4142.356
13	node2.planetlab.mathcs.emory.edu	4069.716
14	pl1.rcc.uottawa.ca	0

Table 5.3: List of test nodes with amount of memory used in KB at baseline.

Node nr.	**Hostname**	**KB Memory**
1	planetlab1.cs.uit.no	0
2	cse-yellow.cse.chalmers.se	0
3	planetlab3.cesnet.cz	4151.704
4	planetlab2.inf.ethz.ch	0
5	planetlab4.mini.pw.edu.pl	4147.476
6	planetlab2.cs.ubc.ca	0
7	ple4.planet-lab.eu	5767.168
8	planetlab1.pop-mg.rnp.br	4064.976
9	planetlab1.comp.nus.edu.sg	4151.704
10	planetlab-2.calpoly-netlab.net	4146.528
11	planetlabeu-1.tssg.org	0
12	planetlab2.pop-pa.rnp.br	4142.356
13	node2.planetlab.mathcs.emory.edu	4069.716
14	pl1.rcc.uottawa.ca	8312.160

Table 5.4: List of test nodes with total amount of memory in KB.

Chapter 6

Discussion

In this chapter I will discuss the test results I found in Chapter 6. The goal of this chapter is to give an overview of the the results I found, answer the research questions in section 1.1, and also look at what I could have done different when it came to the testing scenarios.

The thesis considers the following metrics for the testing done in Chapter 6:

- Round-trip time (RTT)
- Geographical distance
- Incoming bytes/s on the bandwidth
- Outgoing bytes/s on the bandwidth
- Incoming packets/s on the bandwidth
- Outgoing packets/s on the bandwidth
- CPU usage in %
- Memory usage in %
- Memory usage in byte

For the Bandwidth I found that the tool I used for testing might not have been the best tool to test with. Since I couldn't use the tool to specify a process or program to monitor, I had to monitor the entire bandwidth of the node. This bandwidth is shared with all other users connected to the node, meaning the measurements I did wasn't precise enough. On the other hand, this is how it always will be in a real world environment. A big part of this thesis was to do the testing in realistic circumstances, and in the real world, a process on a computer will share the bandwidth with all the other processes on it. The positive thing I found was that since I didn't notice a big increase in the bandwidth traffic, the program couldn't really have such a big impact one the bandwidth capacity on the nodes. This was also kind of expected since all the program really does is pinging all the nodes on the network. A ping doesn't use that much bandwidth, at least not compared to the upload and download speeds that most servers use now a days.

When it comes to the CPU and Memory testing, the tool I used worked pretty much just as I wanted it to. I was able to choose which process I wanted to monitor, and the results I got from the tool seemed to be really accurate. When it comes to the CPU monitoring, it was a bit difficult to get that much out of the average and max CPU usage, when I didn't have a baseline for the CPU usage on the computers to go after. We did however find out that one of the computers used for the testing, seemingly had a slower CPU than the rest of the computers I did the tests with. It doesn't seem like the CPU was a bottleneck in any sense, as we didn't see any special outcome from that node on the other tests I did, but it might had been a bottleneck if I had been able to increase the number of nodes I could get Emerald software to run on.

Another note on the CPU and Memory testing is that I could have used one other metric. The load average is a metric that shows the amount of computational work the system performs. For Linux computers, this is the number of processes in the R and D state at a given time:

- R state: Processes that are running or runnable. [22]
- D state: Processes that are in uninterruptible sleep. This is usually I/O processes. [22]

This could have given me even more information about how hard at work the different nodes where under the load of the program, but unfortunately I didn't think of that metric before it was too late.

Chapter 7

Conclusive Remarks

7.1 Research findings

In this thesis, I have looked at the affect my network map had on a group of nodes, in a realistic environment. All in all, the Network Map I created worked as it should. It spread it self to all available nodes on the network, and from there started to ping all available nodes. The findings was pretty much as expected, as the Network Map didn't have a noticeable affect on neither the bandwidth, the CPU or the memory on any of the Nodes.

It would have probably been better to test the program on a larger amount of nodes then what I got to do, but that was unfortunately out of my control. The load average would have also been a good metric to use for the affect on the nodes hardware, instead of just the CPU and memory monitoring.

Bibliography

[1] Domenico Amalfitano, Anna Fasolino and Porfirio Tramontana. *Rich Internet Application Testing Using Execution Trace Data*. Apr. 2010.

[2] Luciano Barbosa and Juliana Freire. *Siphoning Hidden-Web Data through Keyword-Based Interfaces.* Jan. 2004.

[3] Kamara Benjamin et al. 'A Strategy for Efficient Crawling of Rich Internet Applications'. In: *Proceedings of the 11th International Conference on Web Engineering*. ICWE'11. Paphos, Cyprus: Springer-Verlag, 2011, pp. 74–89. ISBN: 978-3-642-22232-0. URL: http://dl.acm.org/citation.cfm?id=2027776.2027784.

[4] SUPRIYO BISWAS. *A Guide to the Linux "Top" Command*. 2019. URL: https://www.booleanworld.com/guide-linux-top-command/ (visited on 02/04/2019).

[5] Paolo Boldi et al. 'UbiCrawler: A Scalable Fully Distributed Web Crawler'. In: 34 (July 2004).

[6] Sergey Brin and Lawrence Page. 'The Anatomy of a Large-scale Hypertextual Web Search Engine'. In: *Comput. Netw. ISDN Syst.* 30.1-7 (Apr. 1998), pp. 107–117. ISSN: 0169-7552. DOI: 10.1016/S0169-7552(98)00110-X. URL: http://dx.doi.org/10.1016/S0169-7552(98)00110-X.

[7] A. Broido, E. Nemeth and k. claffy k. 'Internet Expansion, Refinement, and Churn'. In: *European Transactions on Telecommunications* 13.1 (Jan. 2002), pp. 33–51.

[8] Suryakant Choudhary et al. 'Building Rich Internet Applications Models: Example of a Better Strategy'. In: *Web Engineering*. Ed. by Florian Daniel, Peter Dolog and Qing Li. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 291–305. ISBN: 978-3-642-39200-9.

[9] C. Duda et al. 'AJAX Crawl: Making AJAX Applications Searchable'. In: *2009 IEEE 25th International Conference on Data Engineering*. Mar. 2009, pp. 78–89. DOI: 10.1109/ICDE.2009.90.

[10] Jenny Edwards, Kevin Mccurley and John Tomlin. 'An Adaptive Model for Optimizing Performance of an Incremental Web Crawler'. In: (Apr. 2001).

[11] A. M. Fard and A. Mesbah. 'Feedback-directed exploration of web applications to derive test models'. In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 278–287. DOI: 10.1109/ISSRE.2013.6698880.

[12] Free Software Foundation. *GNU General Public License, version 2*. 2017. URL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html (visited on 02/04/2019).

[13] Volker Gropp. *bwm-ng*. 2019. URL: https://github.com/vgropp/bwm-ng (visited on 02/04/2019).

[14] Allan Heydon and Marc Najork. 'Mercator: A Scalable, Extensible Web Crawler'. In: 2 (July 1999).

[15] Bradley Huffaker et al. 'k claffy. Distance metrics in the internet'. In: *in IEEE International Telecommunications Symposium*. 2002, pp. 200–2.

[16] Hsin-Tsang Lee et al. 'IRLbot: Scaling to 6 Billion Pages and Beyond'. In: 3 (Jan. 2008), p. 8.

[17] Stephen W. Liddle et al. 'Extracting Data behind Web Forms'. In: *Advanced Conceptual Modeling Techniques*. Ed. by Antoni Olivé, Masatoshi Yoshikawa and Eric S. K. Yu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 402–413. ISBN: 978-3-540-45275-1.

[18] Netcraft Ltd. *NetCraft Web Server Survey*. 2018. URL: https://news.netcraft.com/archives/category/web-server-survey/ (visited on 16/05/2018).

[19] J. Lu et al. 'An Approach to Deep Web Crawling by Sampling'. In: *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Vol. 1. Dec. 2008, pp. 718–724. DOI: 10.1109/WIIAT.2008.392.

[20] Clifford Lynch. 'SEARCHING THE INTERNET'. In: *Scientific American* 276.3 (1997), pp. 52–56. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/24993652.

[21] Seyed M. Mirtaheri et al. 'A Brief History of Web Crawlers'. In: (May 2014).

[22] Marek. *Linux process states*. 2012. URL: https://idea.popcount.org/2012-12-11-linux-process-states/ (visited on 02/04/2019).

[23] A. Mesbah, E. Bozdag and A. v. Deursen. 'Crawling AJAX by Inferring User Interface State Changes'. In: *2008 Eighth International Conference on Web Engineering*. July 2008, pp. 122–134. DOI: 10.1109/ICWE.2008.24.

[24] John Morris. *Data Structures and Algorithms: Red-Black Trees*. 1998. URL: https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html (visited on 18/05/2018).

[25] H. M. Levy N. C. Hutchinson E. Jul. *The Emerald Programming Language*. URL: http://www.emeraldprogramminglanguage.org/ (visited on 10/04/2019).

[26] OneLab. *PlanetLab Europe Technical Overview*. 2006. URL: https://www.planet-lab.eu/files/PlanetLab__Tech_Overview.pdf (visited on 10/04/2019).

[27] Z. Peng et al. 'Graph-Based AJAX Crawl: Mining Data from Rich Internet Applications'. In: *2012 International Conference on Computer Science and Electronics Engineering*. Vol. 3. Mar. 2012, pp. 590–594. DOI: 10.1109/ICCSEE.2012.38.

[28] The Trustees of Princeton University. *Joining PlanetLab*. 2018. URL: https://www.planet-lab.org/joining (visited on 01/06/2018).

[29] The Trustees of Princeton University. *PlanetLab*. 2018. URL: https://www.planet-lab.org/ (visited on 01/06/2018).

[30] Sriram Raghavan and Hector Garcia-Molina. 'Crawling the Hidden Web'. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 129–138. ISBN: 1-55860-804-4. URL: http://dl.acm.org/citation.cfm?id=645927.672025.

[31] V. Shkapenyuk and T. Suel. 'Design and implementation of a high-performance distributed Web crawler'. In: *Proceedings 18th International Conference on Data Engineering*. 2002, pp. 357–368.

[32] P. Zerfos, J. Cho and A. Ntoulas. 'Downloading textual hidden web content through keyword queries'. In: *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '05)*. June 2005, pp. 100–109. DOI: 10.1145/1065385.1065407.