



Edgard Lima



Edgard Lima

Distributed van Emde Boas tree

Adviser: Ph.D. Guilherme Vilar

Co-Adviser: Ph.D. Tiago Alessandro Espínola Ferreira

Master's Dissertation presented to the Graduate Program in Applied Informatics program at Federal Rural University of Pernambuco, as a partial requirement to earn the Master of Science in Applied Informatics degree.

Recife

August, 2017

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas da UFRPE
Biblioteca Central, Recife-PE, Brasil

L732d Lima, Edgard Nicéas Arcoverde Gusmão
Distributed van Emde Boas tree / Edgard Nicéas Arcoverde
Gusmão Lima. – 2017.
179 p. : il.

Orientador: Guilherme Vilar.
Coorientador: Tiago Alessandro Espínola Ferreira.
Dissertação (Mestrado) – Universidade Federal Rural de
Pernambuco, Programa de Pós-Graduação em Informática Aplicada,
Recife, BR-PE, 2017.
Inclui referências e apêndice(s).

1. Algoritmos 2. Estrutura de dados 3. Sistemas distribuídos
4. Van Emde Boas tree 5. Algorithms 6. Data structure 7. Distributed
systems 8. Big Data I. Vilar, Guilherme, orient. II. Ferreira, Tiago
Alessandro Espínola, coorient. III. Título

CDD 004



Federal Rural University of Pernambuco
Department of Statistics and Informatics
Graduate Program in Applied Informatics

Distributed van Emde Boas Tree

Edgard Nicéas Arcoverde Gusmão Lima

Dissertation deemed appropriate for obtaining the title of Master of Science in Applied Informatics. Presented and unanimously approved by the board in August 02, 2017.

Adviser: Ph.D. Guilherme Vilar

Co-Adviser: Ph.D. Tiago Alessandro Espínola Ferreira

Board:

Ph.D. Kátia Guimarães (CIn/UFPE)

Ph.D. Fernando Aires (PPGIA/UFRPE)

Ph.D. Jones Albuquerque (PPGIA/UFRPE)

Recife
August, 2017

To

my lovely children, that are definitively the most important things in my life, and what gives me real motivation to keep always moving forward.

my parents, who always educated, believed and supported me, and therefore deserves to see outcomings from me to my relatives and to the society

Acknowledgements

After 13 years of my Bachelor's diploma and professional experience, I have decided to come back to academy with three goals in mind. The first was to have a chance to develop a ultimate solution that could be launched into the market, the second was my passion for theory and research, and the last was to have a diploma so that I could become a collage teacher/researcher in near future.

To be honest, I wasn't expecting to learn too much from my master course. I thought I already had enough solid theory knowledge, professional experience and maturity for a master program.

Fortunately on the first quarter of the course I was proved to be completely wrong. I had so much to learn from the Master's program. Definitely these two years of master's program were the most profitable ones in my career.

First, I started to see things completely different, things in the way science see, it is a so shine and bright way that I won't even try to describe it here, luckily are the ones with science/researcher eyes.

Second, I learned too much from courses of this master program.

And last but not least, I learned a lot from amazing people I met.

For this all I earned, I would like to sincerely thank so much Professors Tiago Ferreira, Jorge Correia, and Guilherme Vilar. Without them, this research wouldn't be possible.

Finally, I would like to thank all my previous teachers and relatives that certainly have contributed somehow to this work.

Abstract

A very important computational problem is how to organize information. In particular, the contemporaneous world has been presented with a new class of problem, to handle a very large amount of data, called Big Data Problem. Typical data structures have $O(\lg n)$ time cost, where n is the size of the database and \lg is the binary logarithm (\log_2). However, if n is a very large number, like a *googol* (10^{100}) or a *googolplex* ($10^{10^{100}}$), data structures of $O(\lg n)$ still have a hard cost to solve a problem. To address this problem, a data structure named van Emde Boas Tree (vEBt) could be used. A vEBt has $O(\lg \lg U)$ worst case time cost (where U is the data universe size), but this low cost demands a lot of memory. The size of memory to implement a typical vEBt is so big that there is no any today's machine that could just instantiate an empty vEBt of 2^{128} universe size. This research proposes a strategy to implement a class of distributed van Emde Boas tree able to work with huge data mass (big data). The time cost still is $O(\lg \lg U)$ and a computer cluster can be used to run this distributed vEBt, where each cluster's node needs to have very little memory. As we show on experiments, with our solution, now even cheap 4 GB machines can handle up to vEB($2^{2^{17}} = 2^{131,072} \approx 10^{39,457}$) trees, which is much bigger than a *googol* (10^{100}).

Keywords: Algorithms. data structure. distributed systems. van Emde Boas tree. big data.

Resumo

Organizar informações é um problema computacional muito importante. Em particular, no mundo contemporâneo, existe uma nova classe de problemas, relativa ao tratamento de gigantescas quantidades de dados, conhecida como *Big Data Problem*. Estruturas de dados convencionais apresentam custo de tempo de $O(\lg n)$, onde n é quantidade de elementos na base de dados e \lg é o logaritmo binário (\log_2). Contudo, se n é um número muito grande, como um *googol* (10^{100}) ou um *googolplex* ($10^{10^{100}}$), estruturas de dados de custo $O(\lg n)$ ainda têm um alto custo para resolver o problema. Para tratar desta questão, uma estrutura de dados chamada *van Emde Boas Tree* (vEBt) poderia ser utilizada. A vEBt tem custo em tempo $O(\lg \lg U)$ no seu pior caso (onde U é o tamanho de universo), mas este baixo custo demanda muita memória. A quantidade de memória para implementar uma vEBt convencional é tão grande que não existe nenhuma máquina nos dias atuais que poderia sequer instanciar uma vEBt vazia com universo de 2^{128} . Nesta pesquisa foi proposta uma estratégia para implementar uma classe de árvore de *van Emde Boas* distribuída capaz de trabalhar com grandes massas de dados (*big data*). O seu custo permanece $O(\lg \lg U)$ e um cluster de máquinas pode ser utilizado para executar esta vEBt distribuída, onde cada nó do cluster precisa ter apenas uma pequena quantidade de memória. Como foi mostrado em experimentos, com a solução proposta, mesmo simples máquinas com 4 GB podem indexar árvores vEB($2^{2^{17}} = 2^{131,072} \approx 10^{39,457}$), que é bem maior que um *googol* (10^{100}).

Palavras-chave: algoritmos. estrutura de dados. sistemas distribuídos. árvore de van Emde Boas. big data.

List of Figures

Figure 1 – Big data interest in Google trends explorer	27
Figure 2 – van Emde Boas tree	31
Figure 3 – van Emde Boas tree cluster array	33
Figure 4 – Breaking an array into a vEB node recursively	34
Figure 5 – The complete vEB tree	35
Figure 6 – OSI vs TCP/IP reference model	43
Figure 7 – TCP/IP layers and typical protocols	44
Figure 8 – UDP header	44
Figure 9 – A vEB tree with universe 2^8 using a proxy Root	49
Figure 10 – The same vEB tree, now with universe 2^{16}	50
Figure 11 – Proxy Pattern applied to our vEB tree	52
Figure 12 – Mapping GUID into IPv6 address	55
Figure 13 – vEB node mapping into IPv6 addresses.	56
Figure 14 – Sequence Diagram of the Automatic IPv6 solution	57
Figure 15 – Sequence diagram of inserting a node in Network-Agnostic solution. . .	67
Figure 16 – Sequence diagram of inserting a node with Cheater.	70
Figure 17 – <i>insert()</i> mean time by nodes in a dense vEB(2^{24}).	78
Figure 18 – <i>insert()</i> average depth by nodes in a dense vEB(2^{24}).	78
Figure 19 – <i>insert()</i> depths count, of last 2k elements, by nodes in a dense vEB(2^{24}). .	79
Figure 20 – <i>insert()</i> depths time by nodes in a dense vEB(2^{24}).	79
Figure 21 – <i>insert()</i> depth average time of a dense vEB(2^{24}).	80
Figure 22 – <i>successor()</i> mean time by nodes in a dense vEB(2^{24}).	81
Figure 23 – <i>successor()</i> average depth by nodes in a dense vEB(2^{24}).	82
Figure 24 – <i>successor()</i> depths count, of last 2k elements, by nodes in a dense vEB(2^{24}). .	82
Figure 25 – <i>successor()</i> depths time by nodes in a dense vEB(2^{24}).	83
Figure 26 – <i>successor()</i> depth average time of a dense vEB(2^{24}).	83
Figure 27 – <i>predecessor()</i> mean time by nodes in a dense vEB(2^{24}).	85
Figure 28 – <i>predecessor()</i> average depth by nodes in a dense vEB(2^{24}).	85
Figure 29 – <i>predecessor()</i> depths count, of last 2k elements, by nodes in a dense vEB(2^{24}).	86
Figure 30 – <i>predecessor()</i> depths time by nodes in a dense vEB(2^{24}).	86
Figure 31 – <i>predecessor()</i> depth average time of a dense vEB(2^{24}).	87
Figure 32 – <i>search()</i> mean time by nodes in a dense vEB(2^{24}).	88
Figure 33 – <i>search()</i> average depth by nodes in a dense vEB(2^{24}).	88
Figure 34 – <i>search()</i> depths count, of last 2k elements, by nodes in a dense vEB(2^{24}). .	89
Figure 35 – <i>search()</i> depths time by nodes in a dense vEB(2^{24}).	89

Figure 36 – <i>search()</i> depth average time of a dense vEB(2^{24}).	90
Figure 37 – <i>remove()</i> mean time by nodes in a dense vEB(2^{24}).	91
Figure 38 – <i>remove()</i> average depth by nodes in a dense vEB(2^{24}).	91
Figure 39 – <i>remove()</i> depths count, of last 2k elements, by nodes in a dense vEB(2^{24}).	92
Figure 40 – <i>remove()</i> depths time by nodes in a dense vEB(2^{24}).	92
Figure 41 – <i>remove()</i> depth average time of a dense vEB(2^{24}).	93
Figure 42 – <i>insert()</i> mean time by nodes in a sparse vEB($2^{2^{17}}$).	94
Figure 43 – <i>insert()</i> average depth by nodes in a sparse vEB($2^{2^{17}}$).	95
Figure 44 – <i>insert()</i> depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).	95
Figure 45 – <i>insert()</i> depths time by nodes in a sparse vEB($2^{2^{17}}$).	96
Figure 46 – <i>insert()</i> depth average time of a sparse vEB($2^{2^{17}}$).	96
Figure 47 – <i>successor()</i> mean time by nodes in a sparse vEB($2^{2^{17}}$).	98
Figure 48 – <i>successor()</i> average depth by nodes in a sparse vEB($2^{2^{17}}$).	99
Figure 49 – <i>successor()</i> depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).	99
Figure 50 – <i>successor()</i> depths time by nodes in a sparse vEB($2^{2^{17}}$).	100
Figure 51 – <i>successor()</i> depth average time of a sparse vEB($2^{2^{17}}$).	100
Figure 52 – <i>predecessor()</i> mean time by nodes in a sparse vEB($2^{2^{17}}$).	101
Figure 53 – <i>predecessor()</i> average depth by nodes in a sparse vEB($2^{2^{17}}$).	102
Figure 54 – <i>predecessor()</i> depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).	102
Figure 55 – <i>predecessor()</i> depths time by nodes in a sparse vEB($2^{2^{17}}$).	103
Figure 56 – <i>predecessor()</i> depth average time of a sparse vEB($2^{2^{17}}$).	103
Figure 57 – <i>search()</i> mean time by nodes in a sparse vEB($2^{2^{17}}$).	104
Figure 58 – <i>search()</i> average depth by nodes in a sparse vEB($2^{2^{17}}$).	105
Figure 59 – <i>search()</i> depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).	105
Figure 60 – <i>search()</i> depths time by nodes in a sparse vEB($2^{2^{17}}$).	106
Figure 61 – <i>search()</i> depth average time of a sparse vEB($2^{2^{17}}$).	106
Figure 62 – <i>remove()</i> mean time by nodes in a sparse vEB($2^{2^{17}}$).	108
Figure 63 – <i>remove()</i> average depth by nodes in a sparse vEB($2^{2^{17}}$).	108
Figure 64 – <i>remove()</i> depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).	109
Figure 65 – <i>remove()</i> depths time by nodes in a sparse vEB($2^{2^{17}}$).	109
Figure 66 – <i>remove()</i> depth average time of a sparse vEB($2^{2^{17}}$).	110
Figure 67 – Time taken by registry.find() on 2000 keys for each <i>height</i> .	112
Figure 68 – Time taken by registry.find() on 2000 keys for keys of different sizes in bits.	112
Figure 69 – Trees class diagram.	128
Figure 70 – Clusters class diagram.	129
Figure 71 – RPC class diagram.	130

Figure 72 – RPC Client PoV.	135
Figure 73 – RPC Host PoV.	136
Figure 74 – Typical log output.	137
Figure 75 – TCP dump.	138
Figure 76 – Trace log of a Insert operation.	139
Figure 77 – Automated e-mail.	139
Figure 78 – Valgrind.	140
Figure 79 – Bitbucket git repository.	141
Figure 80 – Statistic.	141
Figure 81 – Tmux session.	145

List of Tables

Table 1 – Internet Numbers - < http://www.internetlivestats.com/ >	27
Table 2 – Direct-address vs vEB time cost.	32
Table 3 – Memory cost of a vEB tree.	41
Table 4 – Y value mapped into IPv6 address	56
Table 5 – Y value of Network-Agnostic node mapping	60
Table 6 – Comparison between DHT implementations	71
Table 7 – vEB(2^{24}) depths	74
Table 8 – vEB($2^{2^{17}}$) depths	74
Table 9 – Number of keys by level of a vEB(65536)	77
Table 10 – Machines	143
Table 11 – Machines/Configuration	145

Contents

1	INTRODUCTION	21
1.1	Justification	21
1.1.1	van Emde Boas trees	22
1.2	Objectives	23
1.2.1	Main Objective	23
1.2.2	Specific Objectives	23
1.3	Document Structure	23
2	MOTIVATION	25
2.1	Research problem	25
2.1.1	The three challenges	26
2.2	Research Motivation	26
3	THEORETICAL FOUNDATIONS	31
3.1	van Emde Boas trees	31
3.2	Computer Networks Review	41
3.2.1	Reference Models	41
3.2.2	UDP	43
3.2.3	Unicast	45
3.2.4	Multicast	45
3.3	Distributed Systems	45
4	METHODOLOGY	49
4.1	Challenge 1: Allow a vEB tree to increase its universe	49
4.2	Challenge 2: How to make a vEB cluster with minimal memory cost	53
4.3	Challenge 3: Distribute the vEB tree	54
4.3.1	Automatic IPv6 addressing	54
4.3.2	Network-Agnostic	59
4.3.2.1	Minimalist UDP implementation	63
4.3.2.1.1	Fundamental building blocks	63
4.3.2.1.2	Message Identifier	64
4.3.2.1.3	Marshalling	65
4.3.2.1.4	Framing	65
4.3.2.1.5	Request-Acknowledge-Reply-Acknowledge	65
4.3.2.1.6	Retry	67
4.3.2.1.7	Discarding duplicate and History	68

4.3.2.1.8	Flow control and Congestion control	68
4.3.2.1.9	Cheater	69
5	EXPERIMENTS	73
5.1	Planning	73
5.1.1	Experiment 01 - Dense tree	74
5.1.2	Experiment 02 - Sparse tree	76
5.2	Results	77
5.2.1	Experiment 01 - Dense tree	77
5.2.1.1	Insert	77
5.2.1.2	Successor	81
5.2.1.3	Predecessor	84
5.2.1.4	Search	87
5.2.1.5	Remove	90
5.2.2	Experiment 02 - Sparse tree	94
5.2.2.1	Insert	94
5.2.2.2	Successor	97
5.2.2.3	Predecessor	101
5.2.2.4	Search	104
5.2.2.5	Remove	107
5.2.3	Experiments: last analysis	110
6	CONCLUSION AND FUTURE WORK	113
6.1	Future work	114
	BIBLIOGRAPHY	117
	APPENDIX	123
	APPENDIX A – PROGRAM OPTIONS	125
	APPENDIX B – SOFTWARE ARCHITECTURE	127
	APPENDIX C – REPRODUCING THE EXPERIMENTS	143
C.1	General preparation	143
C.2	Experiment 01	146
C.3	Experiment 02	148
C.4	Correctness test	149
	APPENDIX D – SOURCE CODE	151

1 Introduction

1.1 Justification

The age of Big Data has arrived (LABRINIDIS; JAGADISH, 2012), (MERVIS, 2012), (COMMUNITY... , 2008), (ELDAWY; MOKBEL, 2015). Every day, 2.5 quintillion bytes of data are created, and 90 percent of the data in the world today were produced within the past two years (IBM,)(SINTEF, 2013).

There are numerous definitions of Big Data (MAURO; GRECO; GRIMALDI, 2015)(HASHEM et al., 2015)(WARD; BARKER, 2013). And all of them directly or indirectly refers to the V's of Big Data, "volume", "velocity" and "variety". Volume refers to data size and how it grows. Velocity refers to the speed data is generated (written) and retrieved (read). And Variety refers to heterogeneity embracing different types and different sources of data. Ward and Barker (WARD; BARKER, 2013) states that Big Data as term describing the storage and analysis of large and or complex data sets using a series of techniques, with NoSQL being one of those tools.

NoSQL (commonly referred to as "Not Only SQL") represents a completely different framework of databases that allows high-performance processing of information at massive scale. In other words, it is a database infrastructure that has been very well-adapted to the heavy demands of big data. The efficiency of NoSQL can be achieved, because, unlike relational databases that are highly structured, NoSQL databases are unstructured in nature, trading off stringent consistency requirements for speed and agility. NoSQL centers around the concept of distributed databases, where unstructured data may be stored across multiple processing nodes, and often across multiple servers. This distributed architecture allows NoSQL databases to be horizontally scalable. As data continues to explode, just add more hardware keeps it up, with no slowdown in performance (RANI; KUMAR, 2015).

NoSQL database are distributed key/value table stores or a subclass of it, and provide a lightweight, cost-effective, scalable and available alternative to traditional relational databases. Today, scalable table stores, such as Google BigTable, Amazon Dynamo, Apache HBase, Apache Cassandra, Voldemort, Apache Accumulo and Redis(CHEN et al., 2016), are becoming an essential part of Internet services ¹. They are used for high volume data-intensive applications, such as business analytics and scientific data analysis. In some cases they are available as a cloud service, such as Amazon's SimpleDB and Microsoft's Azure SQL Services, as well as application platforms, as in Google's AppEngine and

¹ <<http://db-engines.com/en/ranking/key-value+store>>

Yahoo's YQL (SEN; FARRIS; GUERRA, 2013).

Such distributed NoSQL solutions are based on Consistent hash, LSM trees, RB-trees, B-trees or B+trees (CORMEN et al., 2009). The ones based on hash has good $O(1)$ time per dictionary operations, *i.e.* *search*, *insert* and *delete*. While the ones based on trees may have $O(\lg n)^2$ time cost for dynamic set operations, *i.e.* *search*, *insert*, *delete*, *successor*, *predecessor*, *minimum* and *maximum* operations.

In this work we propose the grounds of a novel, highly scalable, solution for a NoSQL distributed key/value table store, based on van Emde Boas tree, that performs dynamic set operations in $O(\lg \lg U)^3$ time.

1.1.1 van Emde Boas trees

The van Emde Boas tree has been proposed in 1975 by Peter van Emde Boas (BOAS, 1975), (BOAS, 1977), (BOAS; KAAS; ZIJLSTRA, 1976). It has time cost of $O(\lg \lg U)$ for *search*, *insert*, *delete*, *successor* and *predecessor* operations, and $O(1)$ for *minimum* and *maximum* operations. While it has advantage over regular data structures that runs in $O(\lg n)$ time, it has some drawbacks, like its initial size. Therefore, until now it hasn't been used in Big Data technologies.

However, we believe van Emde Boas tree may have its turn in the world of Big Data and very large databases due to the following advantages.

- Dynamic set operations has the following time cost:
 - Its internal structure, that seems to be convenient for clustering. Once an operation reaches certain node, it does not need the ancestors or sibling nodes anymore to complete its execution;
 - $O(\lg \lg U)$ - *search*, *insert*, *delete*, *successor* and *predecessor*;
 - $O(1)$ - *minimum* and *maximum*;
- It is good for range queries. The cost is the number of elements between the two indexes, multiplied by $O(\lg \lg U)$;
- It has a cache oblivious structure (DEMAINE, 2002);
- We could take advantage of multicore and network paralalism to boost up speed (WANG; LIN, 2007)(KUŁAKOWSKI, 2013).

² "n" is the number of elements present in the tree.

³ "U" is the universe. *i.e.* The maximum number of elements supported by the tree. In a vEBt the elements are represented by integer keys from 0 to U-1

This research is about adapt the van Emde Boas tree to be used as a high scalable Big Data structure.

In Chapter 3 we will see the original van Emde Boas tree in more details, and in Chapter 4 we will explain how it has been distributed.

1.2 Objectives

1.2.1 Main Objective

Design and implement a distributed van Emde Boas tree, keeping its original time cost for dynamic set operations (Subsection 1.1.1), making it suitable to be used as big data structure.

1.2.2 Specific Objectives

In order to achieve the main objective, the following specific goals should be taken:

1. Design a distributed van Emde Boas tree, that holds its time cost, with the following characteristics:
 - a) Increases its size dynamically;
 - b) Allow any machine to run a vEB tree of any size, *i.e.* overcome the initial cluster size limitation;
2. Implement the proposed distributed van Emde Boas Tree;
3. Design and plan correctness and performance experiments;
4. Write test cases and tools for the experiments;
5. Analyze the results and make conclusions.

1.3 Document Structure

This section gives a brief picture of the structured of this document and what is present on each chapter.

Chapter 2, introduces some challenges we are about to face and spices up the motivation for this research.

Chapter 3, explores the theory and foundations of a van Emde Boas tree and reviews few very basic Network and Distributed System concepts.

In chapter 4, at first, we show our initial approach to design the Distributed van Emde Boas tree, and analyze it, exposing its limitations, in special, its scalability restrictions. Then we depict our improved second approach that solves almost all limitations of the previous one, including scalability restrictions, and then we explore and analyze it, also showing its weakness.

Chapter 5 describes the performed experiments and finally analyzes the results.

In chapter 6 we make our final conclusions and bring several ideas for future work.

After that, comes the Bibliography, with all references that have been used as theoretical basis in this research.

And finally, Appendix A documents command line options to run experiments using our testing program. Appendix B explains the software architecture. Appendix C gives detailed instruction on how to setup and reproduce the experiments. And Appendix D has some verbatim copy of some c++ source code, developed during this research, that may be useful to clarify some concepts brought in this document.

2 Motivation

2.1 Research problem

The van Emde Boas tree has been proposed in the 70's (BOAS, 1975)(BOAS, 1977)(BOAS; KAAS; ZIJLSTRA, 1976), since then, despite its low worst-case cost time of $O(\lg \lg U)$, has only been used in theory and academy due to its drawbacks:

- its time cost is $O(\lg \lg U)$, it means the cost is based on the universe size, does not matter the number of elements currently in the tree;
- the constant part of its time cost may be high, making it not worth for non huge amount of data;
- high amount of structural data to support the tree even without carrying any satellite user data, $P(U) = (\sqrt{U} + 1)P(\sqrt{U}) + \Theta(\sqrt{U})$ ¹;
- huge startup memory cost of $\Theta(\sqrt{U})$ to hold its cluster;
- keys must be non-negative integer numbers^{2 3}.

On the other hand, besides the low time cost of operations (Subsection 1.1.1), the vEB tree is cache-oblivious structure (DEMAINE, 2002). A cache-oblivious structure, even unaware of caches levels and sizes, can avoid cache misses. A remote node in a distributed tree could be considered another cache level.

The proposal of this research is to design and implement a distributed van Emde Boas tree without losing the $O(\lg \lg U)$ time cost, capable of instantiate and run arbitrary universe sizes, on cheap distributed machines, and also capable of start at a low universe size and increase it dynamically and efficiently as needed.

By overcoming the huge initial cluster size, and distributing it, we expect to build the grounds of a new solution that could beat the current NoSQL Key-Value Store Databases solutions, that uses RB-Trees², Hashes², B-Trees² or B+Trees² data structure.

¹ $P(U)$ is the size cost of a vEB tree of universe size U , the first term represents the summary plus \sqrt{U} children trees of universe \sqrt{U} and the last term is the size of the cluster array of \sqrt{U} pointers to children trees.

² Sometimes data can be converted into non-negative integers and still preserve the original order, or closely preserve it. As an example, GeoMesa uses a technique called Space Filling curve to convert bi-dimensional coordinates into a integer that can be stored in NoSQL database like HBase.

³ <<http://www.geomesa.org/>>

² Cormen's book(CORMEN et al., 2009)

2.1.1 The three challenges

As we will see in section 2.2, and as stated by Zheng (ZHENG et al., 2015), the common use cases for Key-Value (KV) stores are large scale data-intensive applications as they offer high efficiency, scalability, and availability.

For high efficiency, we trust in van Emde Boas structure itself and optimizations that could be done by taking advantage for multi-core and network parallelism. Although the concurrent implementation of our distributed vEB tree is out of the scope of this research. Availability study is also left out in this research due to time constraints.

Then, the focus of this research is on scalability. How to make a van Emde Boas tree scalable?

The first challenge to make it scalable, is allow it to increase dynamically as the number of elements grows, actually, in case of van Emde Boas trees, as the maximum *element* (or *key*) grows. Since the original proposed van Emde Boas tree has its max number of elements fixed by the time of its creation, it is a limitation that we have to overcome.

The second task, that we need to deal with, is how to make any machine support vEB trees of any size. As we can see from Table 3, an empty vEB(2^{64}) uses 32 Giga Bytes (billion) for its cluster ⁴, and an empty vEB(2^{128}) uses 295 Exa-Bytes (quintillion) just to be instantiated. It is a hard task because we need to replace the cluster array with some other data structure and keep the array time cost of $O(1)$ for *insert*, *delete* and *search* operations on the cluster, to keep the original vEB $O(\lg \lg U)$ time cost for dynamic set operations.

The third challenge, is to make it distributed. It doesn't make any sense thinking of Big Data without think of a distributed structure ⁵. We need to choose a distributed design that couples with the others characteristics of our final tree.

On next Chapters we will come back to these three challenges.

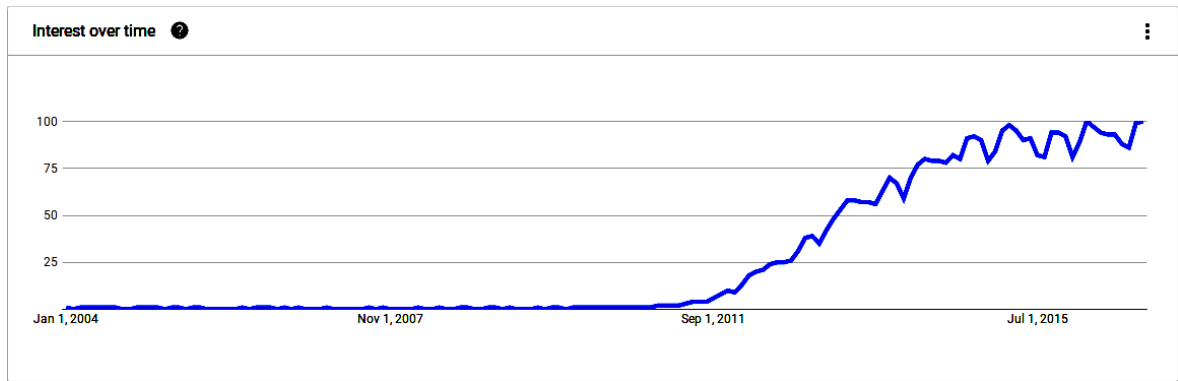
2.2 Research Motivation

The Big Data problem is still an open problem and have to many areas on interest (WANG; YU, 2015), (BELLO-ORGAS; JUNG; CAMACHO, 2016), (ELSHAWI et al., 2015). Digital data are collected at an incredible rate, 2.5 quintillion (2.5×10^{18}) bytes of data generated every day and 90% of the world's data were created in the past 2 years (SINTEF, 2013), (IBM,).

⁴ In this context, *cluster*, has nothing to do with network clusters. *Cluster* is just an contiguous array with pointers to children trees.

⁵ We could change the Operating System kernel to offer distributed memory transparently to processes.

Figure 1 – Big data interest in Google trends explorer



Interest in "Big Data" term search according <http://www.google.com/trends/explore>

The interest for "Big Data" has increased each year. Illustration 1 shows the increasing number of search on Google for the "Big Data" term from 2007 to October 2016.

The website <http://www.internetlivestats.com/> shows some interesting live numbers on Internet, some number are summarized in Table 1:

Internet users in the world	3.48 billion
Number of websites	1 billion
E-mails sent (10 months)	67.9 trillion
Google searches (10 months)	1.4 trillion
Tweets sent (10 months)	410 million
Photos uploaded on Instagram (10 months)	19.2 billion
Skype calls (10 months)	59.8 billion
Internet traffic (10 months)	1 Zetta Bytes

Table 1 – Internet Numbers - <http://www.internetlivestats.com/>

The report of IDC ([PRESS... , 2016](#)) indicates that the marketing of big data was about \$16.1 billion in 2014. Another report of IDC⁶ forecasts that it will grow up to \$32.4 billion by 2017. The reports^{7 8} further pointed out that the marketing of big data will be \$46.34 billion and \$114 billion by 2018, respectively. Even though the marketing values of

That approach is not considered in this research tough.

⁶ <http://www.idc.com/promo/thirdplatform/fourpillars/bigdataanalytics>

⁷ <http://www.eweek.com/database/big-data-market-to-reach-46.34-billion-by-2018.html>

⁸ <https://www.abiresearch.com/press/big-data-spending-to-reach-114-billion-in-2018-100>

big data in these researches and technology reports^{9 10 11 12 13 14} (PRESS... , 2016) are different, these forecasts usually indicate that the scope of big data will be grown rapidly in the forthcoming future.

In addition to business and marketing, from the results of disease control and prevention (MAYER-SCHÖNBERGER; CUKIER, 2014), medicine and health-care (U... , 2015)(LEVIN; WANDERER; EHRENFELD, 2015)(MONTEITH et al., 2015), business intelligence (CHEN; CHIANG; STOREY, 2012), and smart city (KITCHIN,), mining and oil & gas industry (PERRONS; MCAULEY, 2015), we can easily understand that big data is of vital importance everywhere.

Also, recently, data has suddenly become the most interesting element for any kind of scientific analysis. A number of domains, like earthquake simulation, social networking, climate science, astrophysics, bioinformatics (DEDE et al., 2012)(LANARI, 2015), and information retrieval, produce data at massive rate than ever before.

This Big Data has created a hindrance in the development route of both research and industry. Thus, a major tool is required to effectively manage and process this huge amount of data. Processing of this type of data requires computing power that is probably impossible for individual computers to provide. So, researchers preferably opt for parallel/distributed computing techniques (MAITREY; JHA, 2015).

And there isn't any complete solutions on the road. The rate of information growth is 10 times every two years (IBM,)(SINTEF, 2013) and according to Moore's law¹⁵ the processing power and storage just double every 18 months.

One of the today's tools to deal with BigData is NoSQL, more specifically when dealing with some classes of problems, one the tools are NoSQL Key-Value store databases. Since we do believe in this research we are creating the grounds for a new and more effective Key-Value store solution, it is interesting to know few common use cases for such tool¹⁶:

- “Bigness: NoSQL is seen as a key part of a new data stack supporting: big data, big numbers of users, big numbers of computers, big supply chains, big science, and so on. When something becomes so massive that it must become massively distributed, NoSQL is there, though not all NoSQL systems are targeting big. Bigness can be

⁹ <http://wikibon.org/wiki/v/Big_Data_Market_Size_and_Vendor_Revenues>

¹⁰ <http://wikibon.org/wiki/v/Big_Data_Vendor_Revenue_and_Market_Forecast_2012-2017>

¹¹ <<http://www.idc.com/promo/thirdplatform/fourpillars/bigdataanalytics>>

¹² <<http://www.eweek.com/database/big-data-market-to-reach-46.34-billion-by-2018.html>>

¹³ <<https://www.abiresearch.com/press/big-data-spending-to-reach-114-billion-in-2018-loo>>

¹⁴ <<http://siliconangle.com/blog/2012/02/15/big-data-market-15-billion-by-2017-hp-vertica-comes-out-1-according-to-wikibon-research/>>

¹⁵ <<https://www.scientificamerican.com/article/moore-s-law-keeps-going-defying-expectations/>>

¹⁶ <<http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html>>

across many different dimensions, not just using a lot of disk space."

- "Massive write performance: This is probably the canonical usage based on Google's influence. High volume. Facebook needs to store 135 billion messages a month. Twitter, for example, has the problem of storing 7 TB/data per day with the prospect of this requirement doubling multiple times per year. This is the "data is too big to fit on one node" problem. At 80 MB/s it takes a day to store 7TB so writes need to be distributed over a cluster, which implies key-value access, MapReduce, replication, fault tolerance, consistency issues, and all the rest. For faster writes in-memory systems can be used."
- "Avoid hitting the wall: Many projects hit some type of wall in their course. They've exhausted all options to make their system scale or perform properly and are wondering, "what next"? It's comforting to select a product and an approach that can jump over the wall by linearly scale, using incrementally added resources. At one time this wasn't possible. It took custom rebuilding of everything, but that changed. We are now seeing usable out-of-the-box products that a project can readily adopt."
- "Distributed systems support: Not everyone is worried about scale or performance over and above that which can be achieved by non-NoSQL systems. What they need is a distributed system that can span data centers while handling failure scenarios without a hiccup. NoSQL systems, because they have focused on scale, tend to exploit partitions, tend not use heavy strict consistency protocols, and so are well positioned to operate in distributed scenarios."
- "Managing large streams of non-transactional data: Apache logs, application logs, MySQL logs, clickstreams, etc."
- "Fast response times under all loads"
- "Soft real-time systems where low latency is critical. Games are one example."
- "Sequential data reading"
- "User registration, profile, and session data"
- "Priority queues"
- "Simple time-series with roll-ups"

Big Data is an open problem, and it seems the problem will always be there. It is present in so many fields, and it may have huge financial and people's life impact. With this research we expect we can improve the current techniques on how to deal with certain class of Big Data problems.

3 Theoretical Foundations

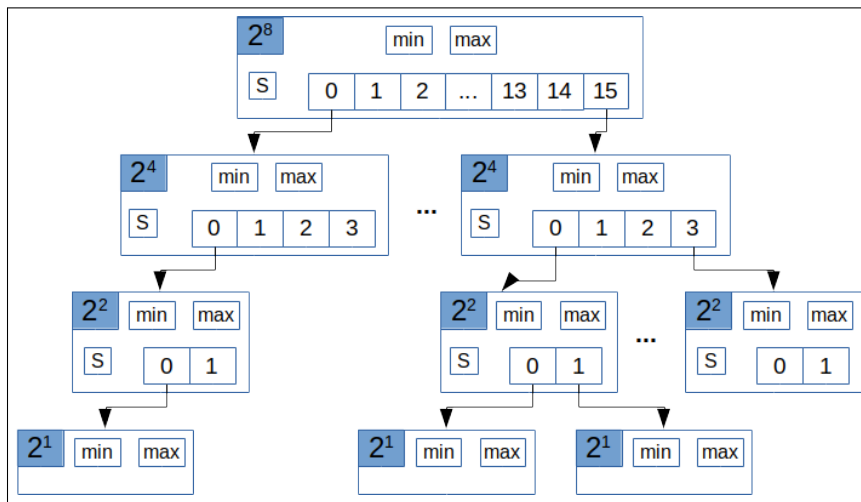
3.1 van Emde Boas trees

In this section we will explain the van Emde Boas tree using a very direct approach, If you rather like a more smooth and didactic explanation, please refer to Cormen's book (CORMEN et al., 2009) (chapter 20).

The van Emde Boas tree (vEBt) is a data structure that performs dynamic set operations, *insert*, *search*, *delete*, *successor*, *predecessor* in $O(\lg \lg U)$ and *min* and *max* in $O(1)$ worst case time cost.

The van Emde Boas (vEB) tree is a recursive structure that all children nodes are also vEB trees. Every single $vEB(u)^1$ node, except the leaves, have $\sqrt{u} \times vEBt(\sqrt{u})$ children, one summary and the minimum and maximum keys, as shown in Figure 2. The minimum and maximum elements are non-negative integer keys and the summary is also a $vEBt(\sqrt{u})$ as we will see in more detail in this section.

Figure 2 – van Emde Boas tree



A $vEB(2^8)$ has $2^4 \times vEB(2^4)$ trees. Each $vEB(2^4)$ has $2^2 \times vEB(2^2)$ trees. And so on until $vEB(2^1)$ leaves

To understand how the van Emde Boas tree works let's build it from a very basic data structure, a direct-address table (CORMEN et al., 2009) (chapter 11.1).

¹ For $vEB(u)$ or $vEBt(u)$ we denote a van Emde Boas tree of universe size u

² u , the universe size of node is the maximum number of keys that can be stored by the node and its children. In vEB trees, keys range from 0 to $u-1$

A direct-address table is a ordered bit vector indexed from ‘0’ to ‘ $U - 1$ ’, in which indexes represent keys of the universe ‘ U ’. If the key is present in the set it has value ‘1’, otherwise it has value ‘0’. As you can see from Table 2, the direct-address table has $O(1)$ time only for dictionary operations and $O(U)$ for the remaining dynamic set operations.

Table 2 – Direct-address vs vEB time cost.

operation	DA table	ordered vector	vEB
insert	$O(1)$	$O(n)$	$O(\lg \lg U)$
remove	$O(1)$	$O(n)$	$O(\lg \lg U)$
search	$O(1)$	$O(\lg n)$	$O(\lg \lg U)$
successor	$O(U)$	$O(\lg n)$	$O(\lg \lg U)$
predecessor	$O(U)$	$O(\lg n)$	$O(\lg \lg U)$
min	$O(U)$	$O(1)$	$O(1)$
max	$O(U)$	$O(1)$	$O(1)$

Comparison of worst case time cost of dynamic set operations between direct-address tables, ordered vector and van Emde Boas tree. The ordered vector has $O(1)$ time for ‘successor’ and ‘predecessor’ operations if the index of the element is already known, and the cost of ‘insert’ and ‘remove’ operations are actually ‘ $\lg n$ ’ to find the index plus ‘ $vector.size() - index - 1$ ’ to move the memory on the tail.

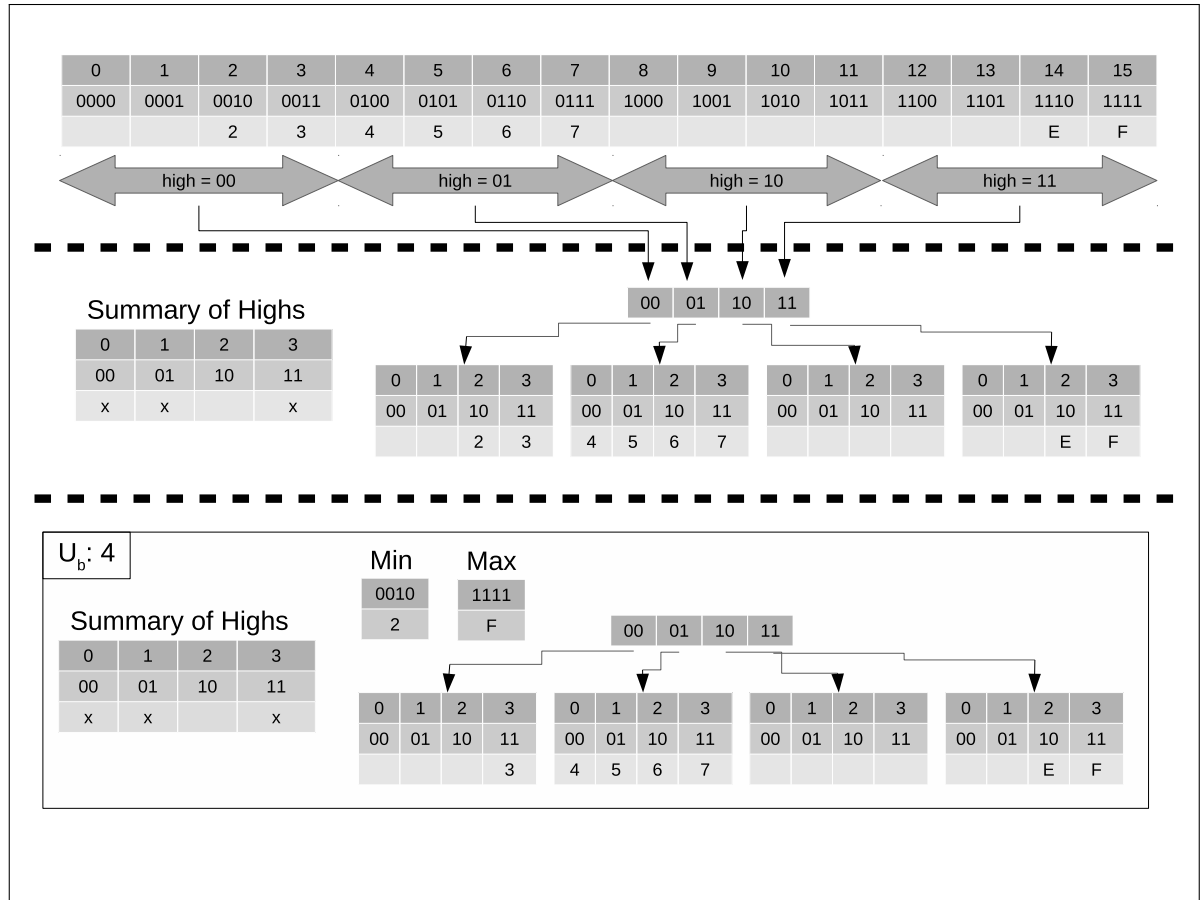
You might be asking yourself, why not start from a dynamic ordered vector instead of a direct-address table? The reason is because there is a lower bound $\Omega(n \lg n)$ for comparison sorting operations (CORMEN et al., 2009) (Pgs 191-193), and with direct-address tables we operate on universe instead of elements present on the set, thus we may have a chance to trade ‘ $\lg n$ ’ by ‘ $\lg \lg U$ ’ time cost. See Radix sort (CORMEN et al., 2009) (Pg 197) as an example of an algorithm exploiting the universe to cheat around the $\Omega(n \lg n)$ sorting limitation.

On the top of Figure 3 there is a direct-address table. The first row of the table has indexes of the table in decimal basis, the second row has indexes in binary basis, and the third row has keys stored as hexadecimal values. The second row wouldn’t be need to represent the direct-address table, the third row could also only have ‘0’s or ‘1’s to represent if keys are present or not, but this redundancy will help us to visualize how a vEB tree is built from it. The arrows represents indexes, in binary basis, that have the same half most significant bits.

The first step to build our vEBt is group together vector keys that have the same half most significant bits, as shown in the mid section of the Figure 3.

Notice that, by dividing the array in groups of most half significant bits we are actually diving the array of size ‘ u ’ in ‘ \sqrt{u} ’ arrays of size ‘ \sqrt{u} ’. That’s because our universe will always be a power of 2, *i.e.* $u = 2^m$, and $\sqrt{2^m} = 2^{m/2}$. The keys into sub-arrays now only hold the half significant bit of their original values. To retrieve back any key from this new structure, we just need to concatenate the index of the sub-array with the new

Figure 3 – van Emde Boas tree cluster array



An array of size 16 broken into one array of size $\sqrt{16}$, with each element pointing to arrays of size $\sqrt{16}$. The summary, of size $\sqrt{16}$, indicates what sub-arrays have at least one element. The last section of the image shows two registers holding the *minimum* and *maximum* keys. Notice the *minimum* key has been removed from its original sub-array. ‘ $U_b : 4$ ’ means the universe is ‘ $U : 2^4$ ’

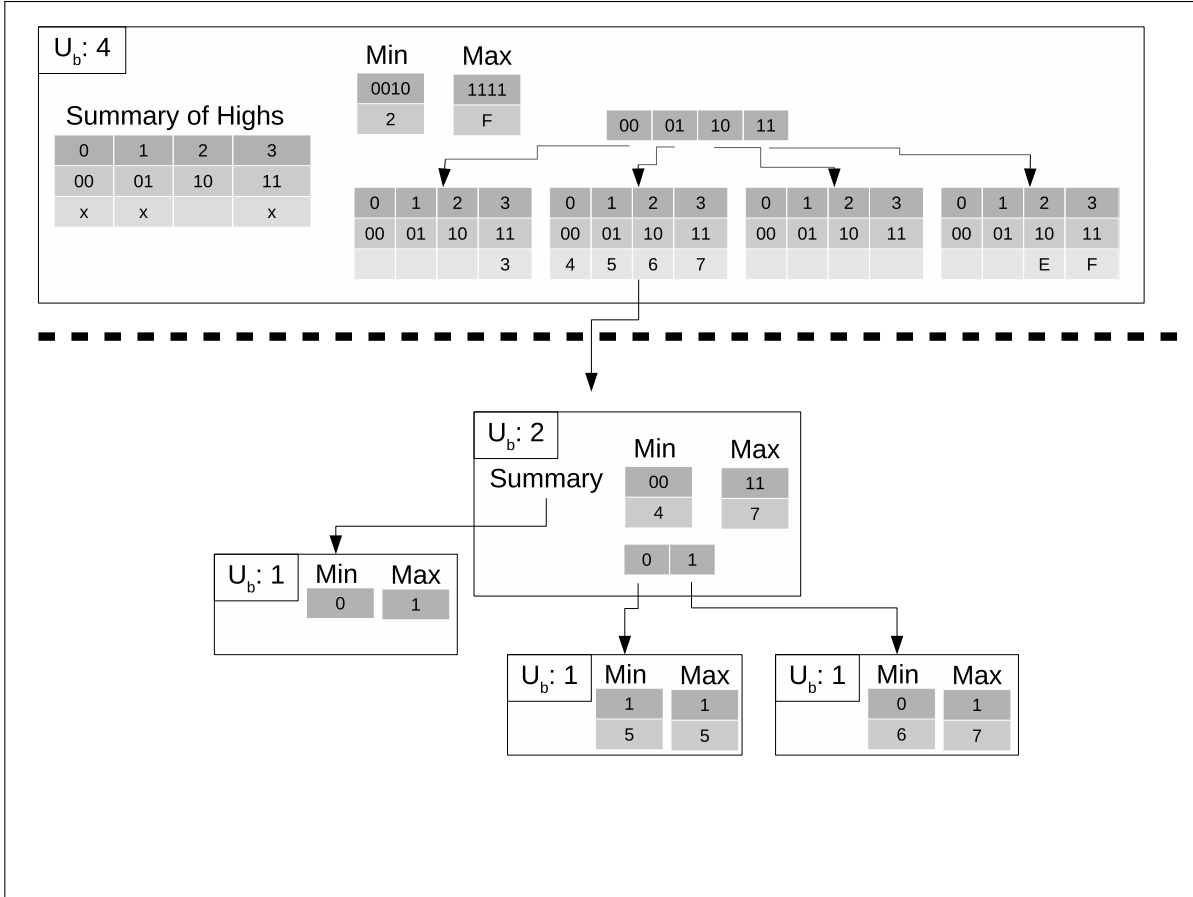
value of the key.

The second step is to create a summary that has ‘x’ marks to indicate what sub-arrays have at least one key present. As an example, the value ‘x’ in the summary at index ‘1’, states there is at least one key in the second sub-array, and the absence of ‘x’ in index ‘2’ indicates there isn’t any key in the third sub-array.

The third step is to store the *minimum* and *maximum* keys in separated registers and remove the minimum element from its original sub-array. After removing, the corresponding summary mark must be cleared if that sub-array just became empty. See the last section of Figure 3.

Finally to build the complete vEB tree as depicted in Figure 5 we just need to apply these three steps recursively for all sub-arrays and summaries until they reach leaves of universe size 2^1 , as shown in Figure 4.

Figure 4 – Breaking an array into a vEB node recursively



The second sub-array has been recursively broken until vEB(2^1) trees. vEB(2^1) trees has only the *minimum* and *maximum* registers.

Now that we understand the vEBt structure, let's get some intuition how $O(\lg \lg U)$ is achieved. To do that, we will exercise the vEB tree we've just built in Figure 5. Calling *search*(6) (Algorithm 1) to retrieve key '6', binary '0110', we have to go down '2' levels on the tree. That is not a coincidence, '2' is ' $\lg \lg 16$ ' and '16' the universe of the tree.

We are ready now to face a the formal approach.

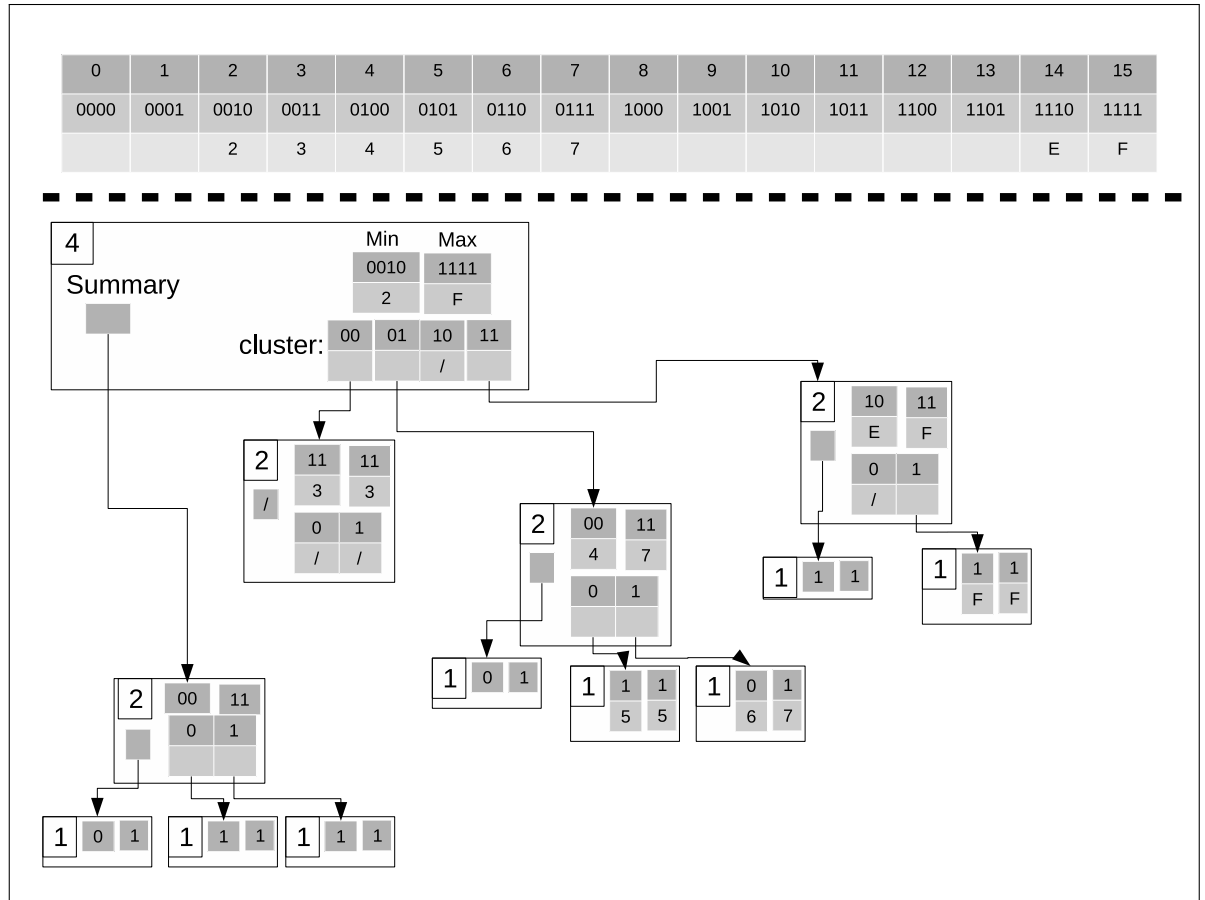
The search Algorithm 1 has a recurrence of the form $T(U) \leq T(\sqrt{U}) + O(1)$. The first term of the recurrence comes from line 7.

To solve this recurrence, let first recall from master theorem method (CORMEN et al., 2009) (Pgs 94, 95) for solving recurrences of the form $T(n) = a \times T(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. Our recurrence matches the second case where $f(n) = \Theta(\log_b a)$, and then $T(n) = \Theta(n^{\log_b a} \times \lg n)$

Back to the recurrence:

Making $m = \lg U$

Figure 5 – The complete vEB tree



The initial array has been completely recursively broken into a vEB tree. The square box with a 4 on the top most vEB structure, actually represents a 2^4 Universe

$$T(2^m) \leq T(2^{\lceil m/2 \rceil}) + O(1)$$

Noting that $\lceil m/2 \rceil \leq 2m/3$ for all $m \geq 2$, we have

$$T(2^m) \leq T(2^{2m/3}) + O(1)$$

Letting $S(m) = T(2^m)$, we rewrite the recurrence as

$$S(m) \leq S(2m/3) + O(1)$$

From the master theorem

$$S(m) = O(m^{\log_{3/2} 1} \times \lg m)$$

$$S(m) = O(\lg m)$$

remembering $m = \lg U$, $T(U) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg U)$

finally $T(U) = O(\lg \lg U)$

It is easy to see the recurrence on *search()* operation. Other operations like *successor()* and *predecessor()* are a bit more tricky, the idea is that for each level we can go down following a child on the cluster array or the summary, but both, so we will go down at most $\lg \lg U$, and as we will see the *min* and *max* keys at each level help in this trick.

Algorithm 1 vEB search (CORMEN et al., 2009)

```

1: procedure vEB::SEARCH(key)
2:   if key = min OR key = max then
3:     return TRUE
4:   else if  $U_b = 1$  then
5:     return FALSE
6:   else
7:     return cluster[high(key)].search(low(key))
8:   end if
9: end procedure

```

The successor Algorithm 2, recursively calls *successor()* on a cluster sub-tree at line 13, or *successor()* on summary at line 16, but both. At line 11 and 20 there are *min()* and *max()*, which are $O(1)$ operations.

The idea of successor Algorithm 2 is check the base case when Universe is 2^1 at lines 2 to 7. If that's the case, it will return *max* if it is '1' and *key* '0', otherwise there isn't a successor at this level and returns 'NIL'. If Universe is greater than 2^1 , and the *key* is less than a valid *min*, it just returns *min* because it will actually be the successor of *key* (line 8). Otherwise, it checks if the successor lies into the same cluster as *key* (line 11 to 13). If not, it returns the *minimal* of successor cluster by searching it in the summary (lines 16 to 22).

The predecessor Algorithm 3 has exactly the same idea as the successor Algorithm 2. It may call *predecessor()* in the cluster sub-tree at line 13, or in the summary at line 16, but both. The algorithm is symmetric to *successor()*, except by a minor detail, *min* values are not present in sub-trees while *max* are.

At the insert Algorithm 4, lines 5-6 are the base case, an empty tree, lines 8-9 set *key* to *min* and insert the previous *min* that is not the minimum value anymore into the tree. Lines 12-14, insert *high(key)* into the summary only if it was empty, making a recursive call at line 13. If that's is not the case, *i.e.* there was something already in *cluster[high(key)]*, *low(key)* is inserted into the cluster making it recurse at line 16. Finally, at line 20, *key* replaces *max* if it is greater. As you can see, it can recurse either at line 13 or 16 but both.

At delete Algorithm 5, lines 2-5 delete *min* and *max* if they are equals to *key*. Lines 6-12 handles vEB with universe of 2, if line 7 is reached, means that *min* is different from *max* and *key* is either 0 or 1, then one of them will be deleted and *max* will be equal to *min* with just one key into that node. At lines 14-18, if *key* is equal *min*, *min* is replaced with the next minimal value and that value will be deleted from the corresponding child tree. Remember that the *min* is never present into children trees. Line 18 deletes the *key* from the sub-tree it belongs to. If it just became empty, lines 21-29, the summary

Algorithm 2 vEB successor (CORMEN et al., 2009)

```

1: procedure vEB::SUCCESSOR(key)
2:   if  $U_b = 1$  then
3:     if  $key = 0$  AND  $max = 1$  then
4:       return 1
5:     else
6:       return NIL
7:     end if
8:   else if  $min \neq NIL$  AND  $key < min$  then
9:     return min
10:  else
11:     $max\_low \leftarrow cluster[high(key)].max()$ 
12:    if  $max\_low \neq NIL$  AND  $low(key) < max\_low$  then
13:       $offset \leftarrow cluster[high(key)].successor(low(key))$ 
14:      return index(high(key), offset)
15:    else
16:       $succ\_cluster \leftarrow summary.successor(high(key))$ 
17:      if  $succ\_cluster = NIL$  then
18:        return NIL
19:      else
20:         $offset \leftarrow cluster[succ\_cluster].min()$ 
21:        return index(succ_cluster, offset)
22:      end if
23:    end if
24:  end if
25: end procedure

```

is updated by removing the corresponding cluster index. If the deleted *key* was equal to *max*, *max* is set to *min* if it is the only element left on the tree, *max* is set to the new maximum value left in the tree, line 27. Lines 30-31 do the same as lines 22 and 27, but without needing to delete the cluster index from summary. At first glance, looks like we can recurse in line 19 and line 21, but line 21 only executes if line 19 has only one element and takes $O(1)$ time.

The minimum Algorithm 6 and maximum Algorithm 7 Algorithms are very straight, executes in $O(1)$ time, and don't need further explanation.

As we have seem, the van Emde Boas cheats around the $\Omega(n \lg n)$ lower bound by operating on the Universe, but this come at a price. Its cluster uses a lot of memory. Even an empty tree, like shown in Table 3, may use a huge amount of memory. The space requirement of the van Emde Boas tree is characterized by the recurrence $P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u})$. Summary and children are represented by the first term and clusters are the second. *Min* and *max* keys requires $\Theta(\lg u)$ space each and are combined into the last term.

Let's solve this recurrence by unrolling it. First notice, the function $f(u) = \sqrt{u}$,

Algorithm 3 vEB predecessor (CORMEN et al., 2009)

```

1: procedure vEB::PREDECESSOR(key)
2:   if  $U_b = 1$  then
3:     if  $key = 1$  AND  $min = 0$  then
4:       return 0
5:     else
6:       return NIL
7:     end if
8:   else if  $max \neq NIL$  AND  $key > max$  then
9:     return max
10:  else
11:     $min\_low \leftarrow cluster[high(key)].min()$ 
12:    if  $min\_low \neq NIL$  AND  $low(key) > min\_low$  then
13:       $offset \leftarrow cluster[high(key)].predecessor(low(key))$ 
14:      return index(high(key), offset)
15:    else
16:       $pred\_cluster \leftarrow summary.predecessor(high(key))$ 
17:      if  $pred\_cluster = NIL$  then
18:        if  $min \neq NIL$  AND  $key > min$  then
19:          return min
20:        else
21:          return NIL
22:        end if
23:      else
24:         $offset \leftarrow cluster[pred\_cluster].max()$ 
25:        return index(pred_cluster, offset)
26:      end if
27:    end if
28:  end if
29: end procedure

```

if iterated, needs $\lg \lg u$ steps to reduce its argument down to 2 or less, *i.e.* $f_2^*(u) = \lg \lg u$ (CORMEN et al., 2009) (Pg 63). It is easier to see by making $u = 2^{2^m}$, it needs $\lg 2^m = m$ steps to reduce it to $2^{2^0} = 2$, *i.e.* it needs $\lg \lg 2^{2^m} = m = \lg \lg u$ steps. If you look at Table 8 it will become very intuitive. So, unrolling the recursion we get:

$$P(u) = \left(\prod_{i=1}^{\lg \lg u} (u^{1/2^i} + 1) \right) P(2) + \sum_{i=2}^{\lg \lg u} \left[\left(\prod_{j=2}^i (u^{1/2^{j-1}} + 1) \right) \Theta(u^{1/2^i}) \right] + \Theta(u^{1/2})$$

Let's simplify the term, $T = \prod_{j=2}^i (u^{1/2^{j-1}} + 1)$

Using only the dominant term:

$$T = O\left(\prod_{j=2}^i u^{1/2^{j-1}}\right) = O\left(\prod_{j=1}^{i-1} u^{1/2^j}\right) = O\left(\frac{\prod_{j=0}^{i-1} u^{1/2^j}}{u}\right)$$

Applying sum of n terms of geometric series:

Algorithm 4 vEB insert (CORMEN et al., 2009)

```

1: procedure vEB::INSERT_EMPTY(key)
2:   min  $\leftarrow$  max  $\leftarrow$  key
3: end procedure
4: procedure vEB::INSERT(key)
5:   if min = NIL then
6:     insert_empty(key)
7:   else
8:     if key < min then
9:       swap(key, min)
10:    end if
11:    if  $U_b > 1$  then
12:      if cluster[high(key)].min() = NIL then
13:        summary.insert(high(key))
14:        cluster[high(key)].insert_empty(low(key))
15:      else
16:        cluster[high(key)].insert(low(key))
17:      end if
18:    end if
19:    if key > max then
20:      max  $\leftarrow$  key
21:    end if
22:  end if
23: end procedure

```

$$O\left(\frac{u\left(\frac{1-1/2^i}{1-1/2}\right)}{u}\right) = O\left(u^{1-(1/2^{i-1})}\right) = O\left(u^{1-(2/2^i)}\right) = O\left(\frac{u}{(u^{1/2^i})^2}\right)$$

The new formula becomes:

$$P(u) = \left(\prod_{i=1}^{\lg \lg u} (u^{1/2^i} + 1)\right) P(2) + \sum_{i=2}^{\lg \lg u} \left[O\left(\frac{u}{(u^{1/2^i})^2}\right) \Theta(u^{1/2^i}) \right] + \Theta(u^{1/2})$$

$$P(u) = \left(\prod_{i=1}^{\lg \lg u} (u^{1/2^i} + 1)\right) P(2) + \sum_{i=2}^{\lg \lg u} \left[O\left(\frac{u}{(u^{1/2^i})}\right) \right] + \Theta(u^{1/2})$$

The highest term of the product will be $\prod_{i=i}^{\lg \lg u} 1/2^i$, which just a few terms of the sum of the geometric series given by $\prod_{i=0}^{\infty} 1/2^i$. The sum of the geometric series converge to 2 by applying the formula $1/(1-r)$. Without the term $i=0$, we have $\prod_{i=i}^{\lg \lg u} 1/2^i < 1$ and thus this production is $o(u)$.

For the summation, the highest term will be for ' $i = \lg \lg u$ '. Making $u = 2^{2^m}$, we have $(2^{2^m})^{1/2^{\lg \lg 2^{2^m}}} = (2^{2^m})^{1/2^m} = 2$ that gives $O(u)$. Since the largest term is $O(u)$ the solution for the recursion is $P(u) = O(u)$.

The last column in Table 3 shows the memory used for a full vEB tree for few

Algorithm 5 vEB remove (CORMEN et al., 2009)

```

1: procedure vEB::REMOVE(key)
2:   if min = max then
3:     if min = key then
4:       min  $\leftarrow$  max  $\leftarrow$  NIL
5:     end if
6:   else if  $U_b = 1$  then
7:     if key = 0 then
8:       min  $\leftarrow$  1
9:     else
10:      min  $\leftarrow$  0
11:    end if
12:    max  $\leftarrow$  min
13:  else
14:    if key = min then
15:      first_cluster  $\leftarrow$  summary.min()
16:      key  $\leftarrow$  index(first_cluster, cluster[first_cluster].min())
17:      min  $\leftarrow$  key
18:    end if
19:    cluster[high(key)].delete(low(key))
20:    if cluster[high(key)].min() = NIL then
21:      summary.remove(high(key))
22:      if key = max then
23:        summary_max  $\leftarrow$  summary.max()
24:        if summary_max = NIL then
25:          max  $\leftarrow$  min
26:        else
27:          max  $\leftarrow$  index(summary_max, cluster[summary_max].max)
28:        end if
29:      end if
30:    else if key = max then
31:      max  $\leftarrow$  index(high(key), cluster[high(key)].max)
32:    end if
33:  end if
34: end procedure

```

Algorithm 6 vEB minimum (CORMEN et al., 2009)

```

1: procedure vEB::MINIMUM(key)
2:   return min
3: end procedure

```

Algorithm 7 vEB maximum (CORMEN et al., 2009)

```

1: procedure vEB::MAXIMUM(key)
2:   return max
3: end procedure

```

universe sizes.

Table 3 – Memory cost of a vEB tree.

k	2^k (bits)	$U = 2^{2^k}$	cluster (bytes)	pointers	memory (bytes)
0	1	2	0	0	2
1	2	4	16	3	32
2	4	16	32	20	202
3	8	256	128	357	3,572
4	16	65,536	2,048	92,006	920,064
5	32	4.29E+09	524,288	6.03E+09	6.03E+10 (56 GB)
6	64	1.84E+19	3.44E+10 (32 GB)	2.59E+19	2.59E+20 (58 EB)
7	128	3.40E+38	2.95E+20	4.78E+38	4.78E+39
8	256	1.16E+77	1.09E+40	1.63E+77	1.63E+78

From $2^k = 1$ to $2^k = 64$, a 64-bits machine is considered. For $2^k = 128$ and $2^k = 256$, a 128-bits and 256-bits machine are considered respectively. It is also considering the tree has no satellite data. The fourth column is the size in bytes of clusters. The fifth column is total number of pointers in the tree structure. The last column is the total space in bytes occupied by the whole tree ³.

As we can see, the size of full vEB tree, as well the size of a empty vEB tree are a big issue. We are addressing these issues in this research.

3.2 Computer Networks Review

In this section we will very briefly recap basic network concepts to help understand some decisions made in our methodology. Let's start from the reference models then jump directly into UDP protocol that was intensively used in this research.

3.2.1 Reference Models

The first computer networks where designed with hardware as main concern and software as an afterthought (TANENBAUM; WETHERALL, 2012).

With complexity growing, networks were designed as a stack of layers. The purpose of each layer was to hide the complexity and details of implementation from higher layers. A layer also offers services to higher layer and the agreement or interface on how to use that layer is called "protocol".

In 1983, the International Standards Organization (ISO), defined a reference model to standardize network layers and its protocols. It is know as Open System Interconnection (OSI). The OSI reference model has the following layers (Figure 6):

³ <https://docs.google.com/spreadsheets/d/1D_zElpJwRuKsx_IxwYNo9Iy8IZ2Mt_GPHE51NXg9SCo/edit?usp=sharing>

1. Physical - Concerns with how to transmit bits over the communication channel. how many bits has byte, whats the voltage level for bit 1 and bit 0, how long a bit lasts, ...;
2. Data Link - Send frames of data. If it is reliable, the receiver send an Ack back to the sender. Deals with framing (*e.g.* byte stuffing). Offer the higher level layer a regulation mechanism to indicate when it is ready to send more data, this is useful when the receiver is slower;
3. Network - Provides routing mechanism for a packet sent from source reach the destination. Also offer some QoS to handle congestion control. Allow heterogeneous networks to interconnect;
4. Transport - Accept data from above layers and splits it in smaller pieces to be sent by Network layer. This layers offer two different services to higher layers, one is an error-free point-to-point channel, that deliver messages in exactly order they are sent, and another is a connection-less transfer where packets may not be delivery or may be delivered out of order. From this layer and above the communication is really end-to-end, while on the layer bellow that the communication happens with their neighbors;
5. Session - Establish a session between peers. Can be used to keep track of whose turn to transmit, or access to critical operations for example;
6. Presentation - Concerns with syntax or semantics of information transmitted, *e.g.* High level layers will see the same data even if they run on machines with different endianness;
7. Application - Protocol commonly used by applications, *i.e.* HTTP, FTP, ...

On late 1960's, the U.S. Department of Defense (DoD), sponsored a named ARPANET, that later was named TCP/IP. The protocol was designed to allow a packet leave source and reach destination even if some machines or transmission lines were suddenly put out of operation, to allow heterogeneous network interconnection and to be very flexible by allowing different transfers, ranging from transferring files to real-time applications.

The TCP/IP reference model has the following layers (Figures 6 and 7):

1. Link - Describe how links like serial lines or classic Ethernet must do to meet the requirements of higher level layers;

Figure 6 – OSI vs TCP/IP reference model

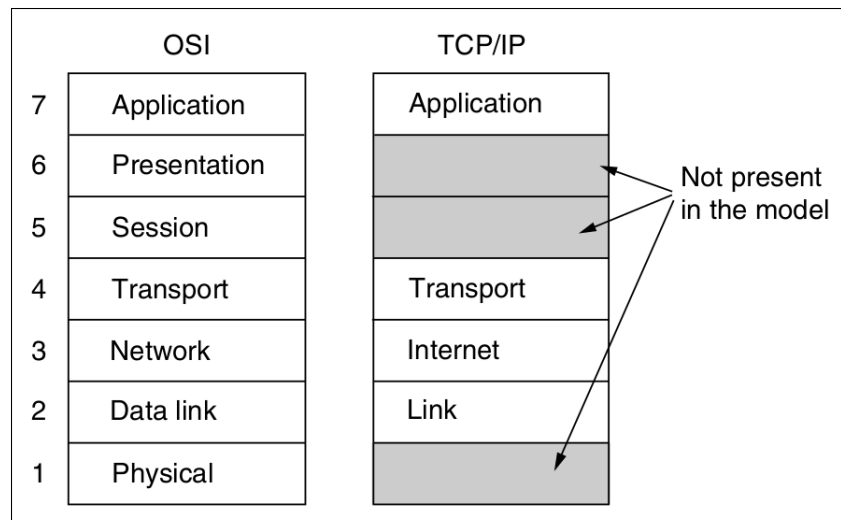


Figure extracted from Computer Networks Tanenbaum's book ([TANENBAUM; WETHER-ALL, 2012](#)) (pg 46)

2. Internet - Allows routing and heterogeneous network interconnection. Packets can be delivered in any order, so higher level layers has to order it if want to. It offers Internet Protocol (IP) and Internet Control Message Protocol (ICMP) protocols;
3. Transport - Like in OSI model, allows end-to-end communication. Offers two services to higher layers, the first one, Transmission Control Protocol (TCP), is reliable connection oriented and also handles flow control. The second, User Datagram Protocol (UDP), is unreliable, connection less and doesn't provide flow control;
4. Application - Higher level protocols.

3.2.2 UDP

The UDP allows to send encapsulated IP datagrams without establishing a connection. It transmits segments consisting of 8-bytes header (Figure 8) followed by the payload. The two ports identifies the source and destinations endpoint, by using these ports the operating system can delivery the message to the right (binded) application.

The minimum UDP packet is 8-bytes length, because of the UDP header, see Figure 8. The maximum is 65,515 bytes because of the maximum length of a IPv4 packet (IP *length* field minus IPv4 header size, *i.e.* '65,535 - 20').

UDP is not reliable and doesn't guarantee packets will be delivered in right order. In addition it doesn't implements flow control as TCP does. But its simplicity some times

Figure 7 – TCP/IP layers and typical protocols

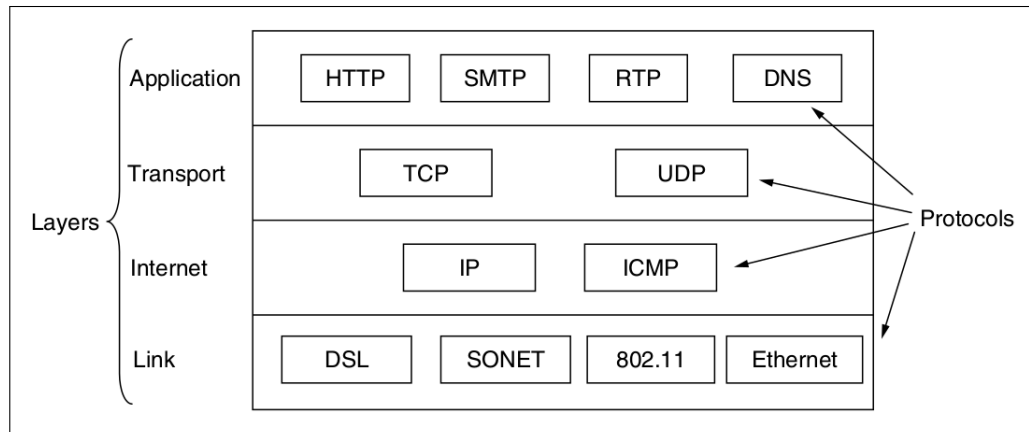


Figure extracted from Computer Networks Tanenbaum's book ([TANENBAUM; WETHERALL, 2012](#)) (pg 48)

Figure 8 – UDP header

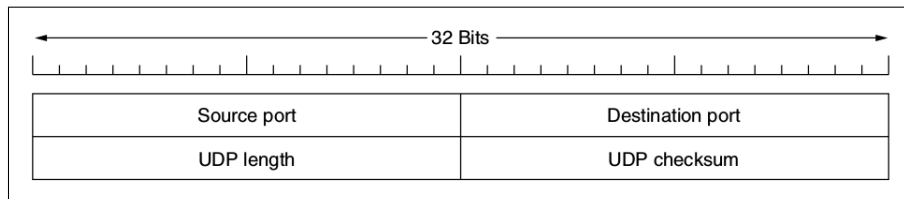


Figure extracted from Computer Networks Tanenbaum's book ([TANENBAUM; WETHERALL, 2012](#)) (pg 542)

makes it better for some applications⁴. It has a smaller header and smaller delay due to TCP initial 3-way handshake.

IPv4 UDP frames and regular IPv6 UDP frames may have up to 65,507 bytes (65,535 - 8 byte UDP header - 20 byte IP header) ([EGGERT; FAIRHURST, 2008](#))([HEFFNER; CHANDLER, 2007](#)) and jumbo-frames can carry up 9000 bytes of payload. Ideally, it is not recommended to send UDP packets bigger than MTU ([EGGERT; FAIRHURST, 2008](#))([HEFFNER; CHANDLER, 2007](#)) because the UDP fragmentation causes less reliability.

UDP messages can be sent as Unicast, when it is directed to a specific destination address, as Broadcast when it is sent to all machines of a sub-net, or Multicast when it is addressed to group of subscriber machines. In this research we are using Unicast and Multicast UDP messages.

⁴ Applications where low latency is a must, *e.g.* games, and/or reliability is not mandatory, *e.g.* multimedia streaming.

3.2.3 Unicast

Unicast addressing uses a one-to-one association between a sender and destination. Each destination address uniquely identifies a single receiver endpoint.

To send a Unicast message on a Local Network, the sender needs to know the MAC address of the destination machine. Each computer in a network keep a table that maps IPs in MAC addresses. That table is named Address Resolution Protocol (ARP) table in IPv4 and Network Discovery Protocol (NDP) table in IPv6. When some MAC is still unknown, the ARP or NDP protocol is used to discover and save it for future use. Entries on the table are valid for a certain period of time and the number os entries is parameterized by the Operating System. A entry a machine may expire or be dropped if table is full, usually the linux ARP table size is 1024 entries.

3.2.4 Multicast

Sometimes we want to send exactly the same message to several machines. Even if there are few machines, send those same messages individually to any single machine may have an undesirable cost. To avoid that, we could send a Broadcast message, so that we just send the message once but all machines receives the message. But Broadcast also have its drawbacks, few machines may not be interested on such messages, or even worse, they are interested but aren't supposed to receive such messages.

To solve that, there is another one-to-many message named Multicast, With multicast, the sender sends a message to a group and only machines signed with that group receives the message.

The IPv4 224.0.0.0/24 range is reserved for multicast on local network. If a multicast group has members on others networks, a routing protocol is need, but that's not the case for this research.

3.3 Distributed Systems

To understand the fundamental building blocks of a distributed system, it is necessary to consider four key questions:

- *Communicating entities* - What are the entities that are communicating in the distributed system? It is helpful to address the first question from a system-oriented and a problem-oriented perspective. From the system perspective, it can be Threads, Processes, or Nodes (machines). From the programming perspective, there is the following problem-oriented abstraction: Objects, Components and Web services.

- *Communication paradigms* - How do they communicate, or, more specifically, what communication paradigm is used?
 - *Interprocess Communication* paradigm refers to the relatively low-level support for communication between processes in distributed systems.
 - *Remote Invocation* represents the most common communication paradigm in distributed systems, covering a range of techniques based on a two-way exchange between communicating entities in a distributed system.
 - * *Request-reply protocols* are effectively a pattern imposed on an underlying message-passing service to support client-server computing. In particular, such protocols typically involve a pairwise exchange of messages from client to server and then from server back to client, with the first message containing an encoding of the operation to be executed at the server and also an array of bytes holding associated arguments and the second message containing any results of the operation, again encoded as an array of bytes. This paradigm is rather primitive and only really used in embedded systems where performance is paramount.
 - * In *Remote Procedure Call (RPC)* procedures in processes on remote computers can be called as if they are procedures in the local address space. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call.
 - * *Remote method invocation (RMI)* strongly resembles remote procedure calls but in a world of distributed objects. With this approach, a calling object can invoke a method in a remote object. As with RPC, the underlying details are generally hidden from the user.
 - In *Indirect Communication* - It can be:
 - * *Group communication* - Is concerned with the delivery of messages to a set of recipients and hence is a multi-party communication paradigm supporting one-to-many communication;
 - * *Publish-subscribe systems* - One-to-many systems wherein a large number of producers (or publishers) distribute information items of interest (events) to a similarly large number of consumers (or subscribers);
 - * *Message queues* - Point-to-point indirection between the producer and consumer processes;
 - * *Tuple spaces* - Many-to-many indirect communication whereby processes can place arbitrary items of structured data, called tuples, in a persistent

tuple space and other processes can either read or remove such tuples from the tuple space by specifying patterns of interest;

* *Distributed shared memory* - Provide an abstraction for sharing data between processes that do not share physical memory.

- *Roles and responsibilities* - What (potentially changing) roles and responsibilities do they have in the overall architecture?
- *Placement* - How are they mapped on to the physical distributed infrastructure (what is their placement)?

We will see in section 4.3.2.1 that we have implemented our communication protocol from the ground up, thus the Request-reply protocols is definitely the dominant communication paradigm we worked with. Let's see some concepts usually present in Request-reply systems:

- *Message Identifiers* - This must be present if we need a reliable message delivery or request-reply. Usually it is composed of two parts, the first one is a sequential number unique in the process, and the second is a unique identifier of the sender in the whole distributed system.
- *Failure model* - When implemented using UDP, there may have omission failures, system failure or out of order reception. The action taken when timeout occurs depends on delivery guarantees being offered (*e.g.* retry, save message to disk for later retry)
- *Timeouts* - When, a timeout occur, the RPC layer could just return an error to the caller, or retry a few times. The timeout between retries can be fixed or adjustable based on QoS or congestion/flow control strategies.
- *Discarding duplicates* - In case of retransmission, the same message may be received more than once. The message identifier are used to detect duplicated messages.
- *Lost reply* - If the reply/answer message wasn't received by the client, may make the client to send the message again. If the messages are idempotent, *i.e.* can be executed repeatedly with the same effect, than it is not that harmful other than use more CPU, otherwise, some special treatment must be taken.
- *History* - Can be used to deal with duplicated messages and lost replies.
- *Style of exchange protocol* - Three different types of protocols, that produces different behaviors in the presence of communication failure are:

- Request (R) - There is reply for the message sent. It is implement with UDP datagram and therefore suffer from the same communication failures;
- Request-Reply (RR) - In this case, the replay message can be used as an acknowledge, and the retransmission may be done if it is not received. A history can be used to deal lost replies;
- Request-Reply-Acknowledge (RRA) - It is like RR, but the final Ack sent by the client can be used to remove the entry from the History;

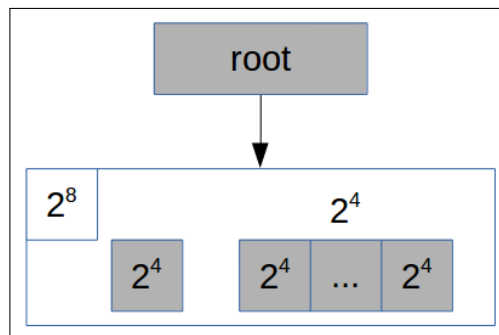
4 Methodology

In Subsection 2.1.1 we depicted challenges we had to overcome in order to achieve the main goal 1.2 of this research. In this chapter we will see the designing decisions taken to solve such challenges.

The first section of this chapter, describes the designing decisions taken to overcome the first challenge, while the remaining sections are devoted to explain the designing decisions take to overcome the second and third challenges. Those last two challenges are tight together and then treated simultaneously. Subsection 4.3.1 shows a preliminary solution and introduce basic concepts that are the foundations of our final approach. We decided to present this preliminary solution because it was the first approach implemented in this research. It is also a simpler approach, that runs on top of IPv6, and probably make easier to understand the basic principles we want to introduce. Then, at the end of Subsection 4.3.1 we present several drawbacks of that *IPv6* solution and move to Subsection 4.3.2 to present our final design.

4.1 Challenge 1: Allow a vEB tree to increase its universe

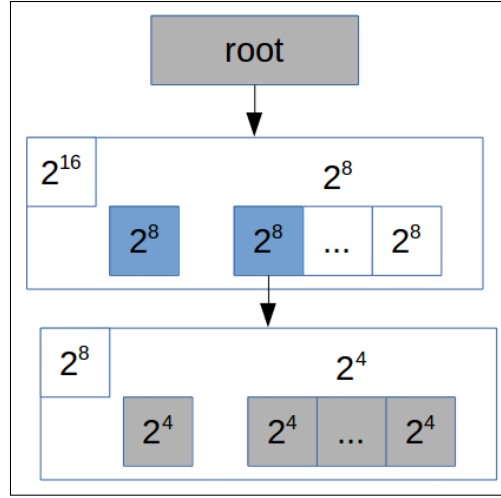
Figure 9 – A vEB tree with universe 2^8 using a proxy Root



The Root delegating to a vEB of universe of 2^8 .

Suppose we have a original vEB tree (Figure 9) with Universe of 256 (2^8) and we want to add a new key that is bigger than 255, for instance, 300, then we just need to:

1. Create a new vEB($2^{8 \times 2}$) tree (Figure 10);
2. Remove the min element from our original vEB(2^8) tree;
3. Set the removed element as min element of our new vEB($2^{8 \times 2}$) tree;

Figure 10 – The same vEB tree, now with universe 2^{16} 

The proxy Root now delegates its calls to a vEB_{2^8} . The previous $\text{vEB}_{2^{8 \times 2}}$ is now pointed by the first cluster's element of the top vEB.

4. Set the max element of our new $\text{vEB}(2^{8 \times 2})$ tree to be the same as the max element of our original $\text{vEB}(2^8)$ tree;
5. Point the first cluster of our new $\text{vEB}(2^{8 \times 2})$ to the original $\text{vEB}(2^8)$ tree;
6. Point the 'Root' to our new $\text{vEB}(2^{8 \times 2})$ tree.

Algorithm 8 vEbDynamic insert operation

```

1: procedure vEBDYNAMIC::INSERT(key, data)
2:   while key > universeMax(mK) do
3:     min = mVEB.getMin()                                ▷ saving before delete
4:     if min ≠ NIL then
5:       mVEB.remove(min)
6:     end if
7:     auxVEB ← new vEbK(SHIFTLEFT(mK, 1))
8:     auxVEB.max ← mVEB.max
9:     if mVEB.min ≠ NIL then
10:      auxVEB.summary.insert(0)
11:    end if
12:    auxVEB.cluster[0] ← mVEB
13:    mVEB ← auxVEB
14:    mK ← SHIFTLEFT(mK, 1)
15:    mVEB.min ← min
16:  end while
17:  mVEB.insert(key, data)
18: end procedure
  
```

Notice that, since a $vEB(2^{k \times 2})$ tree holds a pointer to a $vEB(2^k)$ tree, this procedure is very simple. Algorithm 8 depicts this procedure. The described algorithm allows a $vEB(2^{2^k})$ to grow to any $vEB(2^{2^{k+1}})$ with just $O(\lg \lg 2^{2^k})$ time cost, due to the line `remove()`, at line 5. The `insert()`, at line 10 is $O(1)$. The `insert()` at line 17 is just the regular insert after the expansion of the tree is done to accommodate the key about to be inserted and is not really part of the expansion itself. The described algorithm is actually a method of a *vEbDynamic* proxy class. Notice that *mVEB* is “real subject” attribute of *vEbDynamic* and points the *root* of the tree.

It is worth to mention that during our analysis we considered an alternative solution to increase the universe from 2^k to 2^{k+1} , but it would have a high undesired cost, because, opposed to the $vEB(2^{k \times 2})$ approach, a $vEB(2^{k+1})$ does not contain a $vEB(2^k)$, and then it would be needed to update children vEB clusters and summary to new sizes.

Just to clarify, as an example, suppose we have a $vEBt(2^{32})$, this tree has a $vEBt(2^{16})$ summary and a cluster with $2^{16} \times vEB(2^{16})$ trees. To expand it to $vEBt(2^{33})$ we have to increase the cluster size from 2^{16} to 2^{17} . Then change the summary from $vEBt(2^{16})$ into $vEBt(2^{17})$. This has to be done recursively until the last summary. The resulting $vEBt(2^{33})$ has a $vEBt(2^{17})$ summary and a cluster with $2^{17} \times vEB(2^{16})$ trees. This has a recursion given by $T(u) = T(\sqrt{u}) + \sqrt{u}$.¹

To solve this recurrence let's use the master theorem.

$$T(u) = T(\sqrt{u}) + \sqrt{u}$$

$$\text{making } m = \lg u \rightarrow u = 2^m$$

$$T(2^m) = T(\sqrt{2^m}) + \sqrt{2^m}$$

$$\text{making } S(m) = T(2^m)$$

$$T(m) = T(m/2) + \sqrt{2^m}$$

$$\text{Using the Master theorem } T(m) = aT(m/b) + f(m)$$

$$a = 1, b = 2, f(m) = 2^{m/2}$$

The condition to apply master theorem is fine, $a \geq 1$ and $b > 1$

Checking conditions for case 3 of master theorem.

$$f(m) = \Omega(m^{\log_b(a+\epsilon)}), \text{ for some constant } \epsilon > 0$$

$$2^{m/2} = \Omega(m^2), \text{ check succeeded.}$$

$$af(m/b) \leq c \times f(m), \text{ for some constant } c < 1$$

$$2^{m/4} \leq 1/2 \times 2^{m/2}, \text{ check succeeded.}$$

¹ If the cluster is implemented by a dynamic table, the cost will be amortized $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$.

Then, by case 3 of master theorem $T(m) = \Theta(f(m))$

$$S(m) = \Theta(2^{m/2})$$

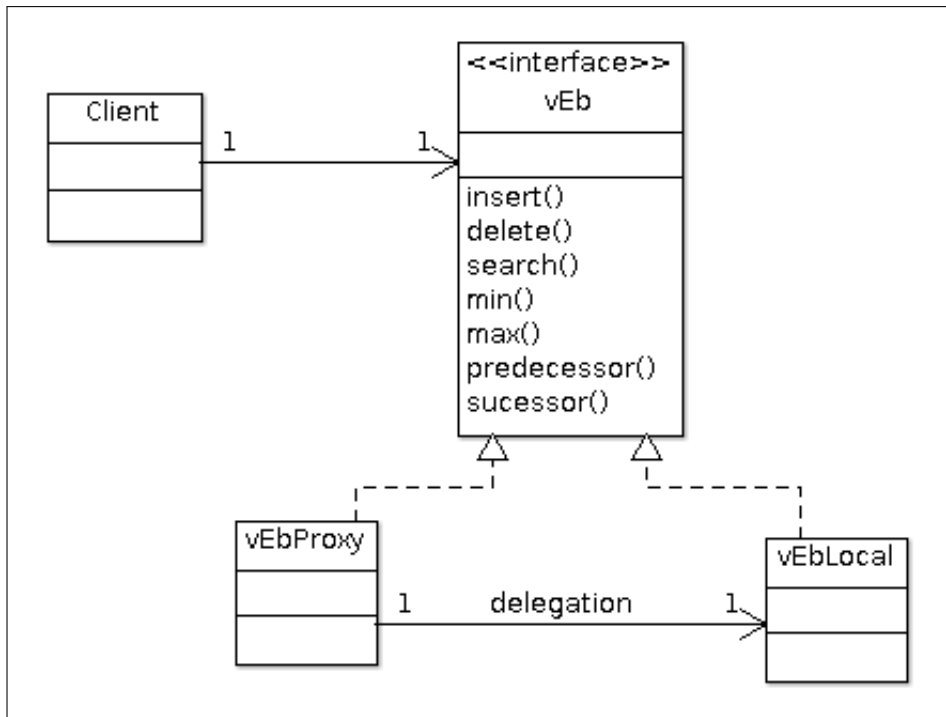
$$T(u) = \Theta(2^{(\lg u)/2}) = \Theta(\sqrt{u})$$

To expanding $\text{vEBt}(2^{33})$ into $\text{vEBt}(2^{34})$ we have to change each $2^{17} \times \text{vEB}(2^{16})$ trees in the cluster into $\text{vEB}(2^{17})$ trees. Indeed it would be even worse. We have to rebuild the whole tree because less significant bit of the “high” part will now be the highest significant bit of the “low” part. This cost will be $O(n \lg \lg u)$, where n is the number of elements present in the tree.

Those cost are not acceptable, and as we will show in the next subsections we solved the problem of the vEB cluster consuming too much memory, so, jumping from $\text{vEB}(2^{2^k})$ to $\text{vEB}(2^{2^{k+1}})$ won't be a issue anymore.

Finally, to reach this behavior in our C++ implementation we have made use of the Proxy Design Partner. As we can see in Figure 11, the Root object implements the vEB interface and delegates its calls to a vEBLocal object.

Figure 11 – Proxy Pattern applied to our vEB tree



The Root is an object of vEbProxy class and delegates all its calls to a vEbLocal object.

4.2 Challenge 2: How to make a vEB cluster with minimal memory cost

This problem could be translated into: How to replace the original cluster, implemented by a vector, so that fits in any machine and still holds its original properties, *i.e.* time cost of $O(1)$ for insert(), delete() and search() operations? The $O(1)$ time cost operations are required to stick with the time cost of $O(\lg \lg U)$ for vEB operations.

The first solution that came into mind was to replace the vector by a hash with a dynamic table, as proposed in (CORMEN et al., 2009) (pg 557). A dynamic table is an array that increases in size whenever the table becomes full. A common heuristic to expand the table is to increase it by twice the size. To analyze the cost of inserting in a dynamic table, let's suppose a table of size " i ", when there is room for data, the insertion cost is $O(1)$, when the table gets full, a new array of size " $2 \times i$ " is allocated and " i " values from previous array are copied into the new one. Notice that, each time " $i - 1$ " is an exact power of 2, the insertion cost is $O(i)$, otherwise it is $O(1)$.

The cost of a single insertion is:

$$c_i = \begin{cases} i, & \text{if } i - 1 \text{ and exaxr power of } 2 \\ 1, & \text{otherwiser} \end{cases}$$

Since the asymptotic cost varies depending on " i ", it makes more sense to calculate the amortized cost of inserting " n " elements:

$$T(n) = \sum_{i=1}^n c_i$$

$$T(n) \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$$

The first " n " is because at most n operations cost 1. The summation are the " $\lg n$ " times " $i - 1$ " becomes the an exact power of 2. Solving the summation as a sum of a geometric series, leads to:

$$T(n) < n + 2n$$

$$T(n) = O(n)$$

Thus, a hash with a dynamic table, also has suitable cost of $O(1)$ for insert(), delete() and search() (CORMEN et al., 2009) (pg 465). But a hash table has space requirement of $O(n \lg n)$ (CORMEN et al.,) and at some point after insert a huge amount of elements, no single machine will be able to store it.

We not only need to distribute the tree nodes to make our vEB scalable, we also need to somehow distribute the cluster to make it viable.

On the next sub-section we will address the tree distribution and cluster distribution

altogether because its too tight to be treated separately.

4.3 Challenge 3: Distribute the vEB tree

In order to distribute our tree we have considered two solutions. The first solution is based on calculate the IPv6 address of the remote node based on the position of the node into the whole tree in $O(1)$ time cost. The second solution is based on not knowing exactly where the hosting machine is, but instead send a multicast across the network to figure out what machine is hosting a specific node. Each solution has its caveats and we will analyze them on the next subsections.

4.3.1 Automatic IPv6 addressing

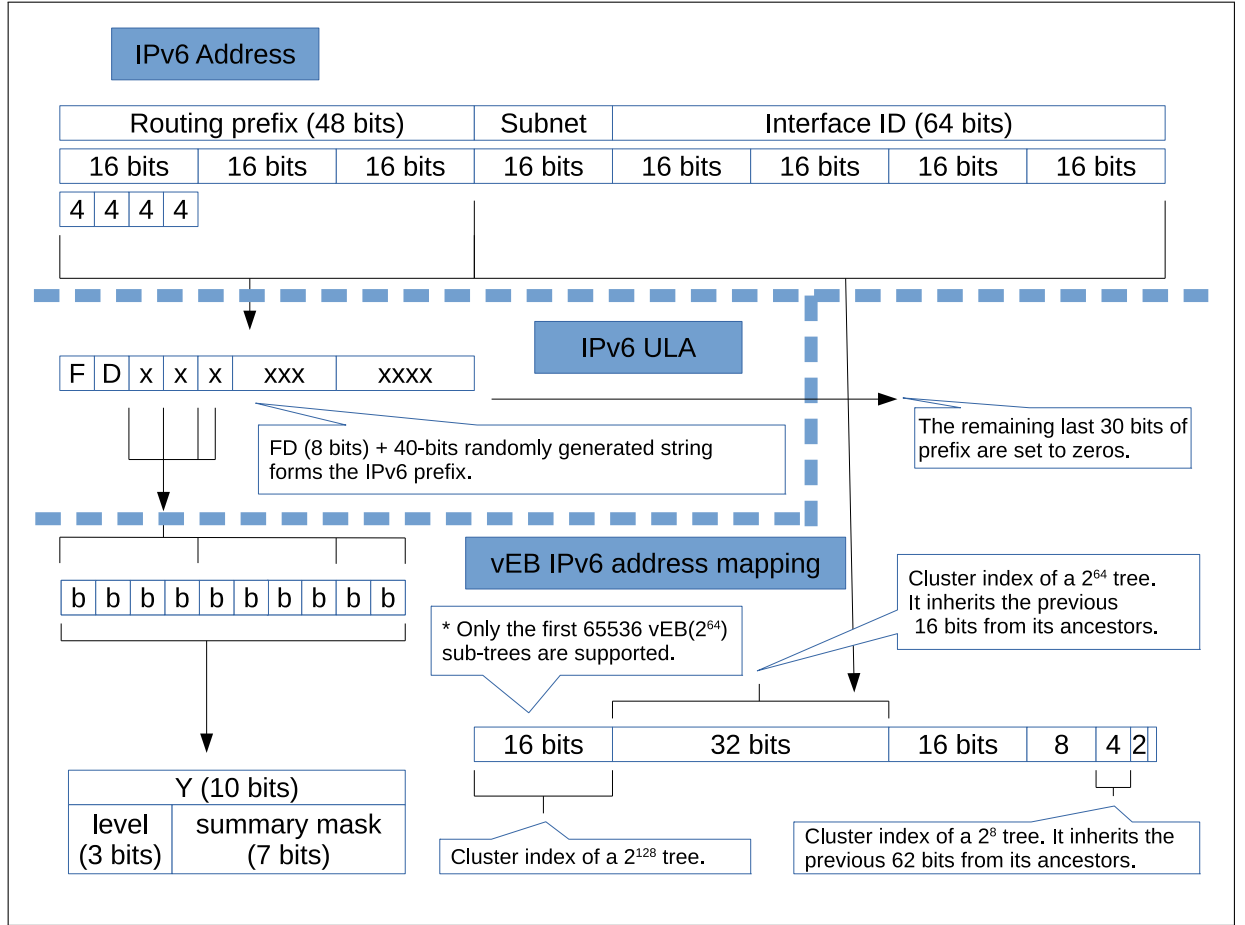
With this approach we can calculate the exactly IPv6 address ([HINDEN; DEERING, 2003](#)) of a remote vEB node, if it is a real node or a summary, and call a RPC to the machine hosting that node.

It is worth to notice that we don't want to assign, to host machines, all range of IPv6 address statically at the very beginning. It would be almost as hazard as allocate the whole cluster for a initially empty tree. We want our tree to be scalable, we don't want to have all the machines and resources statically allocated at the very beginning. It means that the IP addresses will be assigned or allocated dynamically.

Before we explain how IPv6 address are dynamically assigned or allocated to host machines, let's first explain how the position of the node in the whole tree uniquely determines the IP address of the host machine, *i.e.* let's explain how we are doing the IPv6 addressing mapping.

We are mapping nodes to IPv6 Unique Local Address (ULA) ([HINDEN; HABERMAN, 2005](#)). As you can see from Figure 12, we are replacing the most ten significant bits of the randomly generated part of the prefix with something we called "Y", and the remaining bits of the randomly generated part will be replaced by zeroes. "Y" is a value used in our mapping that encodes two things, the height (when full) of the node, represented by the first tree bits and the ancestors summary bit-mask represented by the seven remaining bits. Please have a look at Table 4 to see how these bits are encoded. Finally the 80 less significant bits of the IPv6 address are used to encode the index of a node in its cluster's parent node. Actually it not only stores the index of its cluster's parent node but also hold information of all its ancestors index in theirs cluster's parent nodes. Let's call this 'CLUSTER_ID'. The GUID of the node, *i.e.* the value that uniquely identifies a node, is made of 'Y' and 'CLUSTER_ID'. Now, for a better understanding, look at the Figure 13 as an example the address mapping. The node labeled "B" has $Y = "101\ 0100000"$, where "101" means that it is a node of universe 2^{32} ($\text{height } 5 = \lg \lg 2^{32}$),

Figure 12 – Mapping GUID into IPv6 address



and “0100000” means that this node has a summary ancestor at height 5, which in this case, is the same level as the node itself, meaning that the node is a summary. Now let’s look at node “E”, where $Y = “101\ 0000000”$, “101” to indicates height of 5, “0000000” indicates that none of its ancestors are a summary, and the $CLUSTER_ID = “0:1:0:0”$ indicates it is the node at index 1 in its cluster’s parent node. Now, node “F”, $Y = “100\ 0100000”$, “100” means a node height 4 and “0100000” means that its ancestor at height 5 is a summary. Finally, suppose the node “E” has a child pointed by its cluster at index 110 6eH, that node would have the its IPv6 address formed by $y = “100\ 0000000”$, and $CLUSTER_ID = “0:1:6e:0”$.

With this mapping we could have a tree up universe 2^{128} , *i.e.* height 7, but limited to only the first 65536 vEB(2^{64}) sub-trees that would be addressed by the 16 bits of the Sub-net field of the IPv6 address. See Figure 12. We could also take the 30 last bits of the prefix to encode the cluster index into a vEB(2^{128}), allowing to have up to 2^{46} vEB(2^{64}) sub-trees.

Now, that we already know how the position of the node in the whole tree maps to

Figure 13 – vEB node mapping into IPv6 addresses.

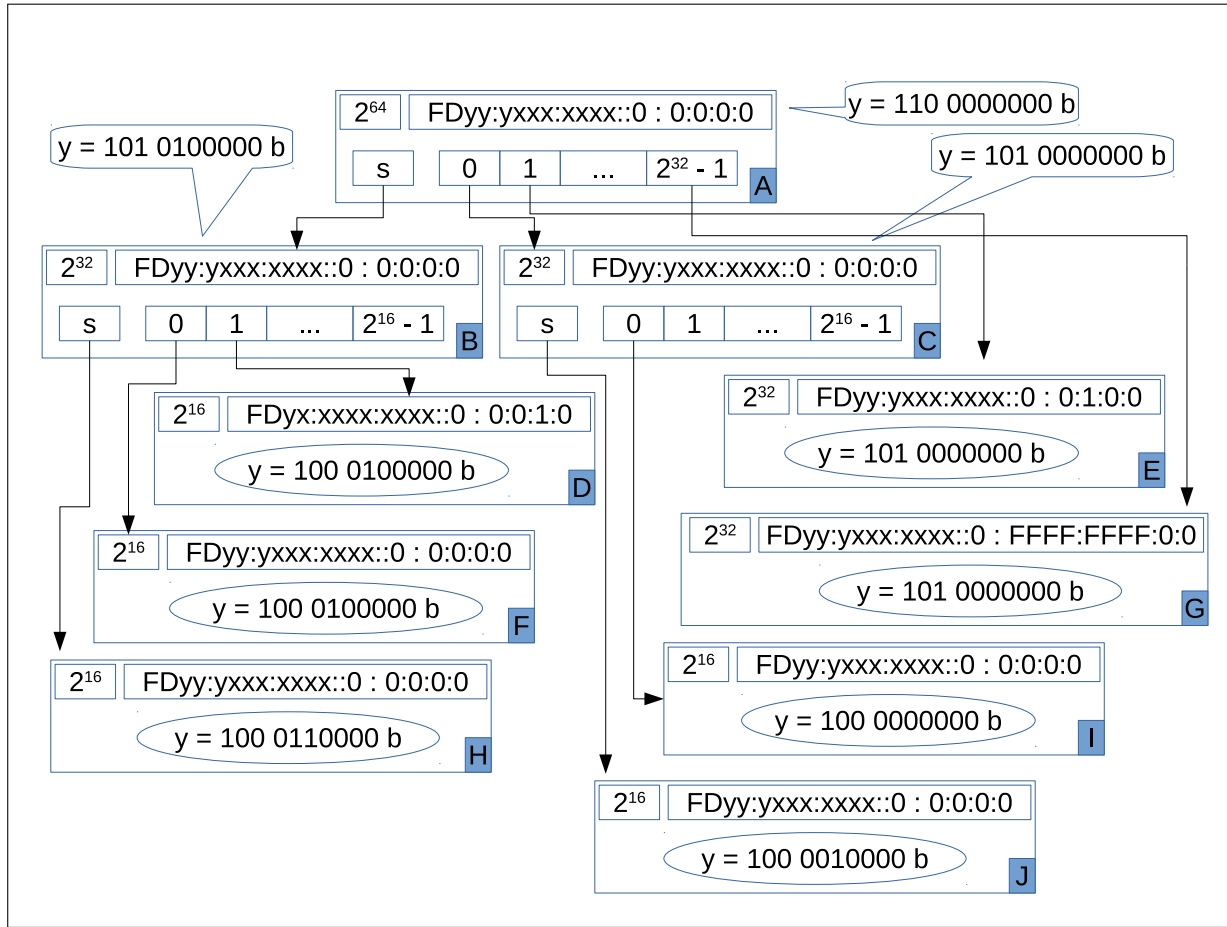


Table 4 – Y value mapped into IPv6 address

U / Y	height	summary mask	U / Y	height	summary mask
2^8	011 b	0001000 b	2^{128}	111 b	-
2^4	010 b	0000100 b	2^{64}	110 b	1000000 b
2^2	001 b	0000010 b	2^{32}	101 b	0100000 b
2^1	000 b	0000001 b	2^{16}	100 b	0010000 b

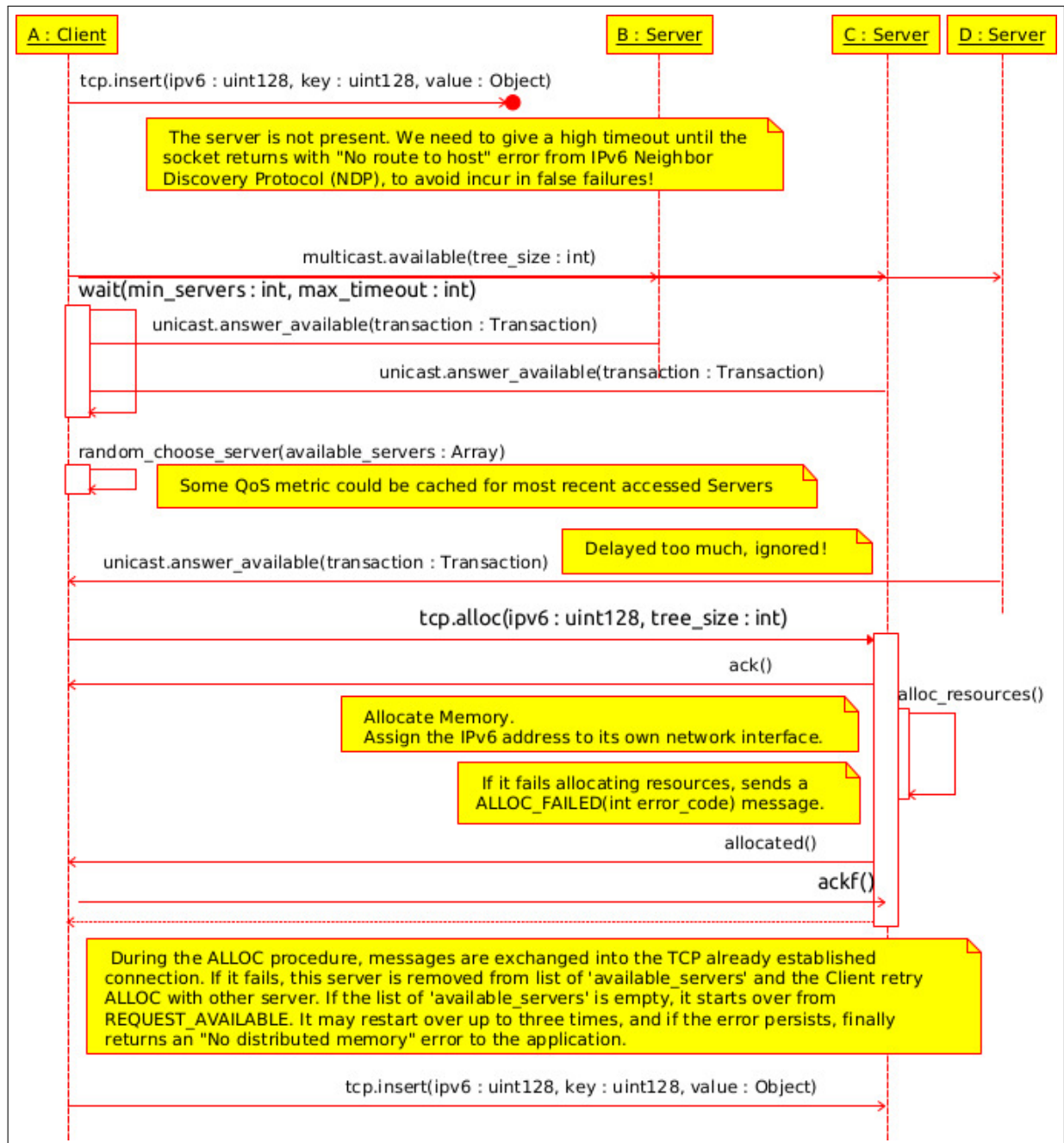
The first/fourth column represents the universe of the tree. The second/fifth column, is the 3-bits encoded height of the node. The third/last column represents the encoded ancestors summary 7-bits mask. And b stands for binary basis.

a unique local IPv6 address, let's see how we dynamically assigns host machines.

The Figure 14 depicts the sequence diagram to the dynamically addressing allocation protocol, please refer to it while we explain the process on the next paragraph.

When a vEB tree, let's say, hosted by a machine "A : Client", needs to insert a new element it calls the insert() RPC in the calculated IPv6 address for that node. Once it doesn't receive a message from that IPv6 address, it knows there is still no machine

Figure 14 – Sequence Diagram of the Automatic IPv6 solution



Notice there is no addressing discovering steps, all addresses are directed and fixed mapped from the vEB node ID. Also notice we only have TCP/IP based RPCs.

hosting that address and then starts the process of assigning a new machine to host it by sending a Multicast REQUEST_AVAILABLE message. When a machine sends its first message across the network, it appends a 32 bits randomly generated number to that message, every next message sent by that machine has that number incremented, it makes it possible to identify to what message a reply has been received. This is similar to what happens internally in TCP. Then several machines could respond with a AVAILABLE

message. Messages received into a time slice are randomly chosen by the “A : Client” to actually host the new node. Some algorithms like assign higher weight to nodes that responded faster could be applied here. By using an approach like that, with a proper time window size, helps in find a machine that probably would best serve the “A : Client” and also avoid some possible starvation that could happen if we had taken just the machine the replied first and then it keeps failing in the next step. Once a machine is chosen, “A : Client” establishes a TCP connection with that machine and sends an ALLOC message, then receives an ACK message while the host machines allocates memory, assigns the new calculated IPv6 address to its own network interface, and then send a ALLOCATED message back to “A : Client”, that sends a ACK message to the host machine that finally closes the socket. If the host machine, for some reason doesn’t receive that last ACK from “A : Client” it will just undo all changes. If for some reason, that TCP socket negotiation fails, let’s suppose the host send back to “A : Client” a ALLOC_FAILED message, or the connection is lost, “A : Client” chooses another host from the previous list of available hosts until that list becomes empty. If errors still persists after try all available host it finally fails. “A : Client” could repeat the whole process, at the very beginning, by sending again a REQUEST_AVAILABLE multicast message, up to tree times, before send an “No distributed memory” error to Application Program Interface (API) client.

In (AGUILERA; GOLAB; SHAH, 2008) is presented the design of a distributed B-Tree. To allocate a new remote node, the authors propose centralized approach in which the Client has a list of the Servers that could host the new node. They also randomly chooses the server for a list of servers and have operations to add/remove servers to/from the list. In contrast, our solution hasn’t a centralized approach, the server chosen is one of few that first answered the multicast client request, and it will probably be the ones that can best serve the Client. We are attempted to state our approach is better, but their research is different from ours, we are more committed to hold the $O(\lg \lg U)$ theoretical time cost of operations, while they are committed to propose practical solution that consider transactions and fault-tolerance. In addition, they are using B-Trees and we are using van Emde Boas trees. So, it is just worth to mention that the approaches are different and we won’t make naive comparison here. A detailed study of a practical implementation is out of scope of this research.

Our solution has some good characteristics, it allows the network to be scalable and dynamic, and also hold the theoretical time cost of $O(\lg \lg U)$ for all vEB operations. That’s because, now items of the cluster are at remote node and its addresses are calculated at $O(1)$. That’s said, let’ see the cons of this approach:

- It would be hard to scale the tree past the vEB(2^{128}) universe. IPv6 has only 128 address bits;

- Only local network machines could be used because we are mapping nodes to IPv6 ULA;
- Only the first 65536 vEB(2^{64}) trees of a vEB(2^{128}) tree would be available. Which would be represented by the fourth 16-bits group of the IPv6 address, see Figure 12;
- Eventually a Client machine Neighbor Discovery Protocol (NDP) table ([NARTEN E. NORDMARK; SOLIMAN, 2007](#)), that maps IPv6 addresses to its corresponding MAC addresses, may become full. The default max size of Linux NDP table is 1024 ([KERNEL.ORG](#),). At this point, if the number of remote nodes are greater than 1024, there is a chance that an addressed node not being present on the NDP table, and then the NDP process has to be started to discover the MAC address of the host machine, increasing the constant time to reach that node.
- Another issue to consider is that, in practice, a machine could be hosting something like 2000 nodes, *e.g.* hosting 2000 vEB(2^{16}) nodes. Looking at the Table 3 we can see that a fulfilled vEB(2^{16}) node has approximately 1.6 MB of RAM, so 2000 vEB(2^{16}) nodes will have approximately 3.1 GB of memory. It seems reasonable to think of a machine hosting 4000 (6.2GB) or even more. While it is theoretically fine for the same network interface to have 2000 or 4000 IPv6 addresses assigned to it, we haven't really tested it to check if the Operating System (OS) won't complain about it.
- For the process to assign the IP address of network interface, it must have administrative execution rights or some sort of 'Set owner User ID up on execution' (SUID);
- And lastly, the client can just figure out that a Node doesn't exist by not receiving an answer from that node, instead of receive a message from someone saying that node doesn't exist. It implies in a huge constant time cost, because we have to have a timeout huge enough to assume the node doesn't really exist.

So, let's start thinking of something else because we can't deal with the fact that our IPv6 mapping solution isn't scalable past 2^{64} universe, or more precisely past a few thousand of 2^{128} trees and with so many drawbacks.

4.3.2 Network-Agnostic

Like in the previous approach, 'Automatic IPv6 addressing', we are still mapping a unique GUID to a remote node based on its position in the whole tree. But, now, we don't map GUIDs into IPv6 addresses and assign IPv6 addresses to host machines, actually we don't assign any addresses at all.

If we don't map GUIDs into network addresses, how can we reach the remote node? We implemented a Network-Agnostic solution that finds the hosting node by sending a multicast message. The details of this solution are explored during this section.

We do still hold 'Y' and CLUSTER_ID, with their same semantics from the previous sub-section. But we are not actually mapping GUIDs to any network addresses, and thus we are not limited by the 128 bits of the IPv6 addressing. 'Y' can be of any size and able to encode up to virtually any level of vEB tree. For our implementation we have chosen a 32 bits 'Y' as depicted in Table 5, which allows us to have up to $vEB(2^{131,072})$ trees. By the way, if such tree is full, we can respond to a query in up to only 17 steps, while a regular $\log n$ algorithm could take up to 131,072 steps to process the same request.

Table 5 – Y value of Network-Agnostic node mapping

Universe	height	encoded height	summary mask
$2^{2^{17}} = 2^{131,072}$	17	10010 b	-
$2^{2^{16}} = 2^{65,536}$	16	10001 b	01000000000000000000 b
$2^{2^{15}} = 2^{32,768}$	15	10000 b	00100000000000000000 b
$2^{2^{14}} = 2^{16,384}$	14	01111 b	00010000000000000000 b
$2^{2^{13}} = 2^{8,192}$	13	01110 b	00001000000000000000 b
$2^{2^{12}} = 2^{4,096}$	12	01101 b	00000100000000000000 b
$2^{2^{11}} = 2^{2,048}$	11	01100 b	00000010000000000000 b
$2^{2^{10}} = 2^{1,024}$	10	01011 b	00000001000000000000 b
$2^{2^9} = 2^{512}$	9	01010 b	00000000100000000000 b
$2^{2^8} = 2^{256}$	8	01001 b	00000000010000000000 b
$2^{2^7} = 2^{128}$	7	01000 b	00000000001000000000 b
$2^{2^6} = 2^{64}$	6	00111 b	00000000000100000000 b
$2^{2^5} = 2^{32}$	5	00110 b	00000000000010000000 b
$2^{2^4} = 2^{16}$	4	00101 b	00000000000001000000 b
$2^{2^3} = 2^8$	3	00100 b	00000000000000100000 b
$2^{2^2} = 2^4$	2	00011 b	00000000000000010000 b
$2^{2^1} = 2^2$	1	00010 b	00000000000000001000 b
$2^{2^0} = 2^1$	0	00001 b	00000000000000000100 b

Y has 32 bits. The five first bits are used to represent the height of the tree, so it could support up 31 heights (height 0 to 30), but the last 27 bits representing the ancestors summary mask bounds this representation to support a summary at height 26, then this representation supports universe size up to $2^{2^{27}} = 2^{134,217,728}$. Notice there is no need to encode a summary at height 27 because it is already the top of the tree. Zero 'encoded height' is not used. It is reserved to mean invalid or not initialized value. We also intentionally left summary mask for height 17 unfilled because it is the top height on our experiment and won't be used. And *b* stands for binary basis.

To understand how exactly the GUID is calculated, *i.e.* Y and Cluster_ID, please have a look at code of "id.cc" transcribed in Appendix D

We could choose Y to have more bits, but as we will discuss in the next Subsection 4.3.2.1.4, limiting it to $2^{131,072}$ trees will make our protocol implementation easier. In

addition $2^{131,072}$ is approximately 10^{39456} which is roughly 10^{394} times a Googol (10^{100}), it is already a number huge enough to satisfy the needs of this research. It is important though to highlight it is not a theoretical or practical limitation.

Since we are not mapping the node GUID to IPv6 addresses anymore, we can't use only Unicast sockets to make the RPCs, we are using Multicast to start communicating with the peer (Figure 4.3.2). So, when a vEB tree wants to make a RPC to a remote node, it does by sending a Multicast message embedded with the node GUID. All the hosting machines, will look into their internal local hash table, named Registry (Appendix B), to see if that node is hosted, and then sends a Unicast Ack message back if it does so. Notice that the size of each machine's internal hash table has nothing to do with $O(\lg \lg U)$, that size exclusively depends upon the machine's RAM size and how many vEB it is hosting. For instance, as we discussed in the previous sub-section, a regular machine with 4 GB RM memory could be hosting like up to 2000 nodes, which means a hash table with up to 2000 entries only. Hash tables with dynamic tables have amortized $O(1)$ cost for insert/delete/search operations, which makes it just fine to use.

Every single node have a unique GUID. This GUID only depends on static position of the node in the whole tree. And it stays the same even if the tree expands.

Now, with our GUID, let's write a general guideline to develop a distributed vEB tree:

- Every parent node has enough information to figure out the GUID of its children and summary, *i.e.* it knows its own GUID;
- Every machine has a $O(1)$ time function that maps GUIDs, for all nodes it is hosting, to the actual vEBt objects instantiating it, *e.g.* a hash table. The hosted vEBt objects may also have its summary and children hosted somewhere else too;
- To discover what machine is hosting certain node, a multicast (or broadcast) message is sent querying it. To hold $\lg \lg U$ this must be done in $O(1)$ time. If no machine answers the query, that node is supposed to not yet exist ².
- If possible, if there is a mechanism that can respond with some sort of Nack to indicate no machine is hosting certain node, would great to avoid having to wait for the timeout. But remember it must be scalable and $O(1)$ to be considered a valid solution on theoretical field;
- To create a node, a multicast (or broadcast) is sent to query for available machines. To hold $\lg \lg U$ this must be done in $O(1)$ time;

² In future research we will evaluate if we can replace this multicast approach by a consistent hash. If that is true, this statement and following ones may change

- Once the hosting machine is discovered, messages could be exchanged in unicast or multicast (or broadcast) way.
- If a node is a local node directly pointed out by the parent it is not required, actually not desirable, to be present on the hash table;
- To allow a network to have more than one distributed tree running, messages must also carry a global identifier of the distributed tree.
- The distributed implementation layer must be completely encapsulated (Appendix B) from the rest of the code, so we can easily exchange implementation and also make then to co-operate simultaneous;
- Security is not a priority. For the time being we are assuming our tree will be used in High End Computing (HEC) and therefore security is usually provided by the OS and firewalls to avoid overhead.

Using this guideline we can pretty much have a distributed vEB on top of any protocol having multicast or broadcast messages. Unicast message are really not required but may be desirable to avoid unnecessary processing on nodes that have nothing to deal with a specific transaction.

The Network-Agnostic approach bring few advantages over IPv6 addressing one:

- This approach is not coupled with the Network layer as the previous one. The only thing we need from Network layer is Multicast (or even Broadcast) capability. In addition, we don't have to concern about machines IP addresses, leaving the network administrator assign it at will;
- We can RPC machines that are out of the LAN;
- We can have vEB trees up to virtually any size. Since we have chosen a 32 bits 'Y', we can have up to a $vEB(2^{131,072})$ tree. We could also prefix the protocol with meta information and allow 'Y' have flexible size;
- We could not have a explosive number of IP addresses blowing up NDP tables, by sending all messages as multicast. But we have decided to make use of Unicast to let other peer's network cards and CPU no bothered with unwanted messages. We do believe it can be a benefit in the future when working with several trees concurrently, or when we develop a concurrent version (KUŁAKOWSKI, 2013)(WANG; LIN, 2007), and it may also affect the power consumption of the whole network. This is hard to predict without experiments, and such experiment is out of the scope of this research due to time constraints;

- There is no need of administrative execution rights.

The Network-Agnostic approach unfortunately couldn't solve the following the IPv6 addressing issues:

- We still have to wait for a safe timeout to figure out that a node doesn't exist. Theoretically it doesn't affect the vEB $O(\lg \lg U)$ time cost, but it hits hard the constant time of the RPC and hurts the performance.

With this design in mind let's implement our own distributed protocol to prove it works and if well implemented preserves $O(\lg \lg U)$ time cost.

4.3.2.1 Minimalist UDP implementation

The choice for UDP was not at random. We already needed it for multicast messages, and it does not have the TCP three-way initial handshake delay. And in future implementation concurrent version of our tree it may be better not to keep TCP connections opened while handling requests, it could consume all ephemeral ports and need too many threads.

We also considered others UDP based protocol like CoAP and UDT, but we got raw UDP because initially we thought it would be a very simple implementation and because we wanted to have our own hand-crafted solution that could be fully customized and compared against others protocols. In addition, to make it possible to collect statistics in the way we did, some modifications to those libraries would be required. Understanding the internals of those libraries and modifying it would be very time consuming task.

We developed our protocol from the ground up, adding distributed system concepts one by one until we get a stable working version on a local network. That is why it has been called *minimalist UDP implementation*.

The next subsections describe how we implemented the basic distributed system concepts that were required.

4.3.2.1.1 Fundamental building blocks

- *Communicating entities* - From the system perspective, we have *Threads* entities calling a remote methods on tree nodes, handled also by other *Threads*. And *tree nodes Objects* if we consider a more problem oriented abstract approach. Also a *cheater Object* if the existence of a node need to be known.
- *Communication paradigms* - We are using:

- *Interprocess Communication* - Raw UDP sockets to communicate between entities;
- *Remote Invocation*
 - * *Request-reply protocols* - Request-reply protocol with ACK intermediate messages.
 - * *RPC* - We have encapsulated remote call in class the have exactly the same signature of local class. A client code using services of such class is completely unaware about the remote call.
 - * *RMI* - We have created a unique global identifier for every single node on the tree, and we actually invoke methods on objects representing those nodes.
- *Indirect Communication* - we are using *Group communication* by sending multicast messages to find out what machine will host a new tree node.
- *Roles and responsibilities* - For us, a *tree node* behaves as *Server* when it receives a method call, and also a *Client* when it forwards the call to another remote *tree node* to complete the operation. The *cheater* is also playing *Server* role.
- *Placement* - In our architecture, entities are placed in local network. We are using a placement where we map services into multiple servers. More precisely *tree nodes* playing as *Server* are randomly distributed across multiple machines. And we have an strategy to discover where *tree nodes* is so that we can requests its services.

4.3.2.1.2 Message Identifier

The first thing we created was a Message Identifier (Section 3.3), where it was named Transaction ID.

We need it to reply back to the right message caller. The same machine can concurrently handling several RPCs, dealing with different levels of the tree, dealing with concurrent access on the same tree (still no implemented) or even dealing with different distributed trees. So, even if replies are unicast we still need a Message Identifier. Threads handling a specific RPC just ignores other's replies.

A Transaction ID has the following fields in order to make it unique across the whole system:

- pid - Process ID;
- seqn - A number granted to be unique and sequential in the process;

- MAC - Network card identifier.

The pid field allows to have other trees running concurrently on the same machine. The seqn is protected by a mutex and allows to have several trees running in threads in the same process. And finally the MAC uniquely identify a machine in the network.

4.3.2.1.3 Marshalling

Initially we used Google Protobuf, then to have a bit more performance we did our own binary serialization. In most of cases we did not even bother with endianness because all machines in the experiment are Little Endian. We just did some endianness conversion for few fields we wanted to debug easily with tcpdump (Figure 75). While Google Protobuf is very powerful, and allows versioning and have ids for each field, we just implemented a very simple solution without ids and versioning, at most, for some big fields we just have a boolean to say if it is present or not. Using our hand-craft was also a bit easier to debug with tcpdump.

With Message Identifiers and Serialization we are now ready to exchange messages.

4.3.2.1.4 Framing

For this research we intentionally limited the GUID to work with up to $vEB(2^{131072})$ trees *i.e.* numbers with up to 2^{17} (131072) bits. And each of single of such number uses 16 KB of RAM memory. And our UDP packets will have up to 33792 bytes (16384 of id + 16384 of key + 1024 of meta-data). Even tough it is a huge UDP packet (Section 3.2.2) at least we send it over a single fragmented UDP packet and do not have to implement by ourselves some sort of byte stuffing for data link framing (TANENBAUM; WETHERALL, 2012) (pg 197).

It is also important to mention that IPv6 UDP packets could have UDP packets up to 4 GB size (BORMAN; HINDEN, 1999), unfortunately, again, we hadn't time to take such challenge, and it is out of scope for this research.

4.3.2.1.5 Request-Acknowledge-Reply-Acknowledge

In a general manner, all messages sent, waits shortly for an Ack reply, and then waits longer for the Result reply. If the Ack reply does not arrive in time it retries sending exactly the same message (same Transaction ID) a parametrized amount of times, if no

Ack is still received, it is assumed that node doesn't exist and then, in most cases, the `create_node()` procedure will be started. Using an Ack give us responsiveness because the Result reply may take long, specially if it needs to make others chained RPCs too. Once the Ack is received we know we can seat longer and wait for the Result reply. The base Ack timeout and number of retries are parametrized, the actual timeout and long timeout used are calculated using our congestion/flow control module (Appendix D).

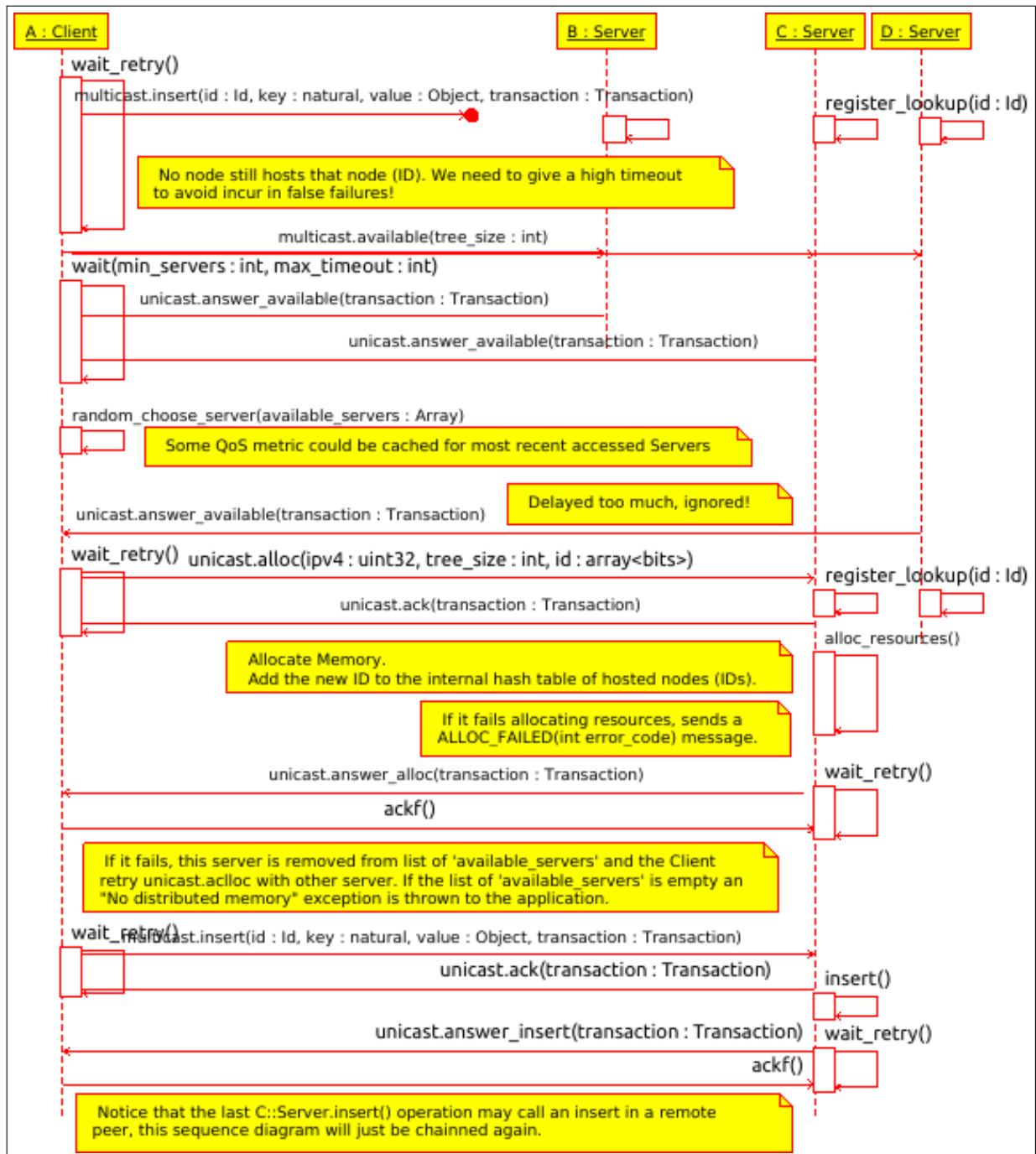
To calculate the long timeout we need some knowledge about the algorithm, or even better, the RPC vEB class that encapsulates the vEB remote implementation (classes `VebRemote` and `TreeRpcMulticast` in Appendix B) should pass, to the `long_timeout()` method in *flowcontrol.cc* (Appendix D), an additional sort of "jump_multi" parameter instead of having it globally set in the program options (Appendix A). Let's consider, as an example, vEB insert() Algorithm 4. If line 12 evaluates to true, we have 1 RPC at line 13 and 4 RPCs at line 14, if it was evaluated to false, we have only 1 RPC at line 16. So, to calculate the long timeout, we have to multiply the worst case, 5 RPCs with the number of retries and the maximum possible timeout value our flow control can return. In our implementation of vEB algorithm, we made a few tricks to avoid unnecessary RPCs, have a look in our implementation of insert algorithm to see that (lines 42 to 44 of *veb.cc* in Appendix D). By using that "there_was_something", make it possible to avoid an extra initial query to see if that cluster is empty. For other tricks like that please check our git repository.

It should be possible to calculate a "long_timeout" long enough to not indicate false timeouts but still as shortest as possible to indicate the application about an timeout error as soon as possible. Right now, since our experiments doesn't really demands a smart "long_timeout", we just implemented a very long timeout to avoid false errors. Make the implementation of what we just proposed will be left for future work.

In addition to Ack, we also have AckF (named Ack2 in few protocols out there). The Ack is send back from the host to the client in response to a method call. The AckF is send from the client to the host as a response to the Reply value sent by the host after processing the method Requested. In similar way, the host may retry sending the Reply value back to the client until it receives an AckF.

The Figure 15 depicts how RPCs are done using this approach. Actually it is quite similar to Figure 14. The differences is that some remote methods, like 'insert()' are now implemented by Multicast, and the function 'alloc_resource()', now, instead of assign a IPv6 address to its network interface, adds the ID into the local hash table. Also notice we have intentionally replaced IPv6 by IPv4 just to emphasize we are not so coupled with the Network Layer anymore. In fact, we have plans, to tests with Infinity Band too, and the designed architecture will allow such adaptation to be very simple.

Figure 15 – Sequence diagram of inserting a node in Network-Agnostic solution.



We still don't have an address discovering step because now we multicast the message. Actually, we could consider the Ack reply some sort of it. We replaced previous TCP messages from UDP unicast and multicast messages.

4.3.2.1.6 Retry

The initial version of our implementation didn't have *retries* (COULOURIS et al.,

2011), we were so naive to think we wouldn't lose packets in our local network at a controlled environment. Dealing with very intensive UDP traffic, and huge fragmented UDP packets (Subsection 4.3.2.1.4) will definitely result in packet loss. After some failures on experiments we figured out it needed *retries* and implemented it. The number of *retries* is now a program option parameter (Appendix A).

4.3.2.1.7 Discarding duplicate and History

With retries, the next thing we need is discard duplicated messages. In our implementation, that may happen in two ways.

The first, the client send a request, the host receives it, sends an Ack back and starts processing it. Then the client doesn't receive the Ack and just sends the same request again (same Message Identifier). While the host is processing the request, and for that we mean until it receives back the very last AckF, it will be present on a History. When a request is received it is checked if it is present on the History, and if so, an Ack is just replied back. Once the Ack is received, the client stops retrying the Request and starts a long wait for the final Reply. Once the Reply is received, it sends back the final AckF and finishes the task and the thread goes back to the pool.

If an Reply is received, *i.e.* it matches the PID and MAC of the process, and no task handles such message, it means that a previous thread have already received and sent an AckF back that was lost. In this case, an AckF is just send back so that host stops send Reply back. The other reason for no task handling a Reply is if it already timed out and finished, in this case, since we don't have the concept of "transactions" there is nothing to tell the host other than just AckF. This is the same situation when a host receives a Request the client has already given up since the very beginning.

For our experiments RPCs are always received in correct order and therefore are idempotent. In addition, if there is a failure, *i.e.* all retries, timeout and long timeout fails, we just abort the experiment.

We may need some more sophisticated history when dealing with transactions and a pipelined version (KUŁAKOWSKI, 2013)(WANG; LIN, 2007) of our tree in future research.

4.3.2.1.8 Flow control and Congestion control

With retries our network traffic becomes even more intensive. Definitely we need flow control.

As you can see in *flowcontrol.cc* (Appendix D), every retry we multiply the timeout value by a power of 2. The initial attempt will be $timeout \times 2^0$, then on the next retry it will be $timeout \times 2^1$. The amount of retries required to work is saved globally so that the next RPC will start from $timeout \times 2^{last_step}$. Each time a RPC fails "last_step" increments until the maximum number of retries. For every four consecutive succeeded RPCs, the "last_step" decrements until zero.

Once the timeout is calculated using that algorithm, it is added to a randomly generated value between 1 and $timeout/2$, this is our congestion controls mechanism.

Until all these techniques listed in this section we couldn't have our distributed tree performing the tests successfully.

4.3.2.1.9 Cheater

The Cheater wasn't required but helped a lot to finish experiments quickly. It tracks all nodes created (Figure 16). Then, when some *Client* sends a message to a node, if that node does not exist, the Cheater replies with a Nack message carrying a "has" parameter set to *false*, indicating that no machine hosts a vEB node identified by the requested GUID. This makes the performance better because the *Client* does not have to wait for a timeout/retry to figure out the node doesn't exist.

To use the Cheater, the root/nodes has to be configured with the option "has_cheater=true" in order to send that extra message to the Cheater.

Actually that extra message may not be needed. Just wait for an AckCheater on "A:Client" and send_retry a sort of "multicast.answer_cheater_alloc()" would be enough.

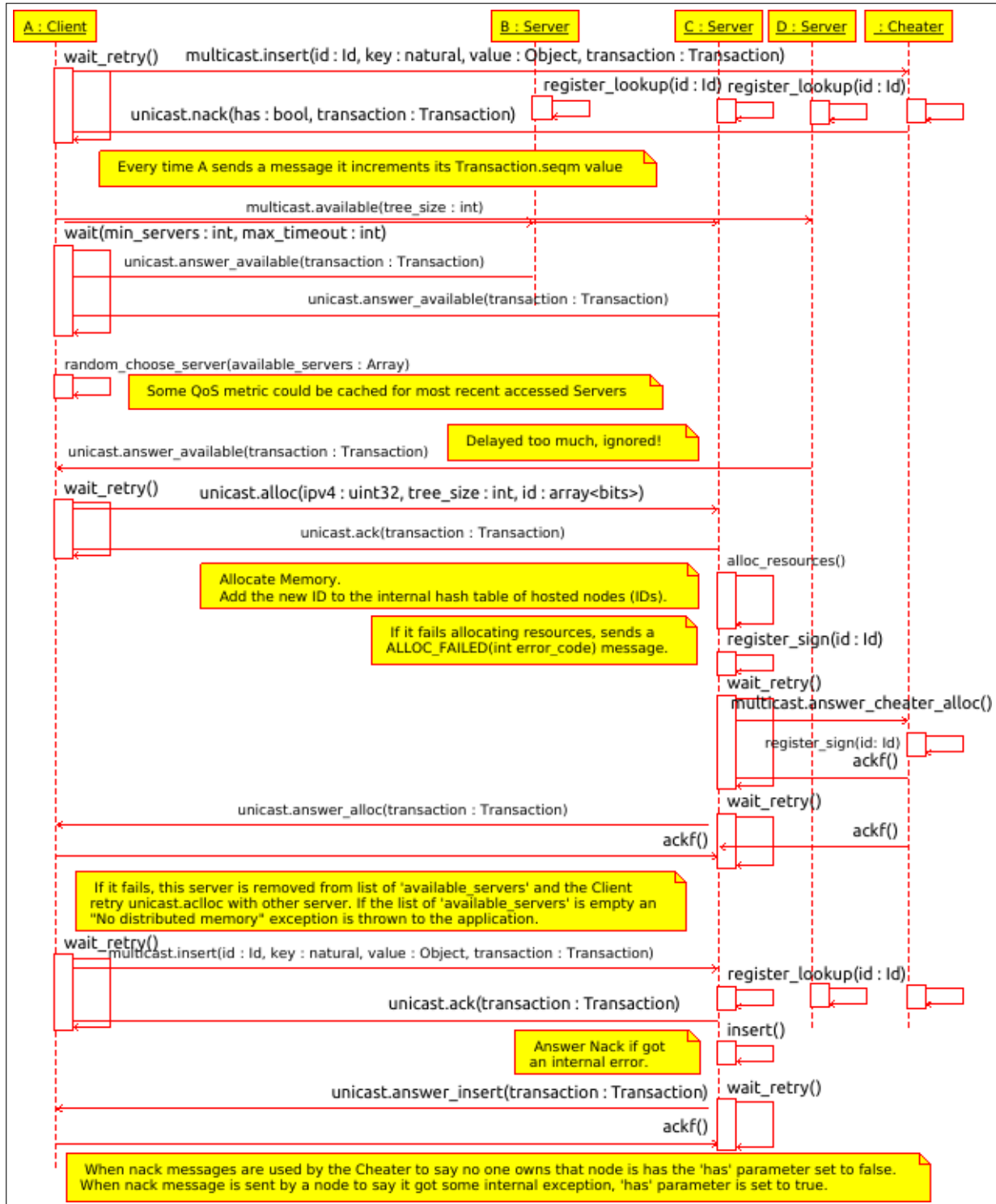
Unfortunately, the Cheater ends up on the same problem we was initially trying to get ride of. At some point the the tree becomes really huge, the Cheater won't be able to hold the list of existing Ids anymore. Remember that a single GUID on a vEB(2^{17}) can have up to 16KB.

Probably, for the Cheater, we could use a solution like ZHT (LI et al., 2013) with a limitation of $2^{64}/partition_size$ nodes (Table 6) may be enough for real world. Actually, for real world problems we could not even need the cheater and implement our whole tree on top of ZHT.

We could also implement our own consistent hashing (KARGER et al., 1997). Actually, a consistent hash could also be considered for the implementation to replace our multicast solution, we won't consider this approach in this research tough.

Others Distributed Hash Tables (DHT) with zero-hop routing could also be considered. Memcached also has $O(1)$ routing but has a limited key size and doesn't allow

Figure 16 – Sequence diagram of inserting a node with Cheater.



The same sequence of the previous diagram without the Cheater to speed up the process by sending Nack messages

dynamic membership. From the DHTs analyzed in Table 6 seems ZHT would be our best choice and non $O(1)$ would be immediately discarded.

Table 6 – Comparison between DHT implementations

Name	Impl.	Routing time	Persistence	Dynamic membership
Cassandra	Java	$\lg n$	Yes	Yes
Memcached	C	2	No	No
C-MPI	C/MPI	$\lg n$	No	No
Dynamo	Java	0 to $\lg n$	Yes	Yes
ZHT	C++	0 to 2	Yes	Yes

Comparison between DHT implementations. ZHT seems to be the only option for our needs. It uses C++, it is $O(1)$ and scalable.

Is it also worth to notice that it is not a complete research to propose a practical solution to a distributed vEB tree. This research is more committed to keep the theoretical $O(\lg \lg U)$ time cost for all vEB operations. Even though it is a theoretical research, we brought up several practical concepts to the discussion because we have future plans to extend this research in practical fields.

This list below summarizes some of basic distributed system concepts ([COULOURIS et al., 2011](#)) we had to concern with in this minimalist working implementation:

- Message Identifiers - We are using a sequential number, the process id and mac address;
- Timeouts - We have a progressive timeout the increments when there is a timeout and decrements and there is a few succeed acknowledges (*flowcontrol.cc* in Appendix D);
- Lost reply - All operations wait a final acknowledge from the reply to be considered done;
- History - We really don't have it because all replies waits a final acknowledge.
- Style of exchange protocol - We are using something similar to RRA, but we have an acknowledge message sent from server to client when the request is received. Request-Acknowledge-Reply-Acknowledge (RARA). This is used because the actual reply can that long, and having this acknowledge allows the client to immediately know if the server has received the request.
- Framing - Wasn't required;
- Marshalling - Proprietary;
- Flow/Congestion Control mechanism.

5 Experiments

5.1 Planning

We intended to run some experiments to prove that our distributed vEB tree holds $O(\lg \lg U)$ time cost, then we naturally need to implement it first.

Our vEB tree was implemented in Modern C++14 ¹, using Boost library ² as a testing and logging framework. Cmake ³ was used as a compiling tool and Qt Creator ⁴ as the IDE (Interface Development Environment). And finally lib GMP ⁵ (GNU Multiple Precision Arithmetic Library) has been used to handle number greater than 2^{64} , *i.e.* greater than 64 bits.

Theoretically GMP could handle could handle with up to 2^{37} bits on 64-bits machines. This is due to its internal representation having a 31-bits integer as counter of 64-bit limbs, which is 2^{37} bits ($64 \text{ bits} \times 2^{31} = 2^6 \text{ bits} \times 2^{31}$). Notice that such a number uses 16 GB of memory. In our experiments we are working with 2^{17} bits, *i.e.* numbers that uses 16 KB of memory and can represent decimal numbers with almost 40 thousand digits. It seems to be insane to work with single numbers having more than 16 GB space, but if we would like to go past that, we would need to replace GMP by something else.

In order to conduct our experiments and make it easy to conduct all future works mentioned in the Conclusion (Chapter 6), we have designed an architecture that have several blocks of functionality that can be effortlessly replaced or configured to model new experiments. We have compiled a list of requirements for this architecture and designed it in Appendix B.

Before run the experiments we first needed to check the correctness of our implementation. We did so by comparing the output of our tree for all vEB operations against the output of a very basic structure based on C++ STL vector implementation. The correctness test has succeeded and its source code is depicted in “void test_sanity(std::vector<T> & tdata)” in *test.cc* (Appendix D).

If you want to reproduce correctness test and experiments, or get more detailed information, please check Appendix C.

The following subsections describe the experiments.

¹ <<https://isocpp.org/>>
² <<http://www.boost.org/>>
³ <<https://cmake.org/>>
⁴ <<https://www.qt.io/ide/>>
⁵ <<https://gmplib.org/>>

5.1.1 Experiment 01 - Dense tree

The idea of this experiment is to analyze how a dense tree behaves, more precisely, analyze its behavior as the number of elements grows. In this experiment, we use a $\text{vEB}(2^{24})$ tree that supports up to 66536 elements, and incrementally insert all these elements until it becomes 100% dense, *i.e.* full.

Before we explain how the experiment is performed and statistics are collected, let's recall the concept of *depth* and *height* because it will be crucial for analyzing the results. Depth of a node is how far it is from the root. The root has depth zero. Height of node is the distance from node to the leaf on the longest path. In this chapter, when we mention *depth*, we mean the highest depth an operation went down in the tree to be completed. See Tables 7 and 8 for possible depths on $\text{vEB}(2^{24})$ and $\text{vEB}(2^{2^{17}})$ trees respectively. Let's also recall that *level* is *depth* + 1, and we also refer to *level*, in this chapter, as the deepest level an operation had to access to be completed.

Table 7 – $\text{vEB}(2^{24})$ depths

Universe	bits	Node depth	Height
$2^{24} = 2^{16}$	16	0	4
$2^{23} = 2^8$	8	1	3
$2^{22} = 2^4$	4	2	2
$2^{21} = 2^2$	2	3	1
$2^{20} = 2^1$	1	4	0

Table 8 – $\text{vEB}(2^{2^{17}})$ depths

Universe	bits	Node depth	Height
$2^{2^{17}} = 2^{131072}$	131072	0	17
$2^{2^{16}} = 2^{65536}$	65536	1	16
$2^{2^{15}} = 2^{32768}$	32768	2	15
$2^{2^{14}} = 2^{16384}$	16384	3	14
$2^{2^{13}} = 2^{8192}$	8192	4	13
$2^{2^{12}} = 2^{4096}$	4096	5	12
$2^{2^{11}} = 2^{2048}$	2048	6	11
$2^{2^{10}} = 2^{1024}$	1024	7	10
$2^{2^9} = 2^{512}$	512	8	9
$2^{2^8} = 2^{256}$	256	9	8
$2^{2^7} = 2^{128}$	128	10	7
$2^{2^6} = 2^{64}$	64	11	6
$2^{2^5} = 2^{32}$	32	12	5
$2^{2^4} = 2^{16}$	16	13	4
$2^{2^3} = 2^8$	8	14	3
$2^{2^2} = 2^4$	4	15	2
$2^{2^1} = 2^2$	2	16	1
$2^{2^0} = 2^1$	1	17	0

We collect statistics for each operation (*insert*, *successor*, *predecessor*, *search*, *remove*) separately. For each operation, we also collect statistics for each *depth* individually. As an example, there will be distinct statistics, for *insert at depth 1*, *insert at depth 2*, *successor at depth 1*, *successor at depth 2* and so on.

The statistics are also separated in groups of 2000 operations and contains, average time taken, standard deviation, minimal and maximal time taken. Finally these statistics

are further collect in four sets. The first contains only operations that have not timed-out/retried. The second is like the first without outliers. The third contains all operations and the fourth is the same without outliers. Samples that are at least 3 times the standard deviation away from the mean are considered as outliers.

In this document we will present only statistics from the second group, *i.e.* operations that hasn't timed-out/retry and are not outliers.

Now let's describe how the experiment is performed and how these statistics are collected.

At the very beginning an empty vEB tree and a shuffled vector containing numbers from 0 to 65535 are created..

Then the first 2000 elements of the vector are inserted into tree and the statistics for *insert()* operations are collected. After that, *successor()* calls, for each of these 2000 elements, are performed on the tree and its statistics are collected. The same is done for *predecessor()* and *search()* operations.

Notice the tree now contains 2000 elements.

Then, the next 2000 elements of the vector are inserted and its statistics collected. After that, once again, *successor()*, *predecessor()* and *search()* are performed and its statistics collected.

It is then repeated until all the 65535 elements from initial vector are processed and the tree contains these 65536 elements. Notice the last iteration has only 1536 elements, and there was 33 interactions at total. Also notice the tree has become more dense after each interaction, reaching 100% density after the last interaction.

After that, the first 2000 elements of the vector are removed from the tree and the *remove()* statistics are collected. This is repeated for the next 2000 elements until all of them are removed. Now, it is the opposite, the tree is becoming more sparse after each interaction.

Now let's dig a bit more on some conditions on which experiments are running. It is very important to notice the test is running with option "multicast_loopback" set to false, this together with the Factory rules we implemented in "factory.cc" (Appendix D), will always make every vEB node to have its summary and cluster elements hosted in a another machine. It means that for each *level* it gets down, there will be RPC going underneath. It is crucial for our analysis because otherwise, we would be dealing with constant times of completely different magnitudes, one being a method call in CPU/memory and the other being a serialized RPC call across the network. In other words, we are always forcing RPC between *levels*. Of course, for a practical real solution a RPC should only be done when a machine has no memory to make a local vEB to host itself.

Another tricky we made to make our analysis easier was to set "force_maxsize" true (Appendix A). Without that, smaller vEB trees would transfer smaller UPD packets, much faster and with much less losses. So, we are forcing all RPCs to take the same time even if it needs to transfer much less data. Without that, a RPC call to a $vEB_t(2^{2^{17}})$ node would carry 33 KB payload (16 KB node ID + 16 KB key + 1 KB metadata) while a RPC call to $vEB_t(2^{32})$ node would take 1032 Bytes payload (4 Bytes node Id + 4 Bytes key + 1 KB metadata). Sending 33 KB needs IP fragmentation while 1 KB might fit in MTU. Metadata contains method identifier, transaction identifier, caller identifier and statistics. It could be much smaller but we set 1KB in our experiments to just not have to tweak it.

For these experiments we have 3 different program configurations, and each machine will be running the right one according to Table 11.

For a complete description of the application parameters please refer to Appendix A.

Please refer to Appendix C.2 for the configuration files for this experiment.

5.1.2 Experiment 02 - Sparse tree

The idea of this experiment is to analyze how a sparse tree behaves, more precisely, analyze its behavior as the number of elements grows. We will start with an empty tree $vEB(0)$ and insert 65536 keys of 131072 bits.

The keys are generated dynamically with lib GMP API using Twister Algorithm (MATSUMOTO; NISHIMURA, 1998), and we grant it will be unique (source code "test.cc" in Appendix D). The order of elements will also be randomized. It will be the hash value order. The hash for those values are calculated using "boost::hash_combine" as you can see in "natural.hh" source code in Appendix D.

Even after inserting those 65536 elements the tree will continue extremely sparse with keys spread out across its universe. The universe of this tree is so huge that would need almost 40 thousand digits to express it with a decimal number.

To understand how this experiment is performed and statistics are collected please refer to "Experiment 01" (Subsection 5.1.1). Everything from Experiment 01 applies here. The only difference is the size of generated random keys. In this experiment we use 131072 bits while in "Experiment 01" uses only 16-bits keys.

Please refer to Appendix C.3 for the configuration files for this experiment.

Table 9 – Number of keys by level of a vEB(65536)

Universe	bits	Depth	Height	Keys	Keys(found)	Cumulative keys(found)
$2^{2^4} = 2^{16}$	16	0	4	$2 \times 1 = 2$	2	2
$2^{2^3} = 2^8$	8	1	3	$2 \times 2^8 = 512$	510	512
$2^{2^2} = 2^4$	4	2	2	$2 \times 2^8 \times 2^4 = 8192$	7680	8192
$2^{2^1} = 2^2$	2	3	1	$2 \times 2^8 \times 2^4 \times 2^2 = 32768$	24576	32768
$2^{2^0} = 2^1$	1	4	0	$2 \times 2^8 \times 2^4 \times 2^1 = 65536$	32768	65536

Considering the vEB is full. Keys are the number of “min” and “max” values at each level. Keys(found) are the number of keys that will be found by a “search()” operation at that level. Notice the “min” and “max” keys were found on earlier levels. And Cumulative keys(found) are the number of keys found until that level by a “search()” operation, *i.e.* the sum of Keys(found) till that level.

5.2 Results

5.2.1 Experiment 01 - Dense tree

5.2.1.1 Insert

In this section we consolidate, in five graphs, the statistics collected for *insert()* operations on a dense tree and analyze them.

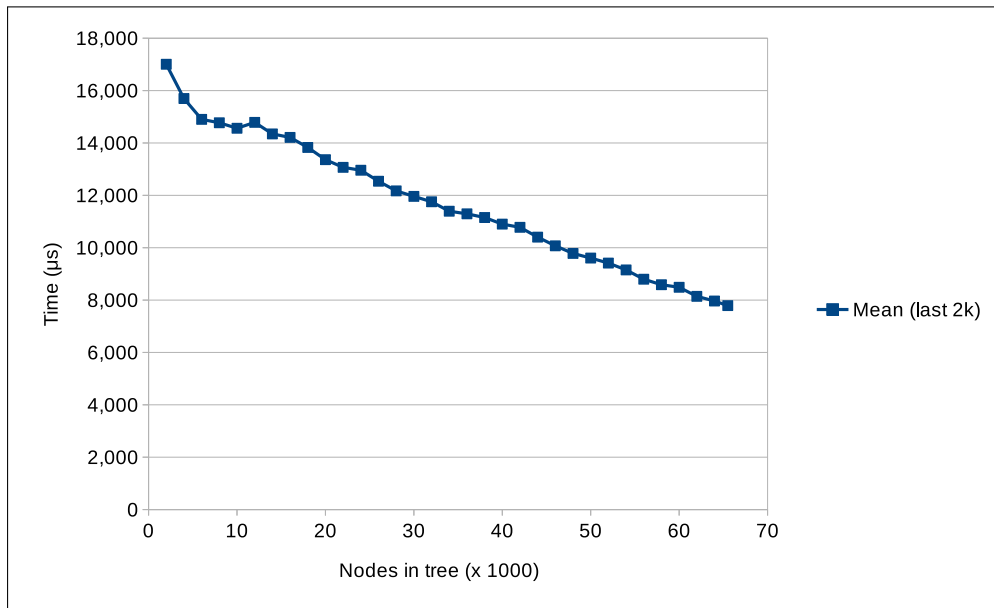
First graph, Figure 17, shows how the *insert()* average time evolves as the tree becomes more dense.

Second graph, Figure 18, shows how the average *depth* taken by *insert()* operations evolves as tree becomes more dense.

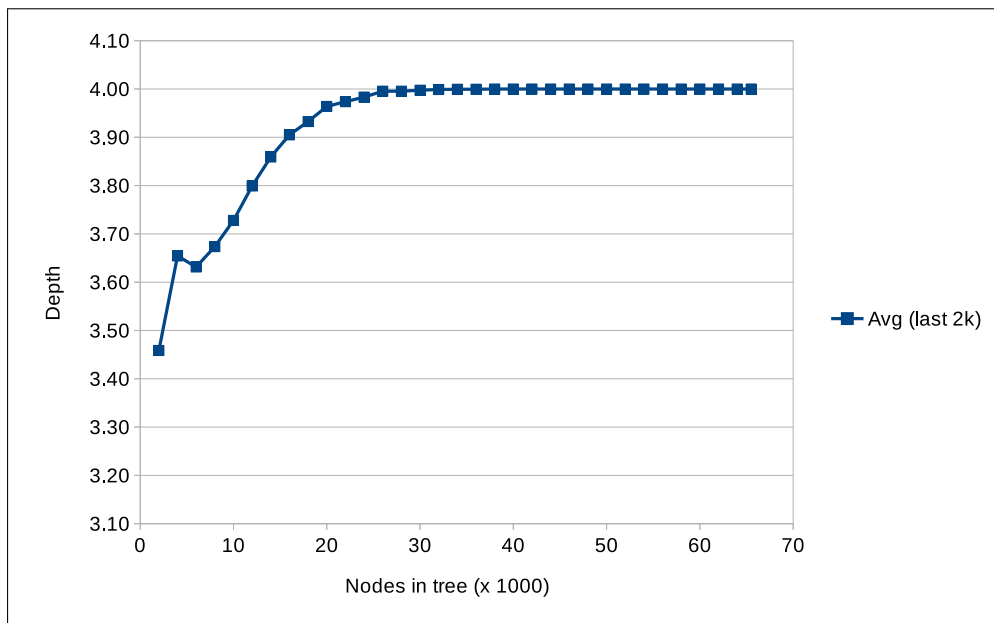
Third graph, Figure 19, shows how *depths* taken by *insert()* operations are distributed as tree becomes more dense.

Fourth graph, Figure 20, shows how the *insert()* average time, for each *depth*, evolves as the tree becomes more dense.

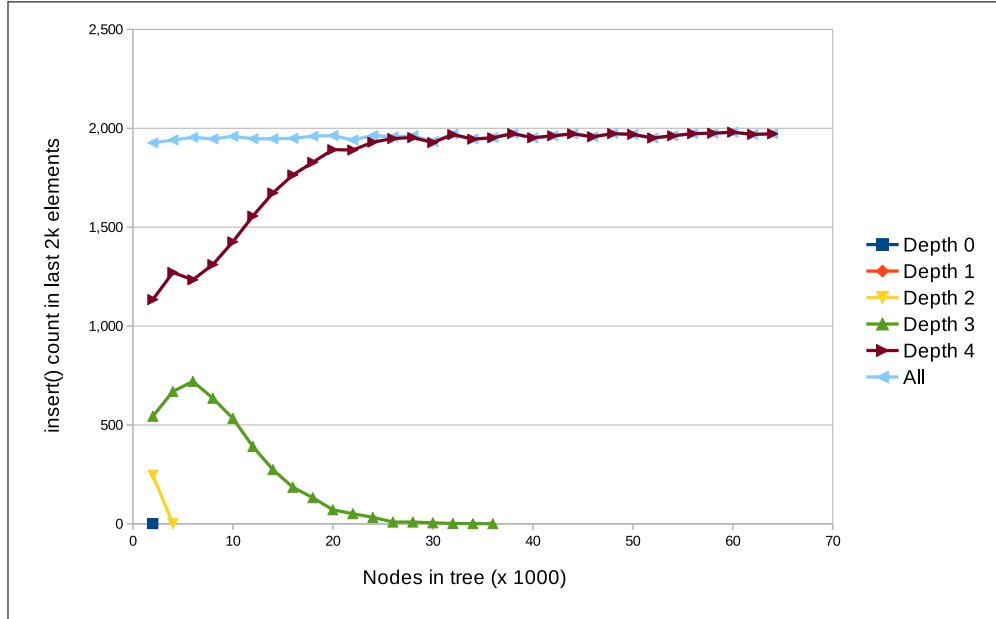
And fifth graph, Figure 21, shows the overall average time taken by *insert()* operations for each *depth*.

Figure 17 – *insert()* mean time by nodes in a dense $vEB(2^{24})$.

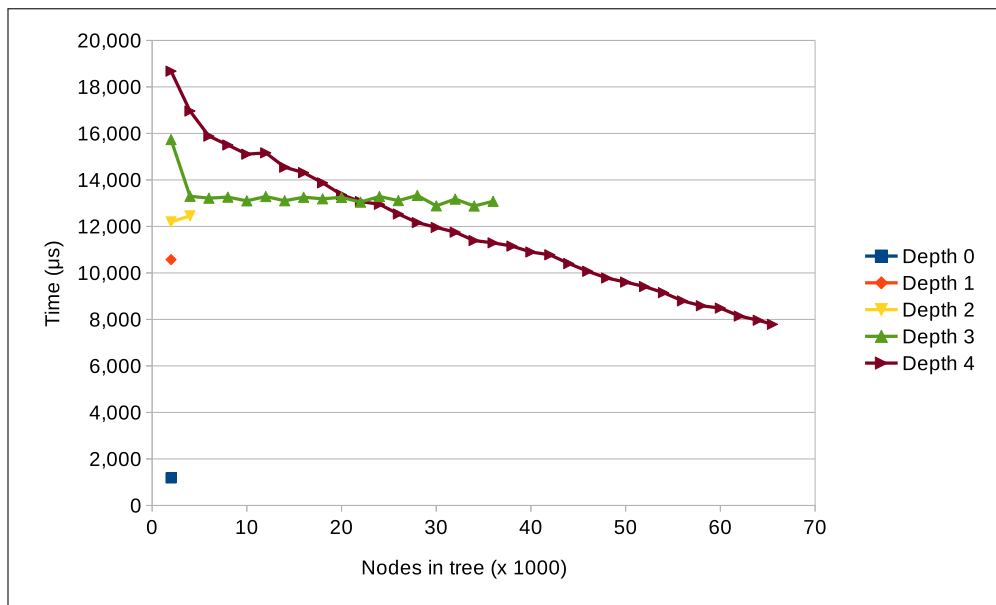
The *insert()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 18 – *insert()* average depth by nodes in a dense $vEB(2^{24})$.

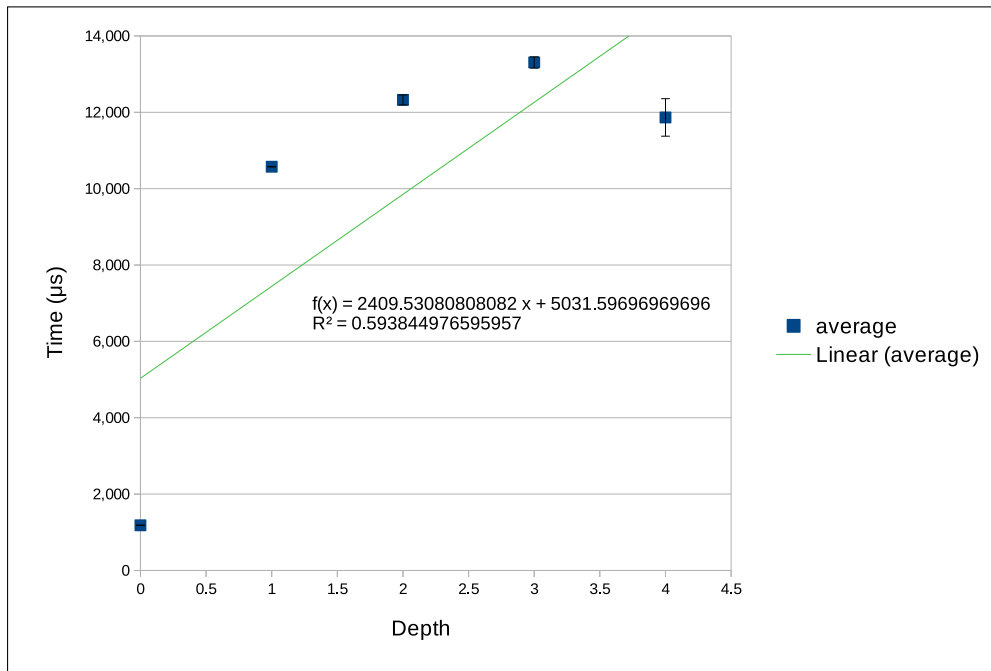
The *insert()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 19 – *insert()* depths count, of last 2k elements, by nodes in a dense $vEB(2^{24})$.

A counting of depths taken by *insert()* operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 20 – *insert()* depths time by nodes in a dense $vEB(2^{24})$.

The *insert()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 21 – *insert()* depth average time of a dense vEB(2^{24}).

The average time taken by *insert()* operations per depth.

In Figure 17 the average time taken by *insert()* operations becomes faster as the tree gets more populated, despite the fact that the average level gets higher (Figures 18 and 19). This happens because time taken for *insert()* operations at deepest level decreases (Figure 20). It is not hard to understand why. The insertion on a new node takes 4 RPCs to create it plus 1 RPC to insert it, while the insertion on a existing node only takes 1 RPC to insert it. For instance, on best case, the insertion at *depth* 3 takes 4 (1+1+1+1) RPCS, and on worst case scenario 23 RPCS ((4+1)+(4+1)+(4+1)+(4+4)).

Unfortunately, we can't still figure out why at deepest *level* it becomes faster than the previous *level* as you can see in Figures 20 and 21.

Actually we have an explanation in mind but we can't really confirm that. At initial *levels*, while tree is still sparse, it has to create a lot of nodes down the path, since the insertion use keys at random order. When the tree becomes dense, there won't be any insertion at initial *levels* anymore, because even if the key is placed at a lower *level*, at "min" slots, it has to push down the key that was previously there. Notice the path down the pushed key is already created, and our statistics we consider the deepest *level* accessed by an operation to be completed. Table 9 shows how many keys will be sitting on each level of a vEB(2^{24}) tree. When the tree is half full, all levels other than the last one will be almost full.

5.2.1.2 Successor

In this section we consolidate, in five graphs, the statistics collected for *successor()* operations on a dense tree and analyze them.

First graph, Figure 22, shows how the *successor()* average time evolves as the tree becomes more dense.

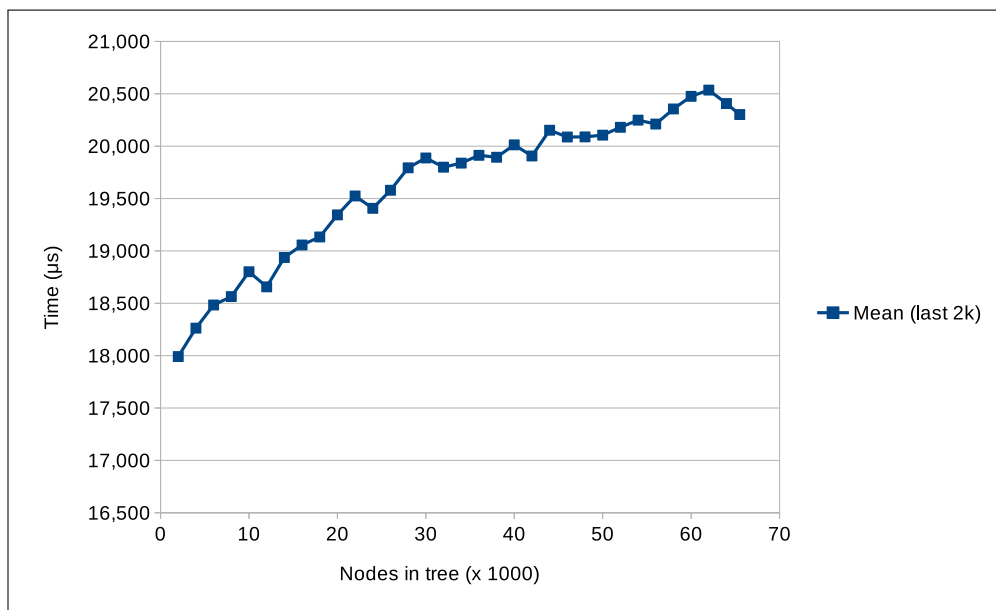
Second graph, Figure 23, shows how the average *depth* taken by *successor()* operations evolves as tree becomes more dense.

Third graph, Figure 24, shows how *depths* taken by *successor()* operations are distributed as tree becomes more dense.

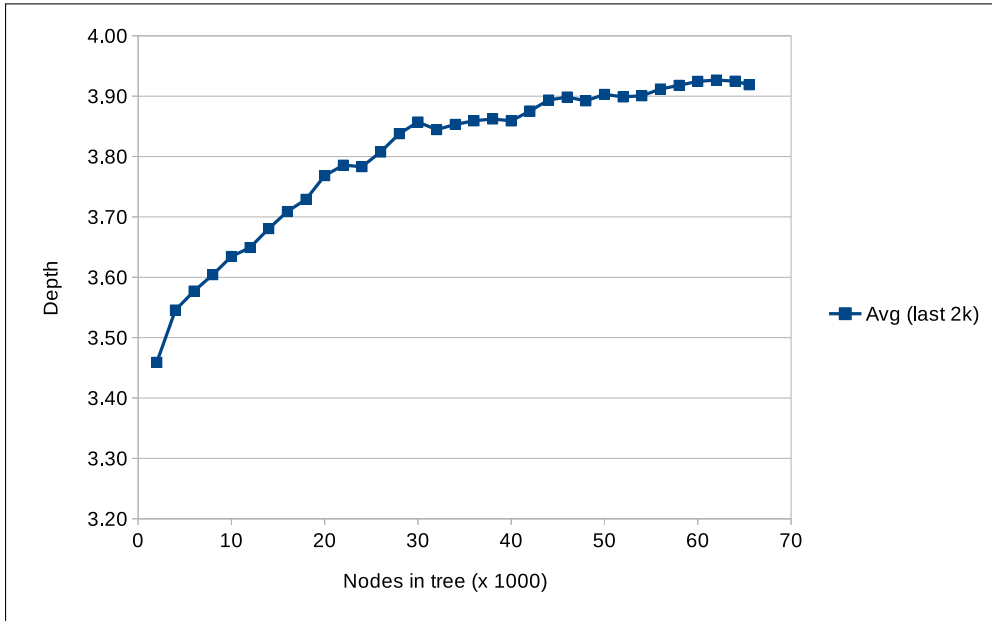
Fourth graph, Figure 25, shows how the *successor()* average time, for each *depth*, evolves as the tree becomes more dense.

And fifth graph, Figure 26, shows the overall average time taken by *successor()* operations for each *depth*.

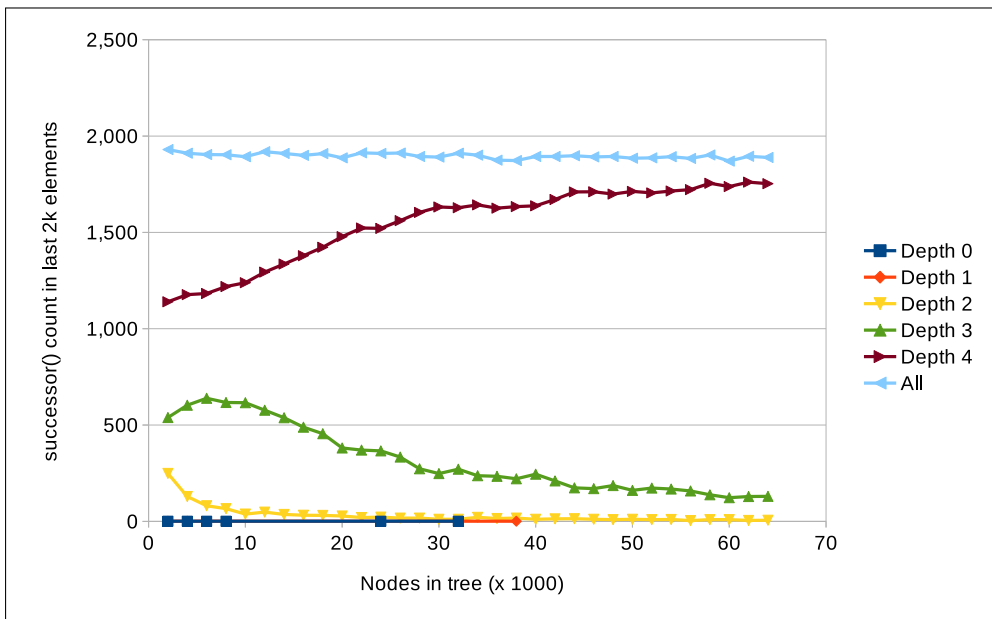
Figure 22 – *successor()* mean time by nodes in a dense vEB(2^{24}).



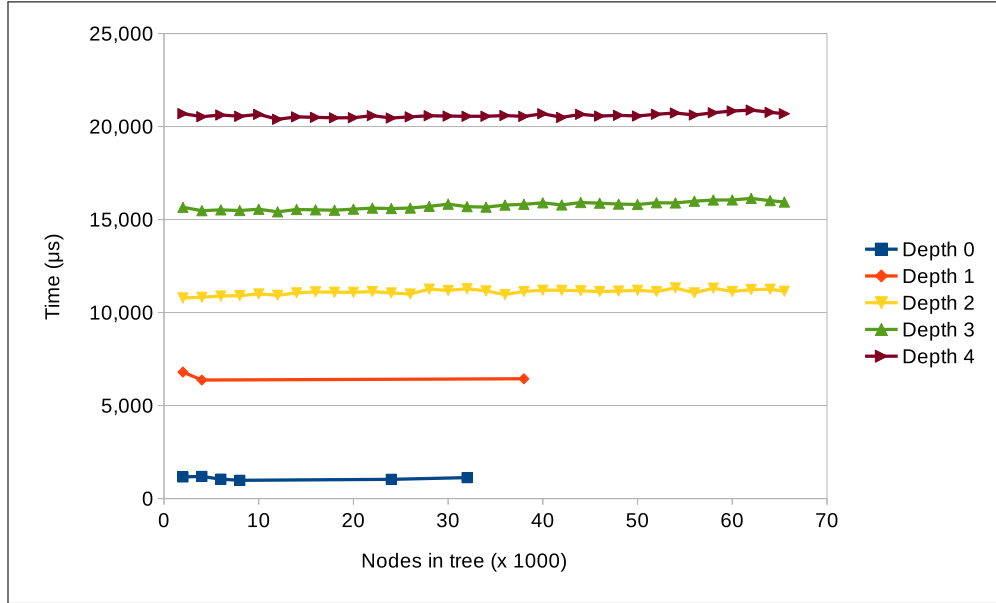
The *successor()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 23 – *successor()* average depth by nodes in a dense vEB(2^{24}).

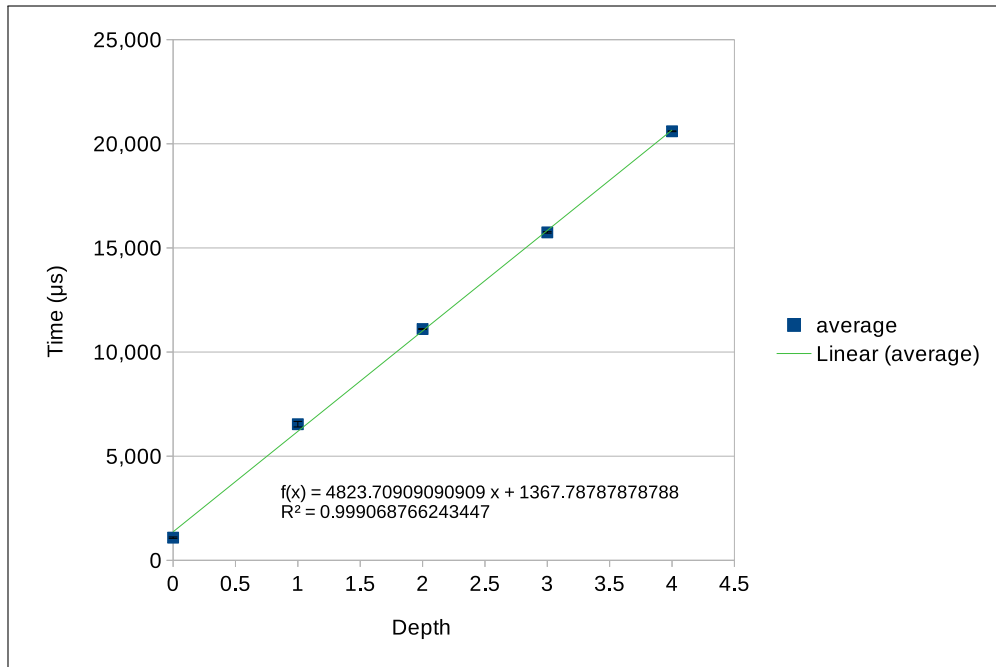
The *successor()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 24 – *successor()* depths count, of last 2k elements, by nodes in a dense vEB(2^{24}).

A counting of depths taken by *successor()* operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 25 – *successor()* depths time by nodes in a dense $vEB(2^{24})$.

The *successor()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 26 – *successor()* depth average time of a dense $vEB(2^{24})$.

The average time taken by *successor()* operations per depth.

As you can see from *successor()* Algorithm 2, it stops recursion when it hits the min value, and digs down as the tree becomes populated because chances to hit the minimal

values on nodes will decrease. It is easy to see the average *depth* increasing on Figures 23 and 24.

What could bring our attention is the Figure 25, it seems each level is slightly increasing its time. We bet it is just a fluctuation because we got the same thing on Experiment 2, but when we repeated Experiment 2 we got a slightly decreasing trend. We haven't repeated this experiment tough, what would be desired to confirm that.

5.2.1.3 Predecessor

In this section we consolidate, in five graphs, the statistics collected for *predecessor()* operations on a dense tree and analyze them.

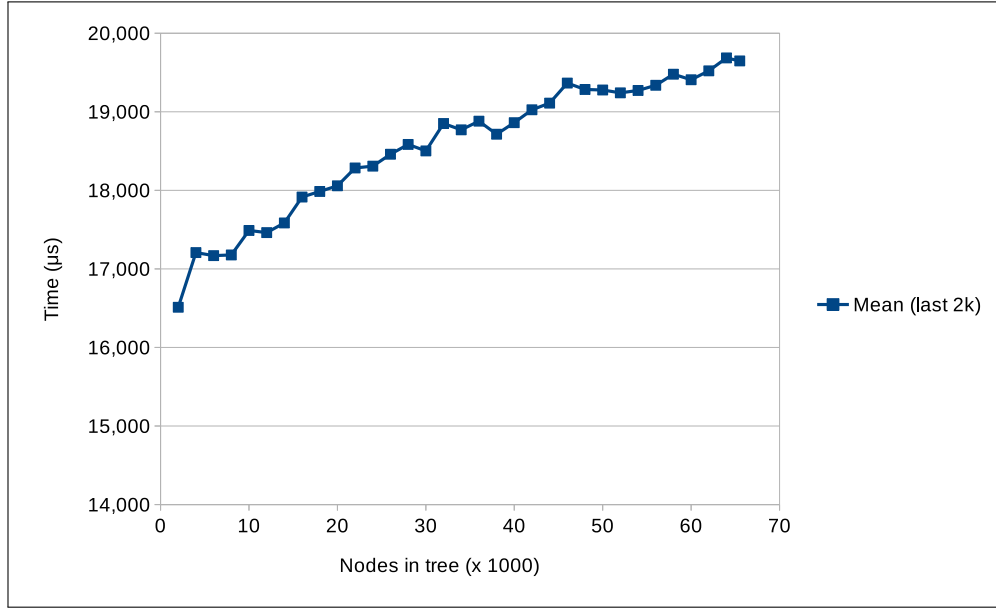
First graph, Figure 27, shows how the *predecessor()* average time evolves as the tree becomes more dense.

Second graph, Figure 28, shows how the average *depth* taken by *predecessor()* operations evolves as tree becomes more dense.

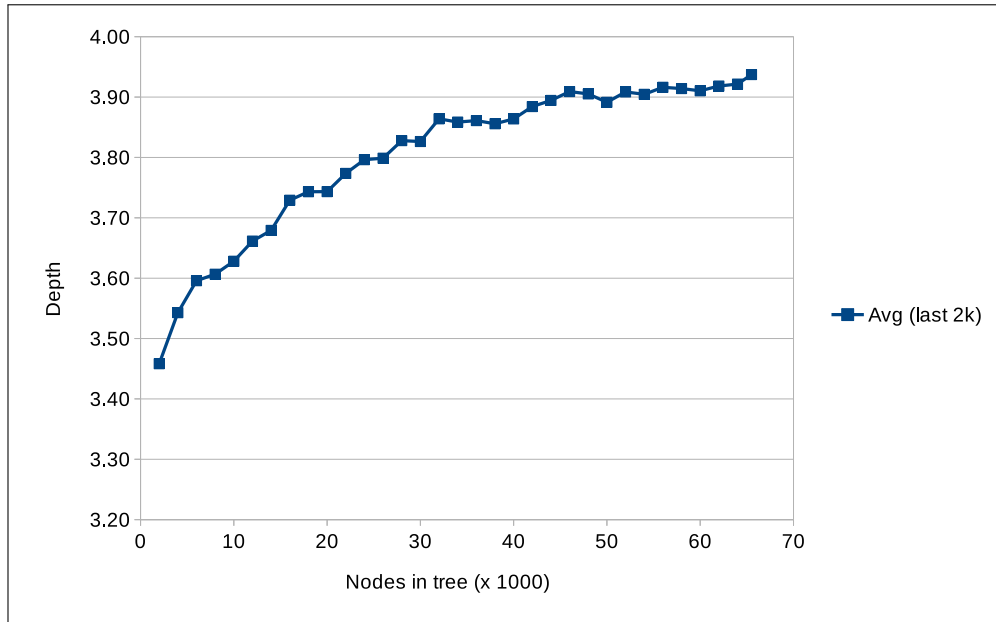
Third graph, Figure 29, shows how *depths* taken by *predecessor()* operations are distributed as tree becomes more dense.

Fourth graph, Figure 30, shows how the *predecessor()* average time, for each *depth*, evolves as the tree becomes more dense.

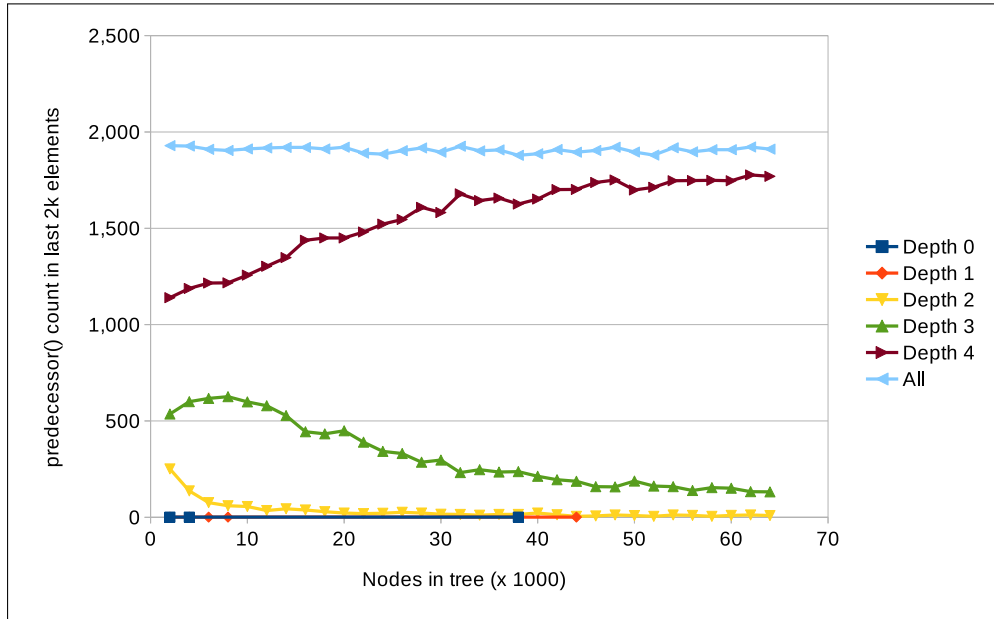
And fifth graph, Figure 31, shows the overall average time taken by *predecessor()* operations for each *depth*.

Figure 27 – *predecessor()* mean time by nodes in a dense vEB(2^{24}).

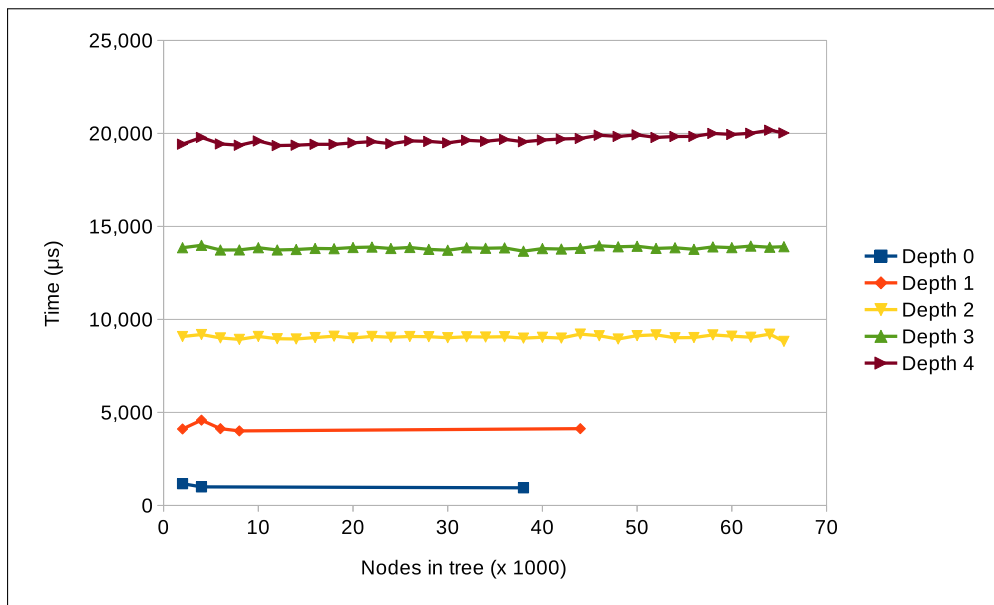
The *predecessor()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 28 – *predecessor()* average depth by nodes in a dense vEB(2^{24}).

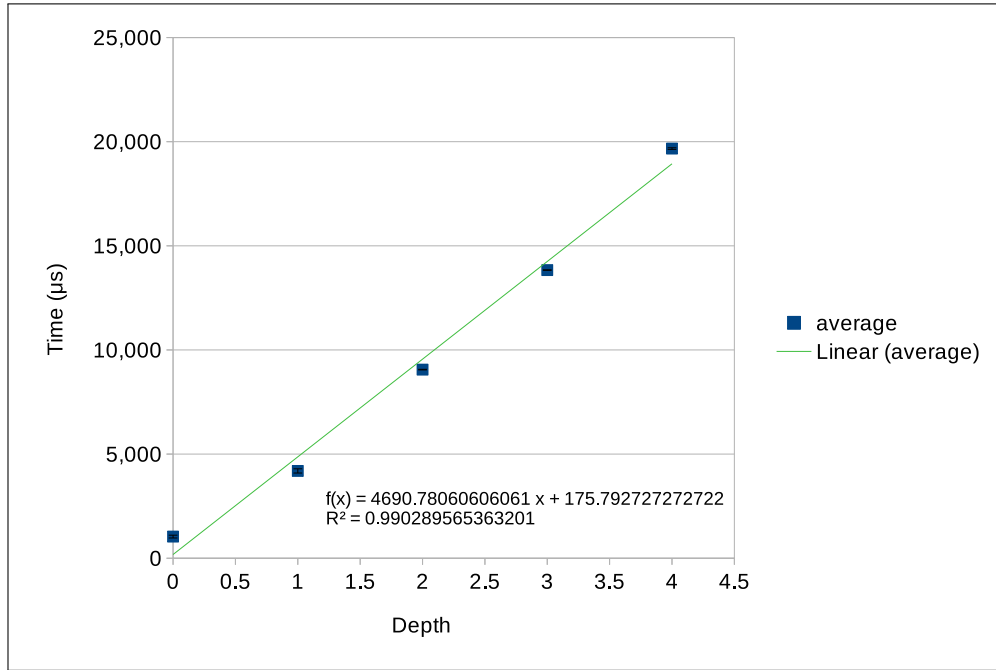
The *predecessor()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 29 – *predecessor()* depths count, of last 2k elements, by nodes in a dense $vEB(2^{24})$.

A counting of depths taken by *predecessor()* operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 30 – *predecessor()* depths time by nodes in a dense $vEB(2^{24})$.

The *predecessor()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 31 – *predecessor()* depth average time of a dense $vEB(2^{24})$.

The average time taken by *predecessor()* operations per depth.

Like in *successor()*, it digs down as the tree becomes populated because chances to hit the minimal values on nodes will decrease. It is easy to see the average *depth* increasing on Figures 28 and 29.

The Figure 30 also seems to have a small fluctuation. Just like for *successor()* it would be advisable repeat the experiment to confirm that.

5.2.1.4 Search

In this section we consolidate, in five graphs, the statistics collected for *search()* operations on a dense tree and analyze them.

First graph, Figure 32, shows how the *search()* average time evolves as the tree becomes more dense.

Second graph, Figure 33, shows how the average *depth* taken by *search()* operations evolves as tree becomes more dense.

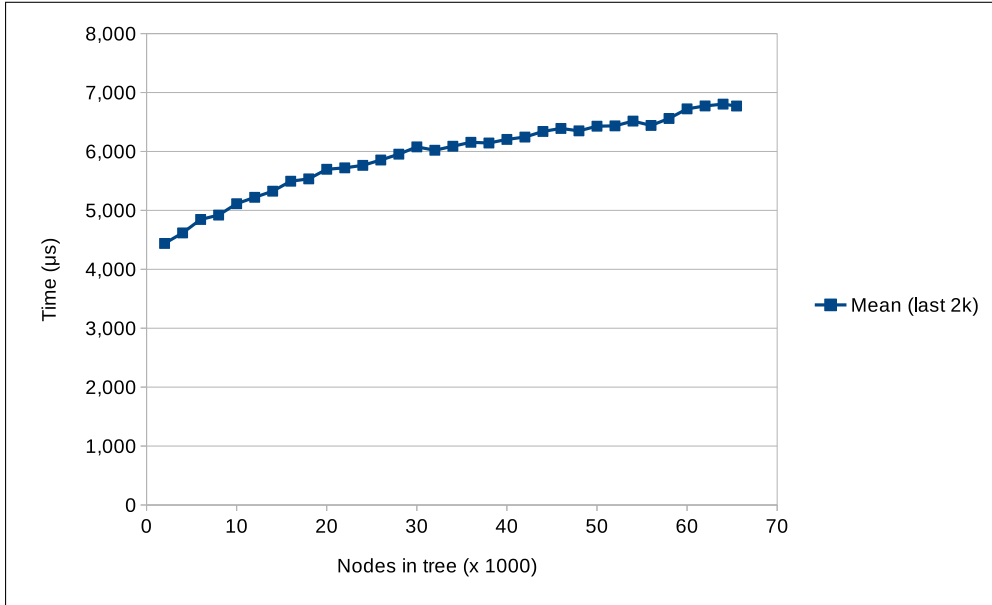
Third graph, Figure 34, shows how *depths* taken by *search()* operations are distributed as tree becomes more dense.

Fourth graph, Figure 35, shows how the *search()* average time, for each *depth*, evolves as the tree becomes more dense.

And fifth graph, Figure 36, shows the overall average time taken by *search()*

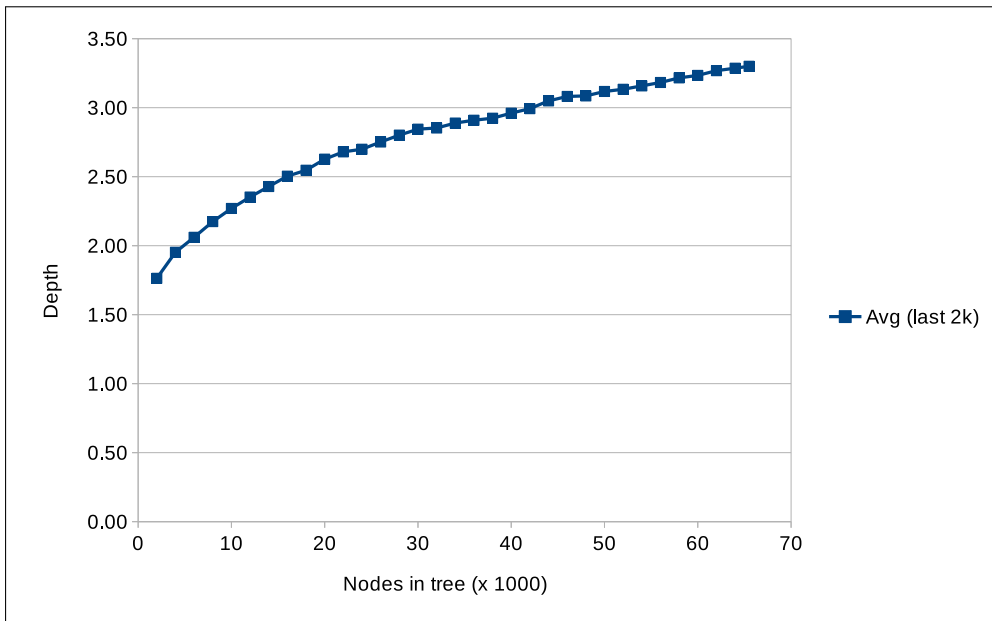
operations for each *depth*.

Figure 32 – *search()* mean time by nodes in a dense vEB(2^{24}).

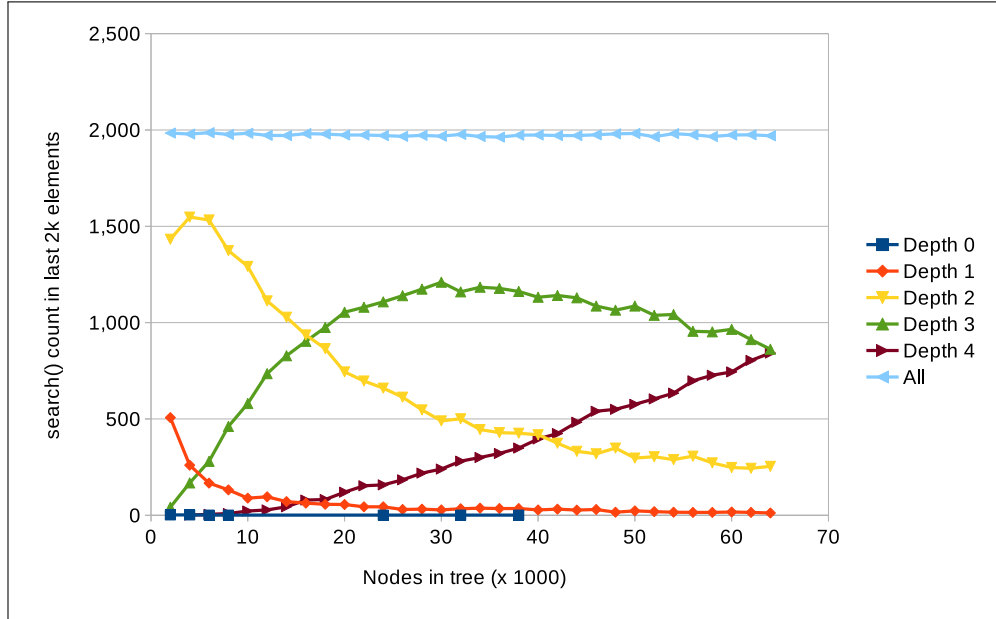


The *search()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

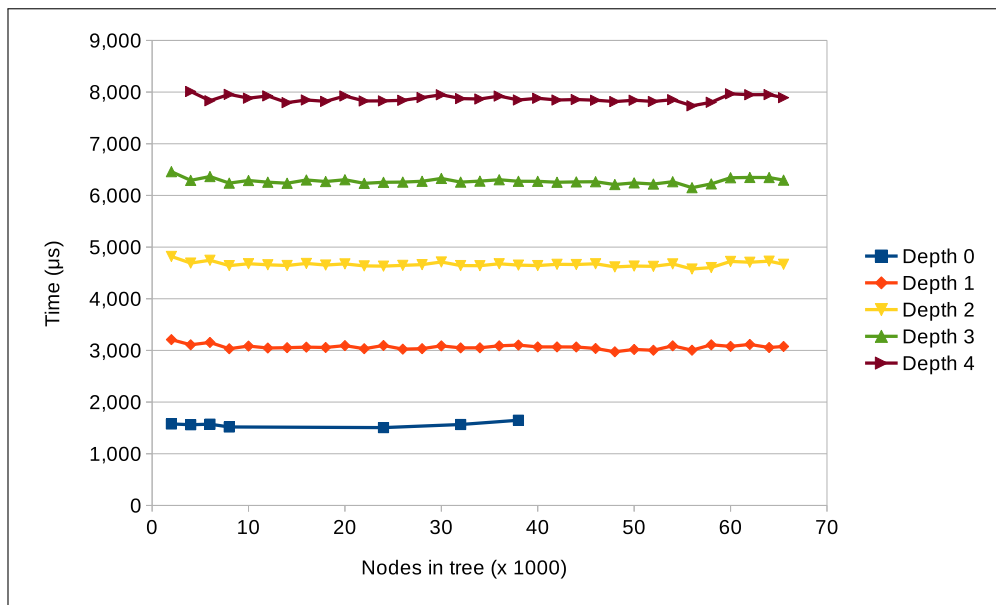
Figure 33 – *search()* average depth by nodes in a dense vEB(2^{24}).



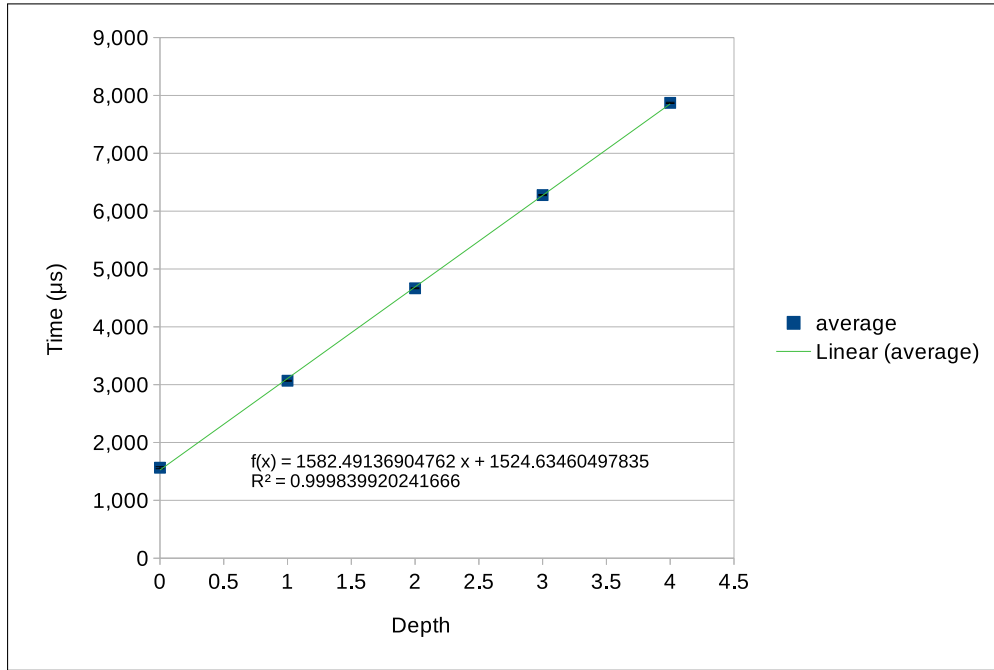
The *search()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 34 – $search()$ depths count, of last 2k elements, by nodes in a dense $vEB(2^{24})$.

A counting of depths taken by $search()$ operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 35 – $search()$ depths time by nodes in a dense $vEB(2^{24})$.

The $search()$ mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 36 – *search()* depth average time of a dense vEB(2^{24}).

The average time taken by *search()* operations per depth.

The average *depth* of *search()* operations increase (Figures 33 and 34) because the algorithm only look into the cluster and it naturally gets deeper as the tree becomes filled. It is a dense tree and it will end up being a tree like the one in Table 9 where half keys will be found at the bottom of the tree.

5.2.1.5 Remove

In this section we consolidate, in five graphs, the statistics collected for *remove()* operations on a dense tree and analyze them.

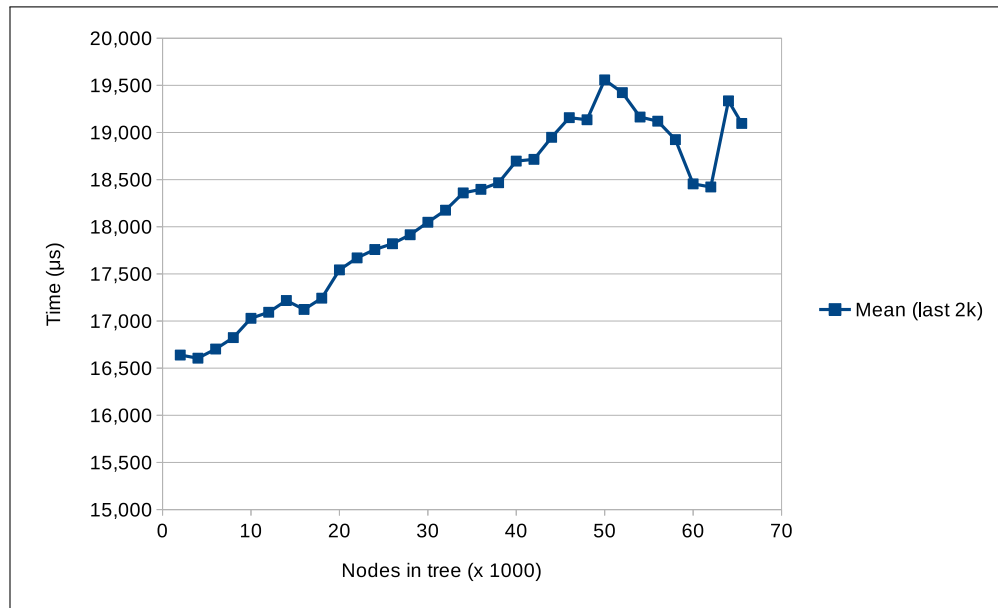
First graph, Figure 37, shows how the *remove()* average time evolves as the tree becomes more dense.

Second graph, Figure 38, shows how the average *depth* taken by *remove()* operations evolves as tree becomes more dense.

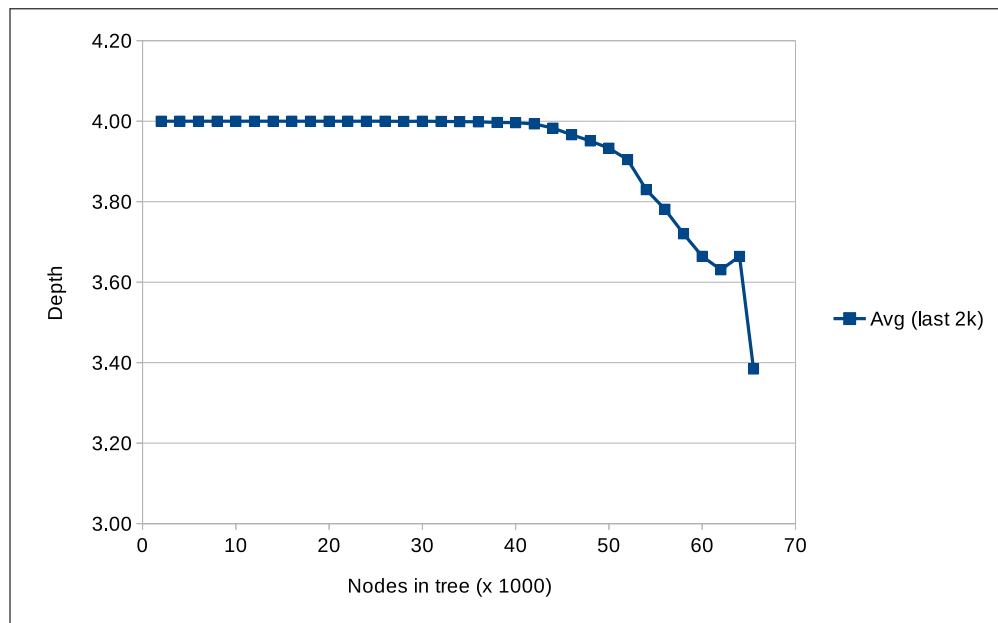
Third graph, Figure 39, shows how *depths* taken by *remove()* operations are distributed as tree becomes more dense.

Fourth graph, Figure 40, shows how the *remove()* average time, for each *depth*, evolves as the tree becomes more dense.

And fifth graph, Figure 41, shows the overall average time taken by *remove()* operations for each *depth*.

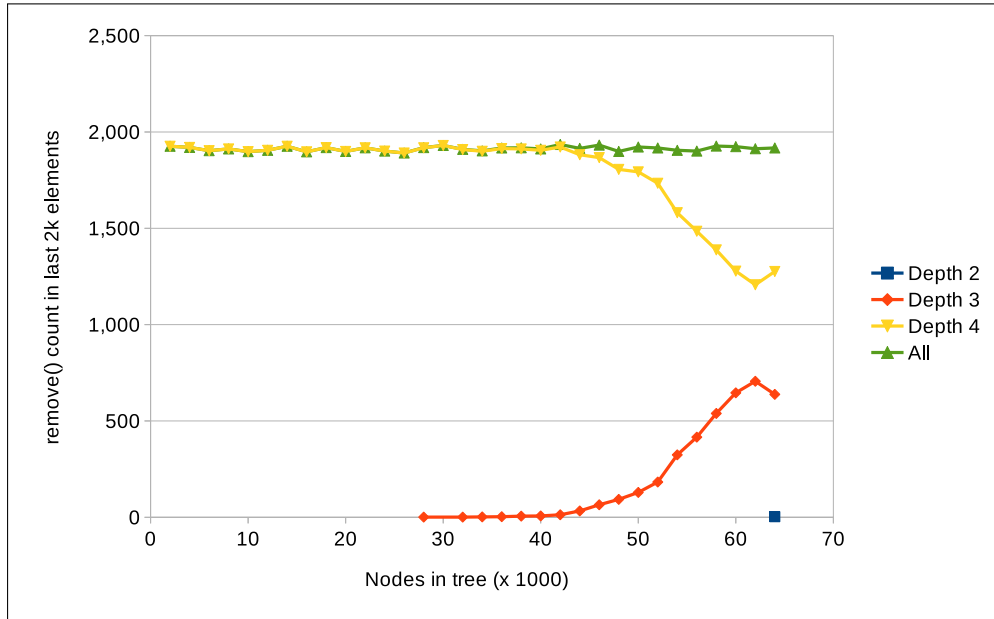
Figure 37 – *remove()* mean time by nodes in a dense vEB(2^{24}).

The *remove()* mean time of a population of 2000 elements. The tree is less populated on the right of the chart and more populated on the left.

Figure 38 – *remove()* average depth by nodes in a dense vEB(2^{24}).

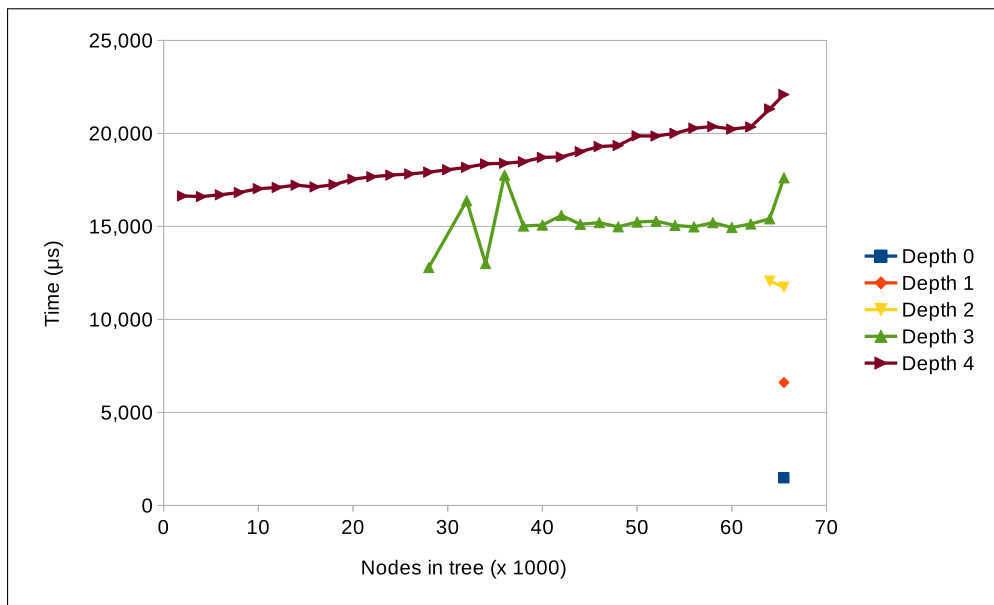
The *remove()* average depth of a population of 2000 elements. The tree is less populated on the right of the chart and more populated on the left.

Figure 39 – *remove()* depths count, of last 2k elements, by nodes in a dense $vEB(2^{24})$.

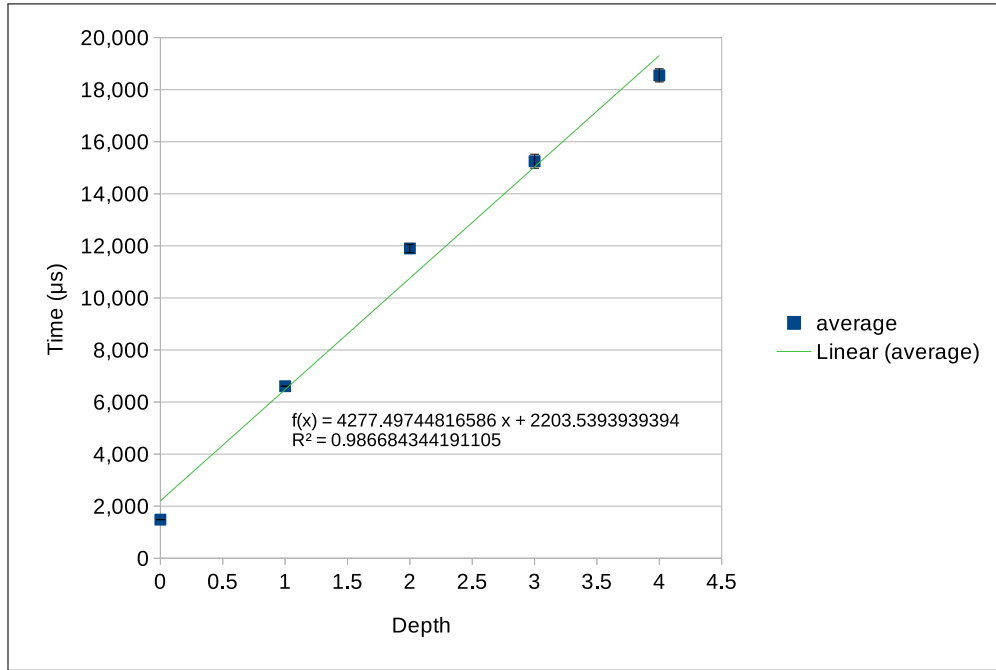


A counting of depths taken by *remove()* operations of a population of 2000 elements. The tree is right populated on the left of the chart and more populated on the left.

Figure 40 – *remove()* depths time by nodes in a dense $vEB(2^{24})$.



The *remove()* mean time, by depth, of a population of 2000 elements. The tree is less populated on the right of the chart and more populated on the left.

Figure 41 – *remove()* depth average time of a dense vEB(2^{24}).The average time taken by *remove()* operations per depth.

The way we look the *remove()* operations is reversed regarding the other operations, the tree is full on the left and empty on the right.

The first interesting thing to notice here is the average *depth* of *remove()* operations, Figure 38 is almost a perfect mirror of insert Figure 18. The same effect also happens for each *depth* (Figures 39 and 19). It is also similar considering the time taken for each *depth* (Figures 40 and 20).

Like in *insert()* algorithm, it is a bit surprising the average *remove()* time in Figure 37 does not follow the trend the average *depth* in Figure 38. If we look *remove()* Algorithm 5, when the tree is dense there is only one RPC by *level* at line 19. When the tree becomes sparse, the RPC at line 19 will only reach the next level, *i.e.* $O(1)$, but it will make an RPC at line 21, and chances becomes greater to pass condition at line 22 and make another RPC at line 23. So, at each level, it can make one RPC, two RPCs or three RPCs. The one RPC case will happen more often when the tree is very populated and start moving towards 3 RPCs when tree gets more sparse.

To summarize, both *remove()* and *insert()* operations will become faster as the tree gets more dense.

5.2.2 Experiment 02 - Sparse tree

5.2.2.1 Insert

.

In this section we consolidate, in five graphs, the statistics collected for *insert()* operations on a sparse tree and analyze them.

First graph, Figure 42, shows how the *insert()* average time evolves as the tree has few more elements.

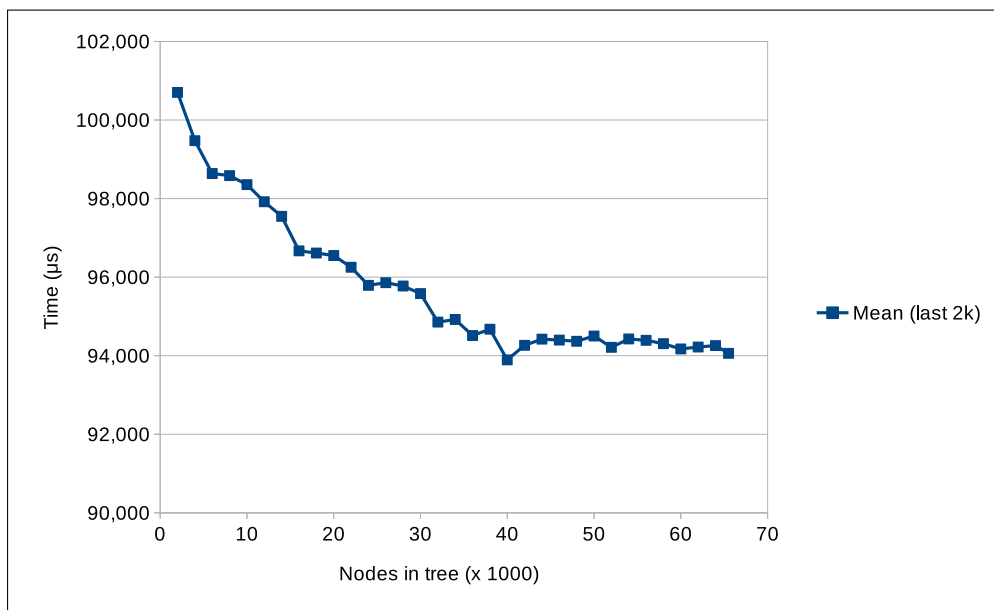
Second graph, Figure 43, shows how the average *depth* taken by *insert()* operations evolves as the tree has few more elements.

Third graph, Figure 44, shows how *depths* taken by *insert()* operations are distributed as the tree has few more elements.

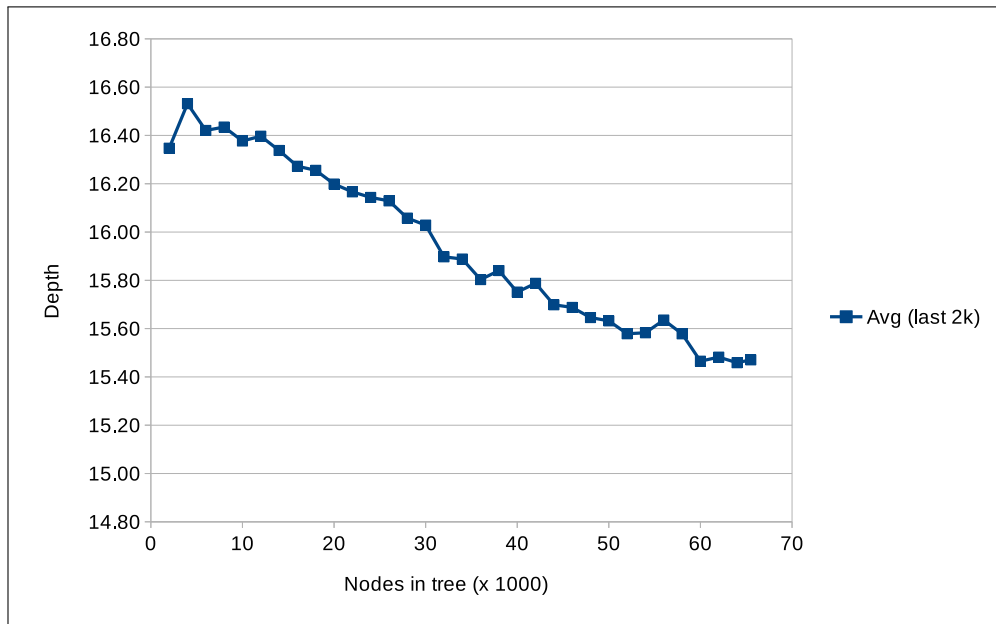
Fourth graph, Figure 45, shows how the *insert()* average time, for each *depth*, evolves as the tree has few more elements.

And fifth graph, Figure 46, shows the overall average time taken by *insert()* operations for each *depth*.

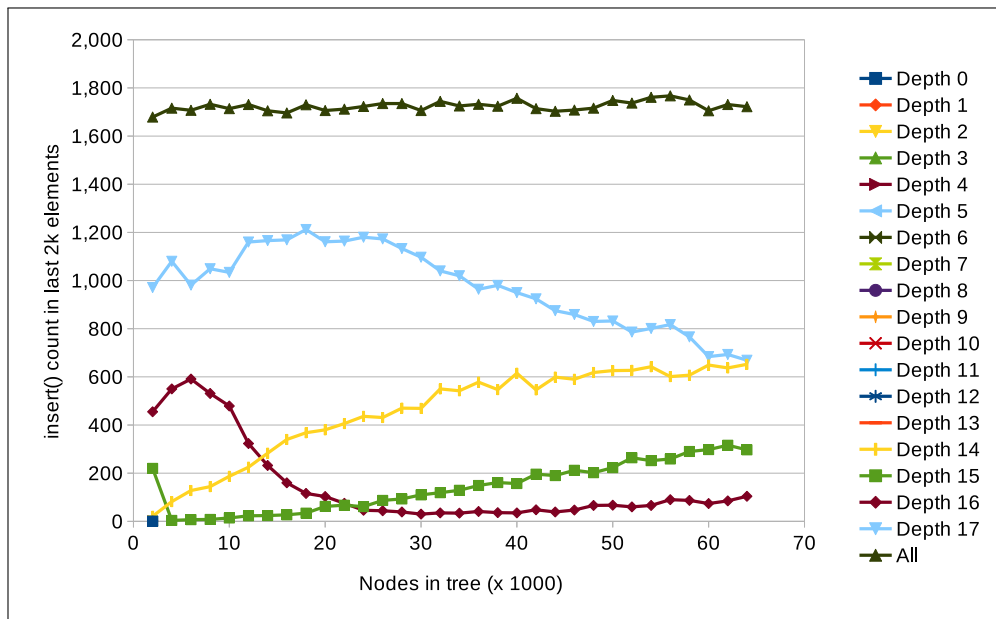
Figure 42 – *insert()* mean time by nodes in a sparse vEB($2^{2^{17}}$).



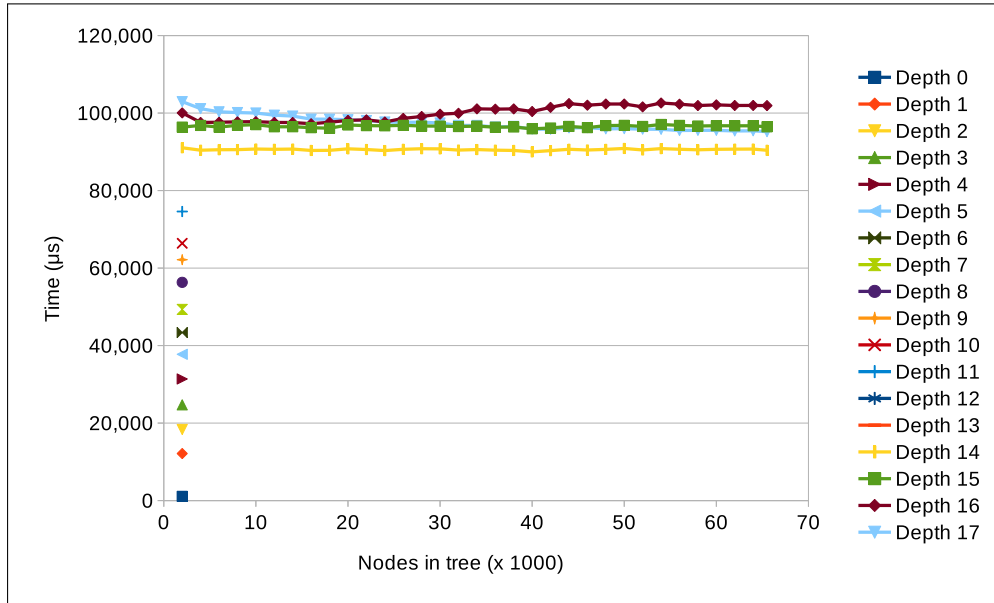
The *insert()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 43 – *insert()* average depth by nodes in a sparse vEB($2^{2^{17}}$).

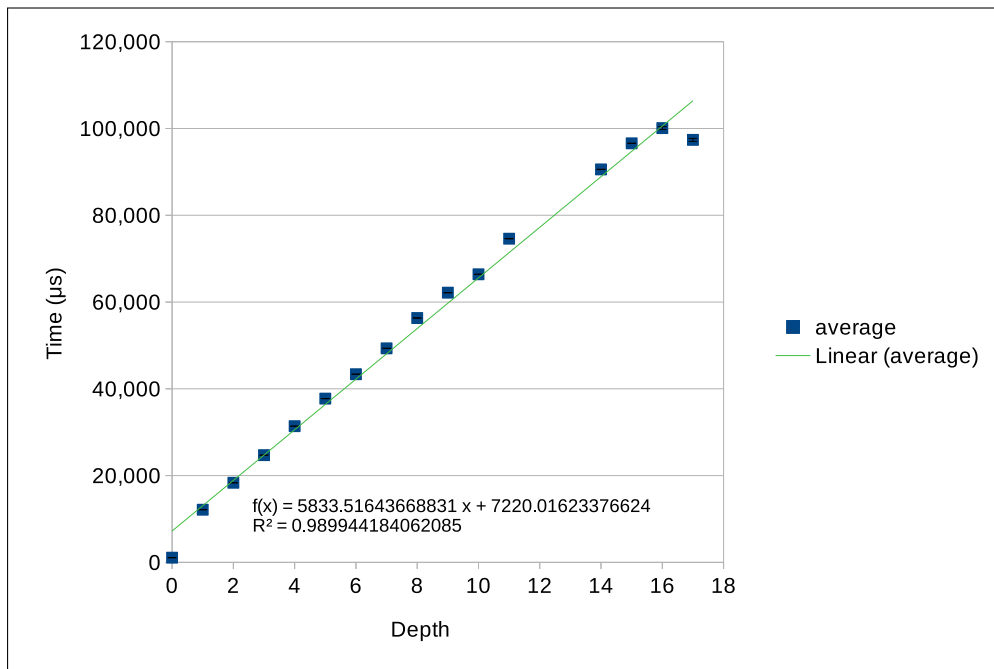
The *insert()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 44 – *insert()* depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).

A counting of depths taken by *insert()* operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 45 – *insert()* depths time by nodes in a sparse $vEB(2^{2^{17}})$.

The *insert()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 46 – *insert()* depth average time of a sparse $vEB(2^{2^{17}})$.

The average time taken by *insert()* operations per depth.

The average *insert()* time decreases on sparse trees (Figure 42) as more elements are inserted, like in Experiment 01 with on dense trees (Figure 17). But it has different

characteristics. The average *depth* of *insert()* operations increases on dense trees (Figures 18 and 19) while in opposite direction decreases on sparse trees (Figure 43). This together with fact that insertion on new node costs much higher than inserting on a existing node (Please see discussion in Subsection 5.2.1.1) make it easy to figure out why the average time decreases as we insert more elements.

It is hard to visualize why the average *depth* for *insert()* operations decrease (Figures 43 and 44) on sparse trees as we insert more elements. The reason for that is because, at *depth* 13 (please refer to Table 8) we will start inserting keys, actually its "high" with 2^4 bits, that are already present on the cluster, and therefore won't need to go down at the very left summary. This will start populating empty *depth* 14 trees inserting the "low", with 2^4 bits on it. Notice *depth* 13 is a $vEB(2^{2^8})$ and looks like is not uniform on its higher 2^4 bits.

Notice, that creating new paths down to empty *depth* 14, 15 or 16 trees may increase the insertion time, but decreasing the average *depth* helps decrease the insertion time. It could explain why the average time (Figure 42) at some point (specifically after 60 K elements) starts getting more stable.

It is hard to predict, but we believe, after this point, while the tree is still sparse, the tree will start behaving like a dense tree at children of the very left $vEB(2^{2^8})$ summary.

In addition, the time for the deepest *level* decreases since beginning (Figure 45), this is probably the same effect we got on Experiment 01, when the summary at very left becomes dense.

While we can pretty much predict the average time will decrease when the tree becomes dense, because there will be less nodes to create, it is hard to predict how it will float in between. Probably we need more experiments to predict that, but that is not an easy task because such big tree is too huge to became dense. This experiment took around one hour to insert 65536 elements, even if we had enough memory, it would take 3.7 centuries to make a $vEBt(2^{2^{17}})$ full.

5.2.2.2 Successor

.

In this section we consolidate, in five graphs, the statistics collected for *successor()* operations on a sparse tree and analyze them.

First graph, Figure 47, shows how the *successor()* average time evolves as the tree has few more elements.

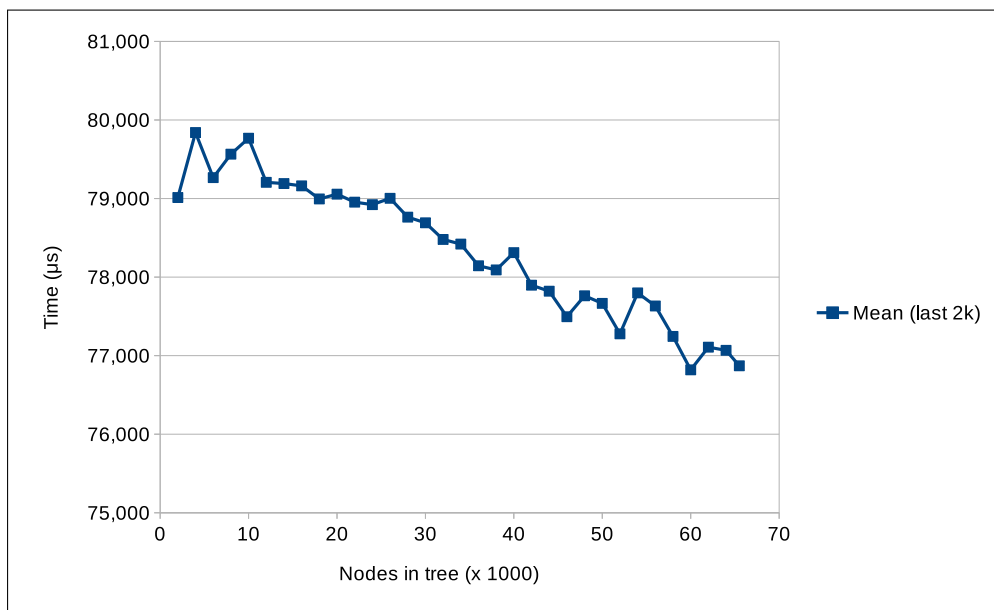
Second graph, Figure 48, shows how the average *depth* taken by *successor()* operations evolves as the tree has few more elements.

Third graph, Figure 49, shows how *depths* taken by *successor()* operations are distributed as the tree has few more elements.

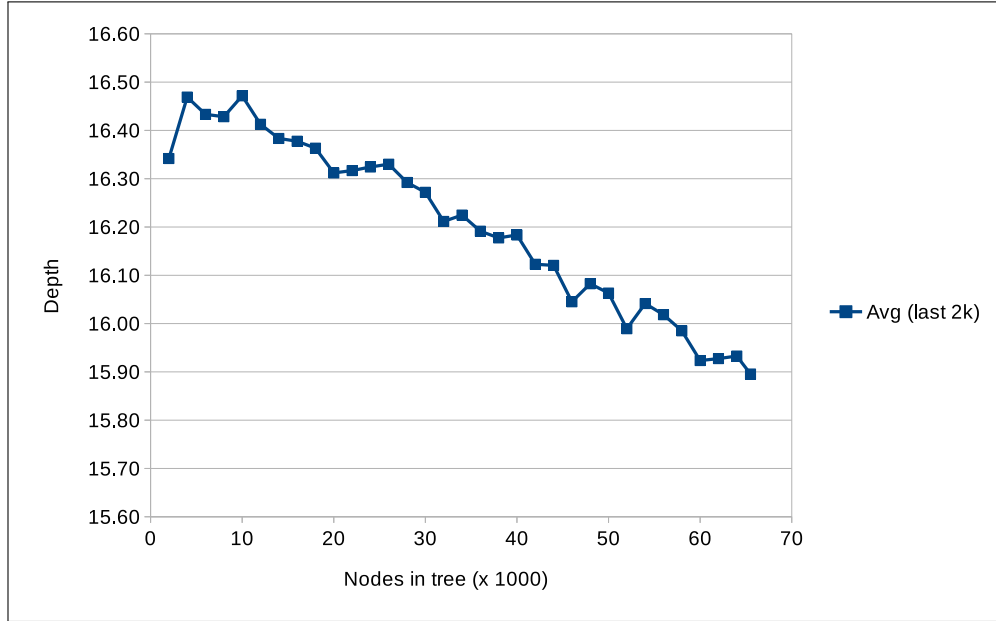
Fourth graph, Figure 50, shows how the *successor()* average time, for each *depth*, evolves as the tree has few more elements.

And fifth graph, Figure 51, shows the overall average time taken by *successor()* operations for each *depth*.

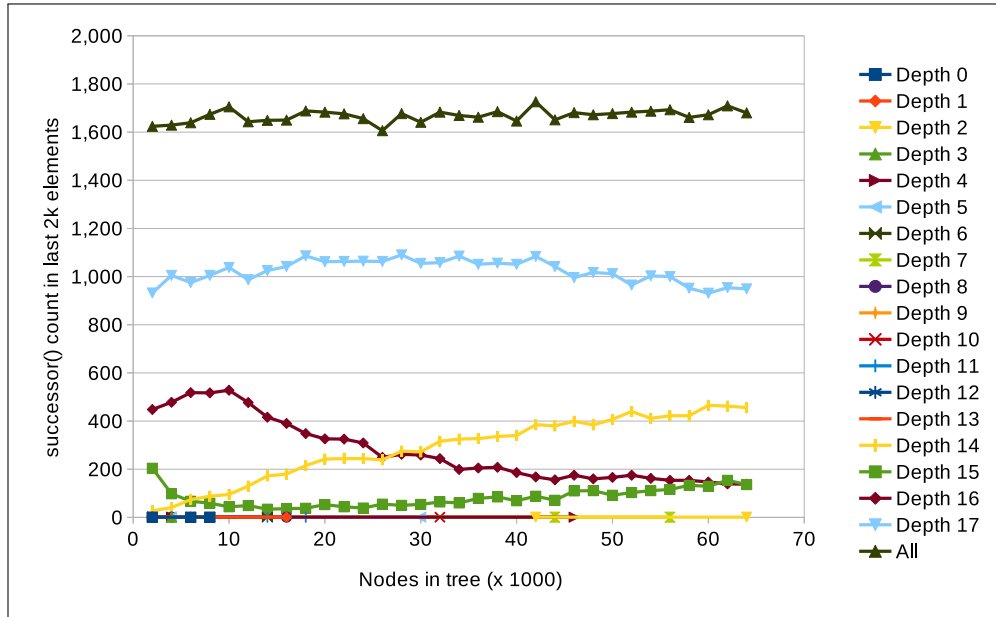
Figure 47 – *successor()* mean time by nodes in a sparse vEB($2^{2^{17}}$).



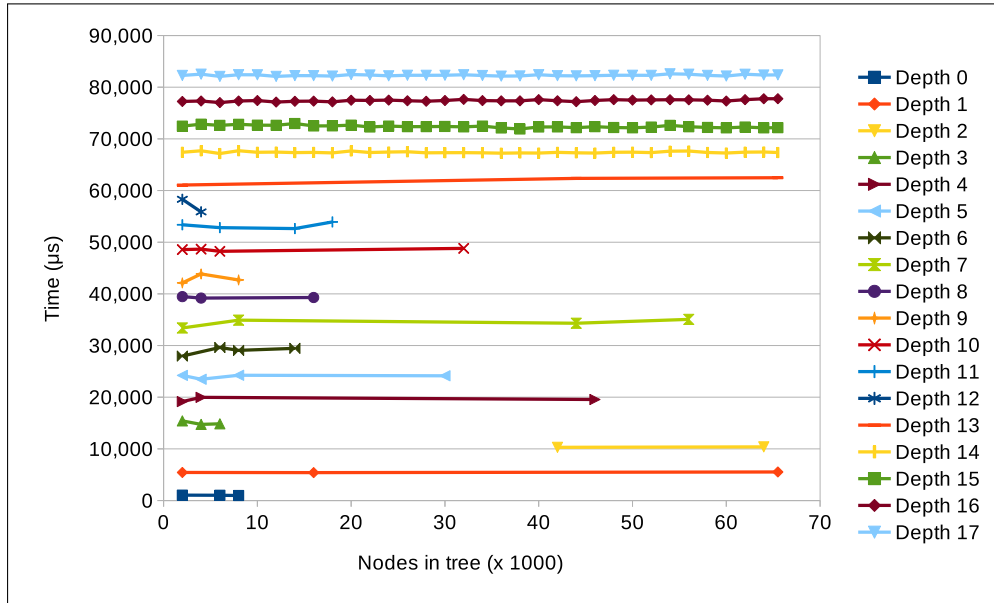
The *successor()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 48 – *successor()* average depth by nodes in a sparse vEB($2^{2^{17}}$).

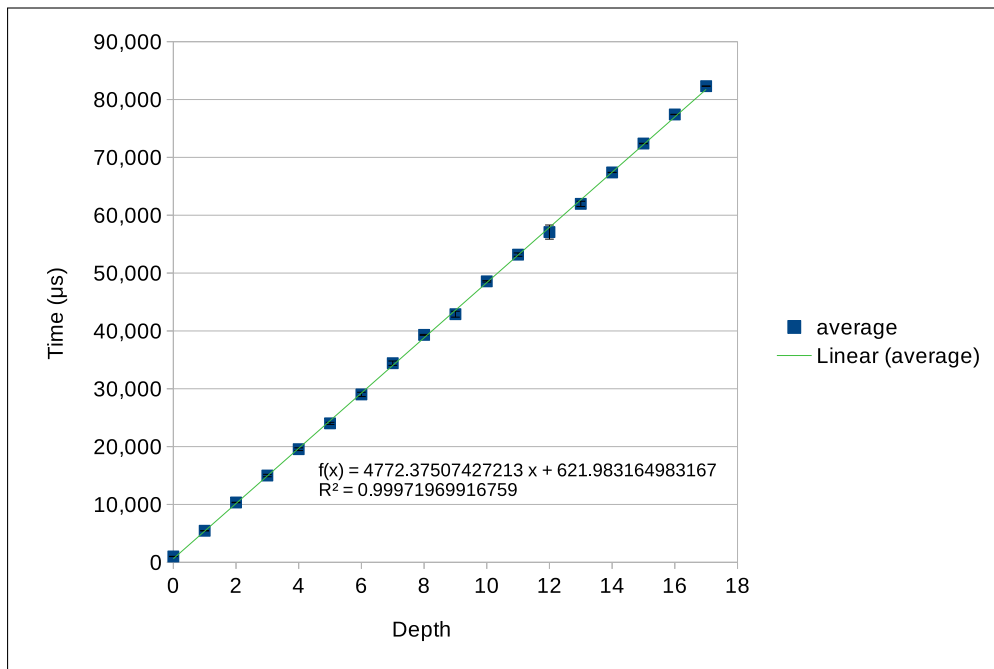
The *successor()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 49 – *successor()* depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).

A counting of depths taken by *successor()* operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 50 – *successor()* depths time by nodes in a sparse vEB($2^{2^{17}}$).

The *successor()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 51 – *successor()* depth average time of a sparse vEB($2^{2^{17}}$).

The average time taken by *successor()* operations per depth.

It is a bit surprising the average time of *successor()* operations decrease (Figure 47) on sparse trees while increases on dense trees (Figure 5.2.1.2), although it makes sense

when we see the average *depth* decreasing for *successor()* operations (Figures 48 and 49). The question is, why the average *depth* decreases? That's because it will go down at the very left of the summary to find the successor. In Algorithm 2, the condition in line 12 will fail because the tree is sparse, and it will enter down the summary at line 16.

5.2.2.3 Predecessor

In this section we consolidate, in five graphs, the statistics collected for *predecessor()* operations on a sparse tree and analyze them.

First graph, Figure 52, shows how the *predecessor()* average time evolves as the tree has few more elements.

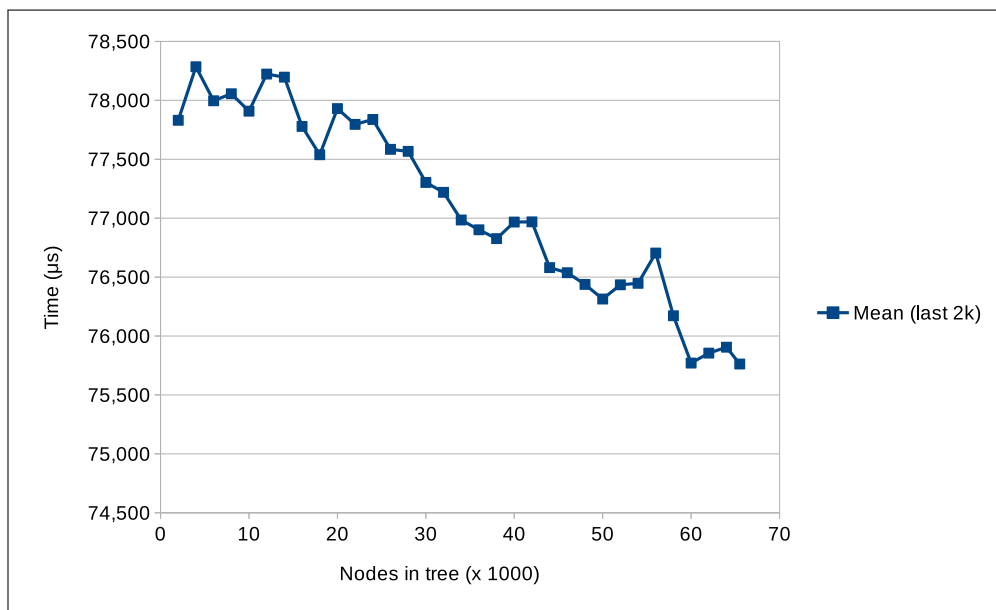
Second graph, Figure 53, shows how the average *depth* taken by *predecessor()* operations evolves as the tree has few more elements.

Third graph, Figure 54, shows how *depths* taken by *predecessor()* operations are distributed as the tree has few more elements.

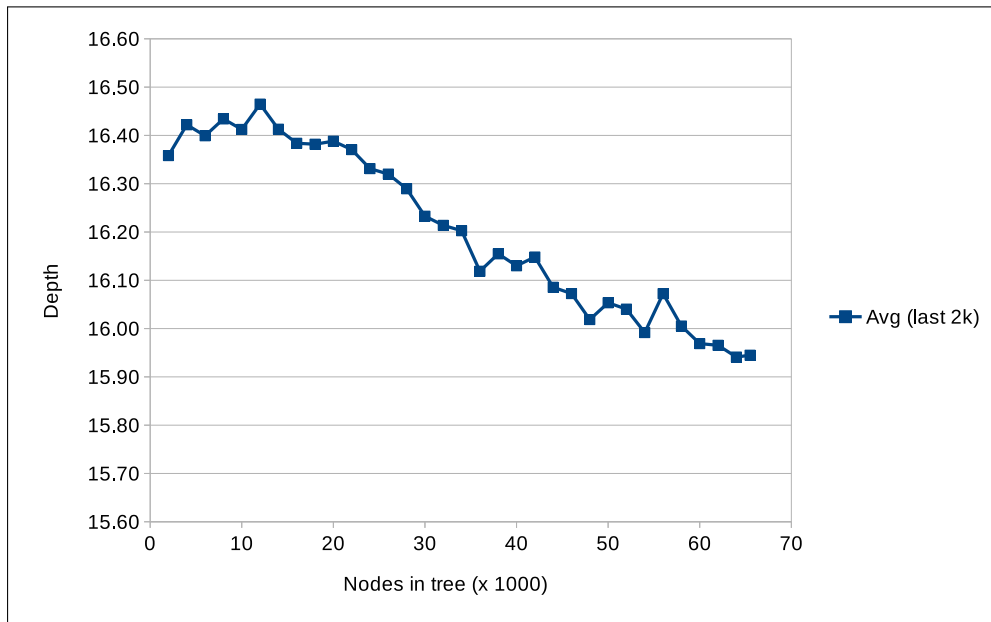
Fourth graph, Figure 55, shows how the *predecessor()* average time, for each *depth*, evolves as the tree has few more elements.

And fifth graph, Figure 56, shows the overall average time taken by *predecessor()* operations for each *depth*.

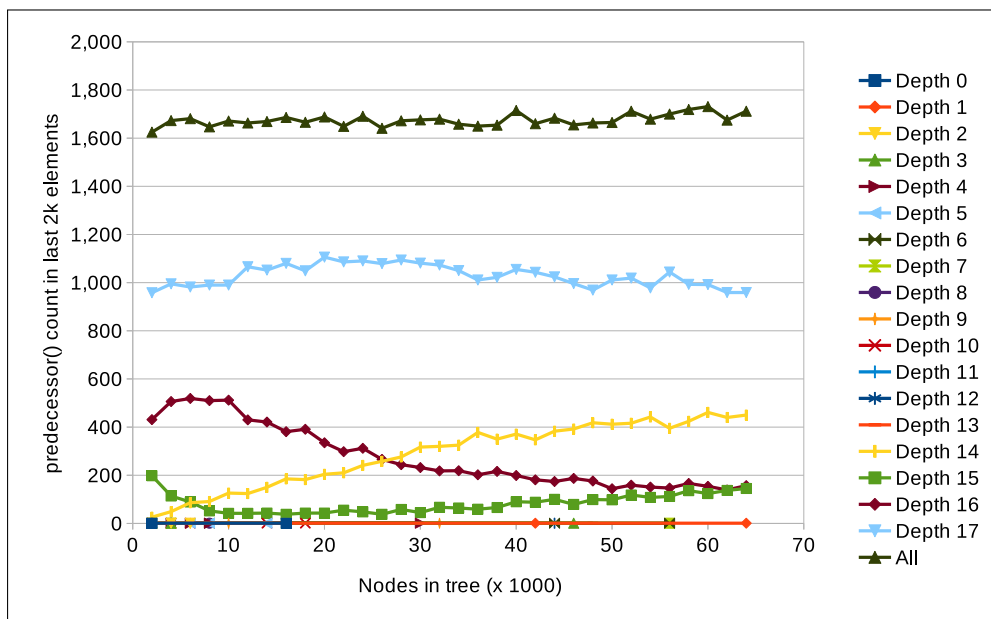
Figure 52 – *predecessor()* mean time by nodes in a sparse vEB($2^{2^{17}}$).



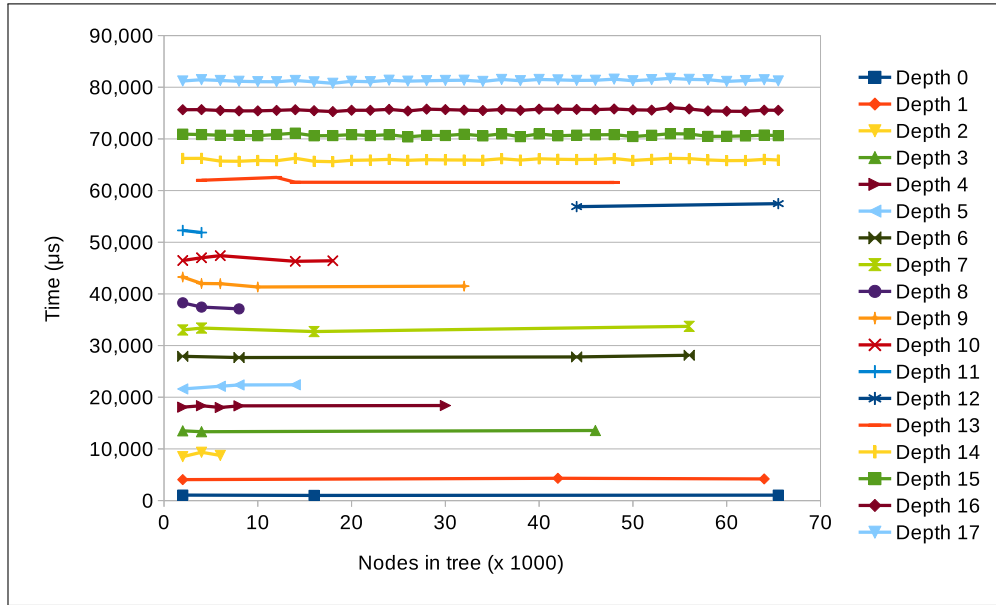
The *predecessor()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 53 – *predecessor()* average depth by nodes in a sparse vEB($2^{2^{17}}$).

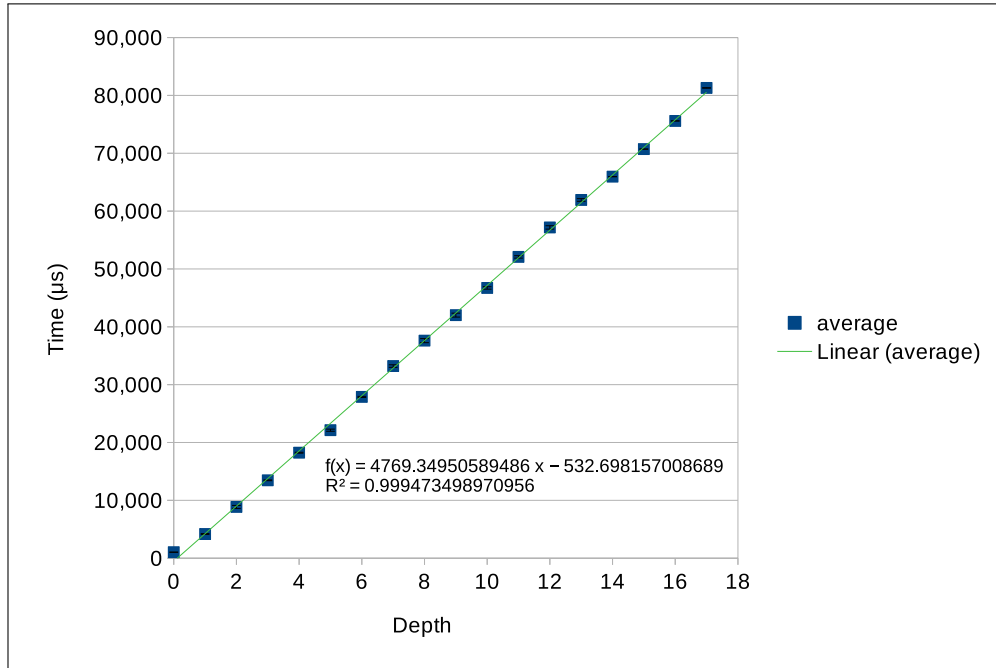
The *predecessor()* average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 54 – *predecessor()* depths count, of last 2k elements, by nodes in a sparse vEB($2^{2^{17}}$).

A counting of depths taken by *predecessor()* operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 55 – *predecessor()* depths time by nodes in a sparse $vEB(2^{2^{17}})$.

The *predecessor()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 56 – *predecessor()* depth average time of a sparse $vEB(2^{2^{17}})$.

The average time taken by *predecessor()* operations per depth.

For predecessor applies exactly the same analysis as successor.

The average time of *predecessor()* operations decrease (Figure 52) on sparse trees while increases on dense trees (Figure 27), although it makes sense when we see the average *depth* decreasing for *predecessor()* operations (Figures 53 and 54). The question is, why the average *depth* decreases? That's because it will go down at the very left of the summary to find the predecessor. In Algorithm 3, the condition in line 12 will fail because the tree is sparse, and it will enter down the summary at line 16.

5.2.2.4 Search

In this section we consolidate, in five graphs, the statistics collected for *search()* operations on a sparse tree and analyze them.

First graph, Figure 57, shows how the *search()* average time evolves as the tree has few more elements.

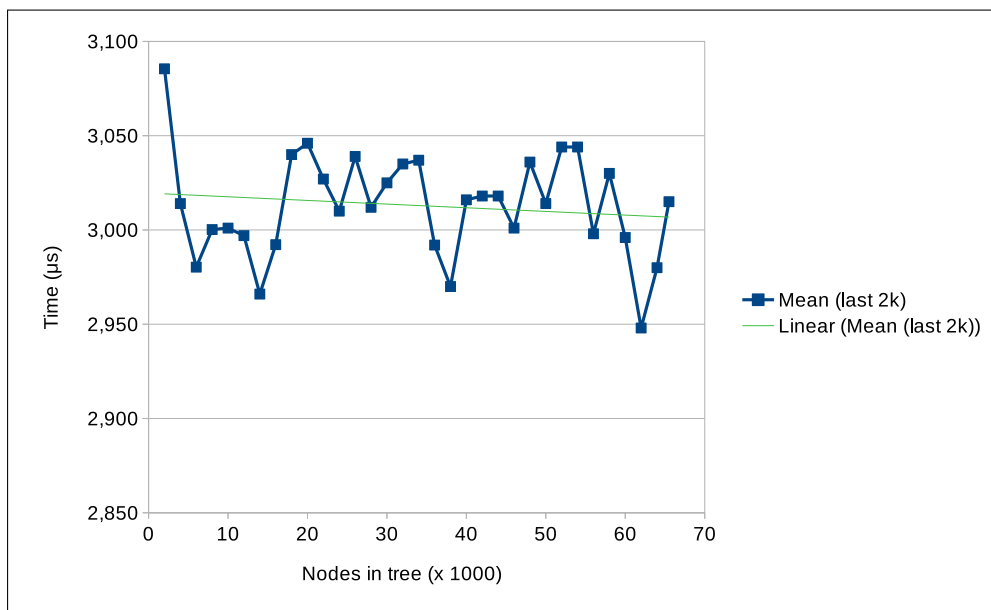
Second graph, Figure 58, shows how the average *depth* taken by *search()* operations evolves as the tree has few more elements.

Third graph, Figure 59, shows how *depths* taken by *search()* operations are distributed as the tree has few more elements.

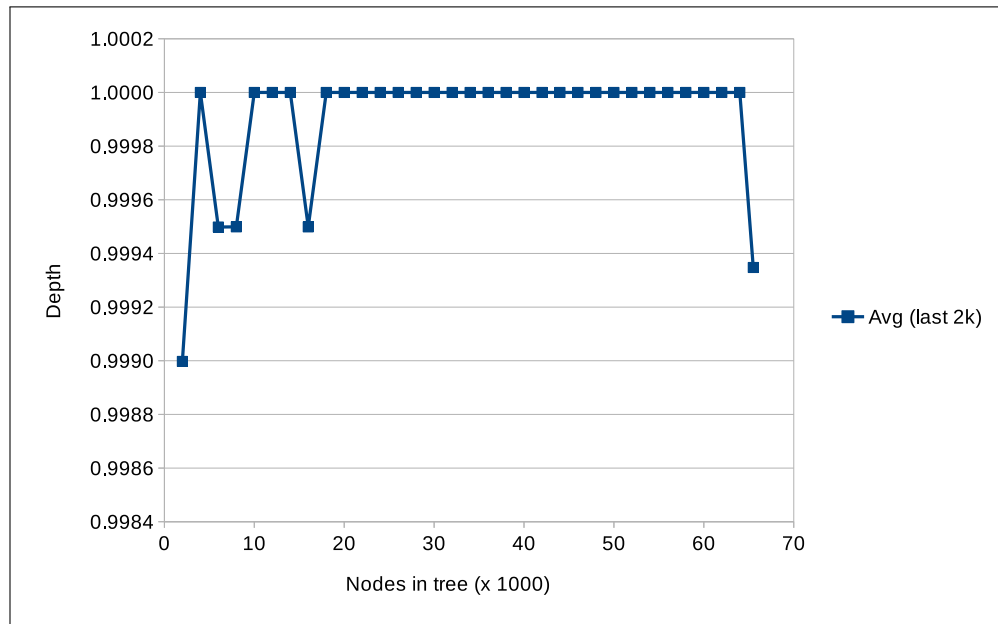
Fourth graph, Figure 60, shows how the *search()* average time, for each *depth*, evolves as the tree has few more elements.

And fifth graph, Figure 61, shows the overall average time taken by *search()* operations for each *depth*.

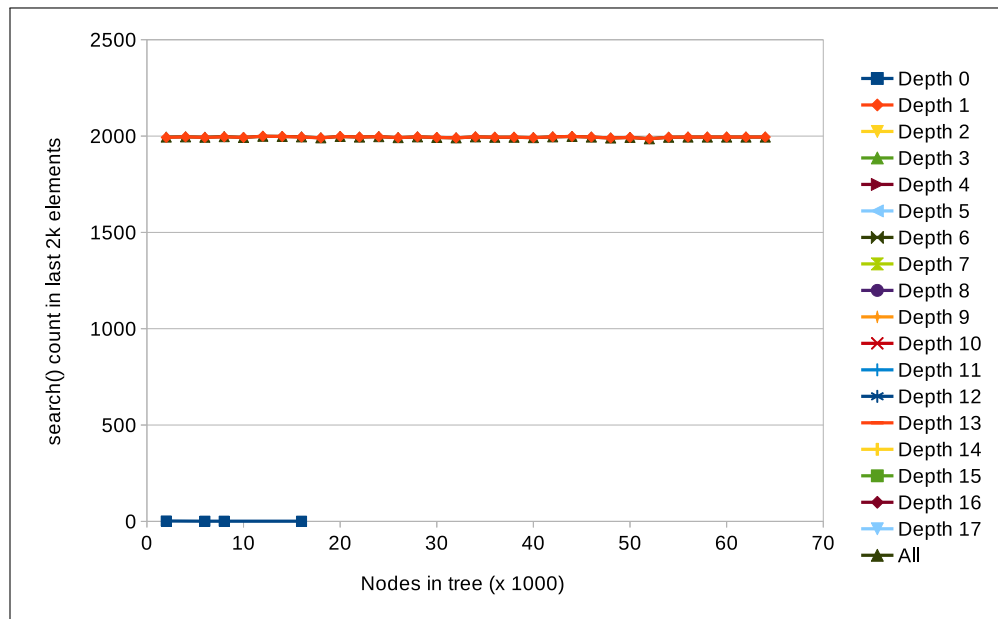
Figure 57 – *search()* mean time by nodes in a sparse vEB($2^{2^{17}}$).



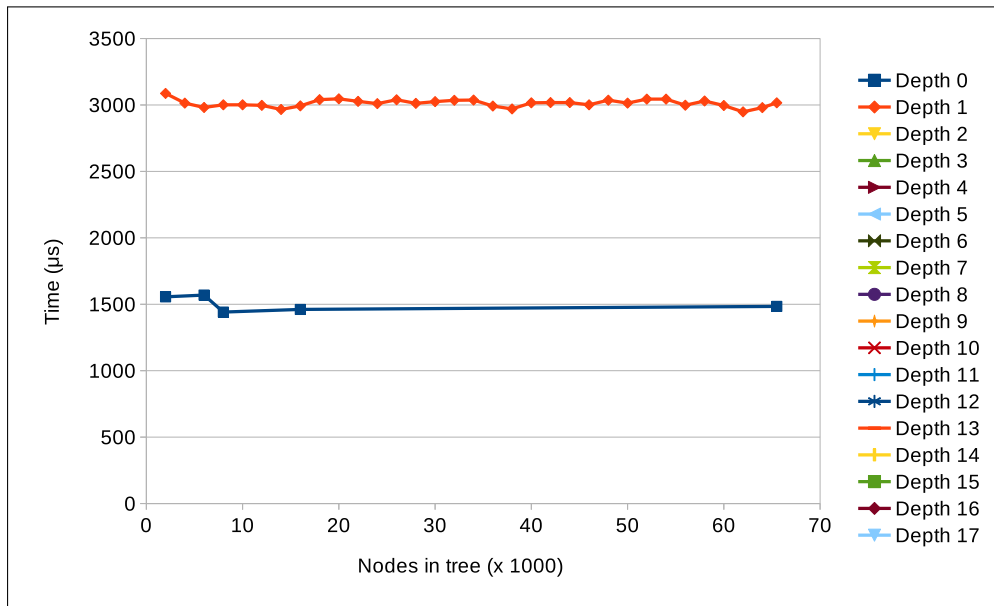
The *search()* mean time of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 58 – $search()$ average depth by nodes in a sparse $vEB(2^{2^{17}})$.

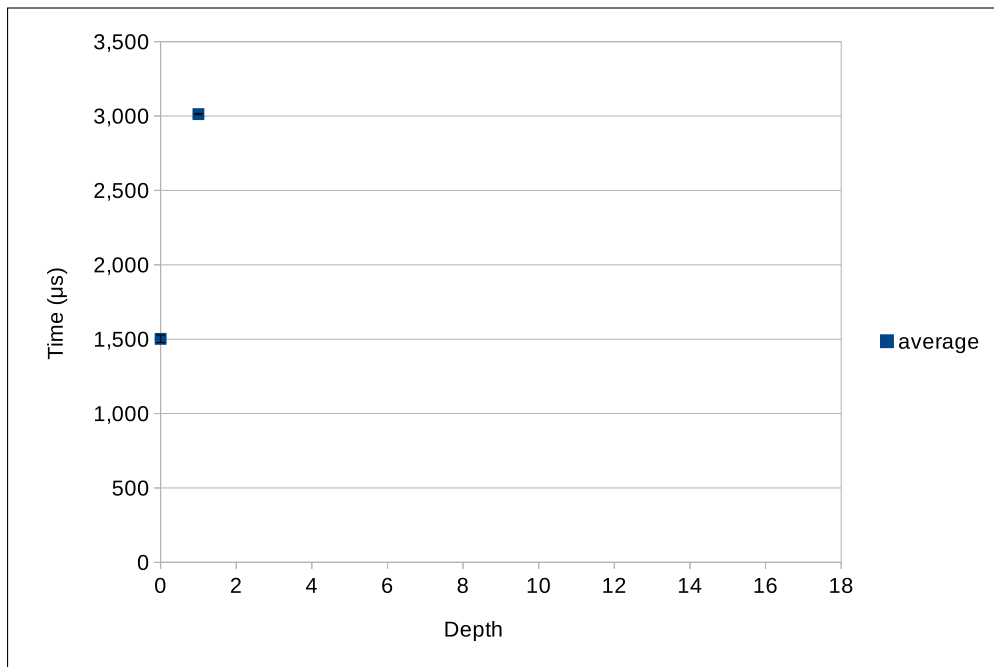
The $search()$ average depth of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 59 – $search()$ depths count, of last 2k elements, by nodes in a sparse $vEB(2^{2^{17}})$.

A counting of depths taken by $search()$ operations of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 60 – *search()* depths time by nodes in a sparse vEB($2^{2^{17}}$).

The *search()* mean time, by depth, of a population of the last 2000 inserted elements. The tree is less populated on the left of the chart and more populated on the right.

Figure 61 – *search()* depth average time of a sparse vEB($2^{2^{17}}$).

The average time taken by *search()* operations per depth.

All the insertions on a sparse tree stops at cluster *depth* 1 and then goes down into the very left summary. Thus, all searches will go at most to *depth* 1 because search

Algorithm 1 don't look summaries. This is pretty clear on the search graphs except with only one surprise. We really expected to see an inclination of zero on average *search()* time Figure 57 and we got a slightly decreasing trend. Although it is better than what we expected, it is not fair to accept it. Fortunately, we repeated this experiment and got a slightly increasing trend line, and therefore, we conclude it is just a floating. By the way, it is floating in only 100 micro-seconds range, which is definitely in the range error of our measurement.

5.2.2.5 Remove

In this section we consolidate, in five graphs, the statistics collected for *remove()* operations on a sparse tree and analyze them.

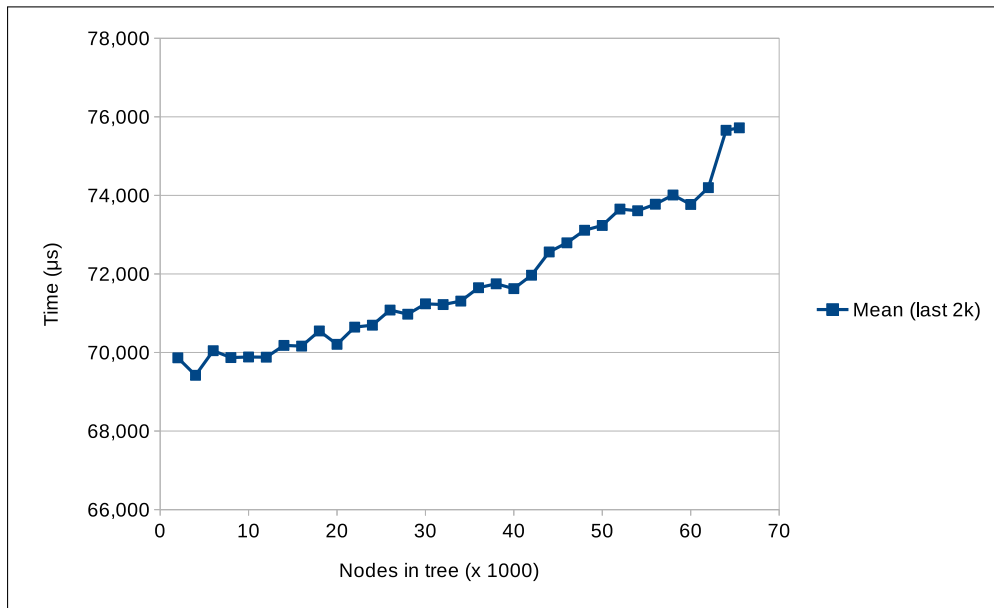
First graph, Figure 62, shows how the *remove()* average time evolves as the tree has few more elements.

Second graph, Figure 63, shows how the average *depth* taken by *remove()* operations evolves as the tree has few more elements.

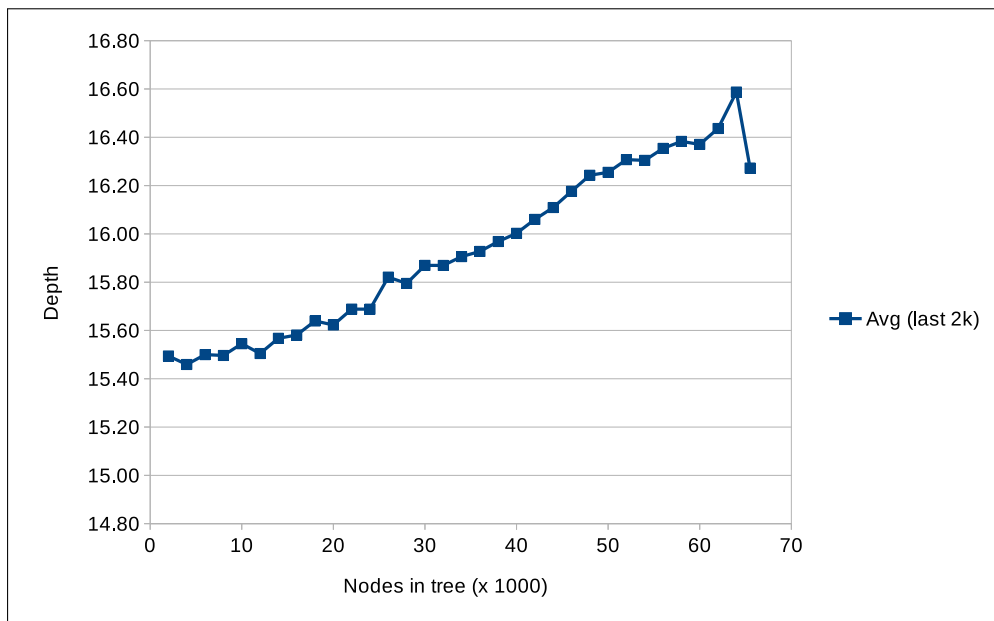
Third graph, Figure 64, shows how *depths* taken by *remove()* operations are distributed as the tree has few more elements.

Fourth graph, Figure 65, shows how the *remove()* average time, for each *depth*, evolves as the tree has few more elements.

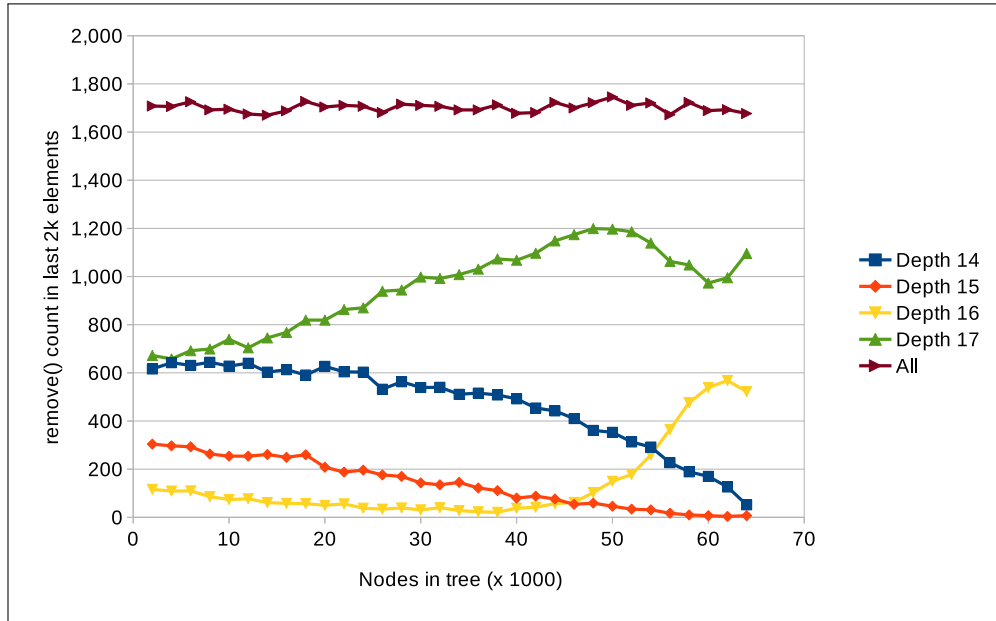
And fifth graph, Figure 66, shows the overall average time taken by *remove()* operations for each *depth*.

Figure 62 – *remove()* mean time by nodes in a sparse vEB($2^{2^{17}}$).

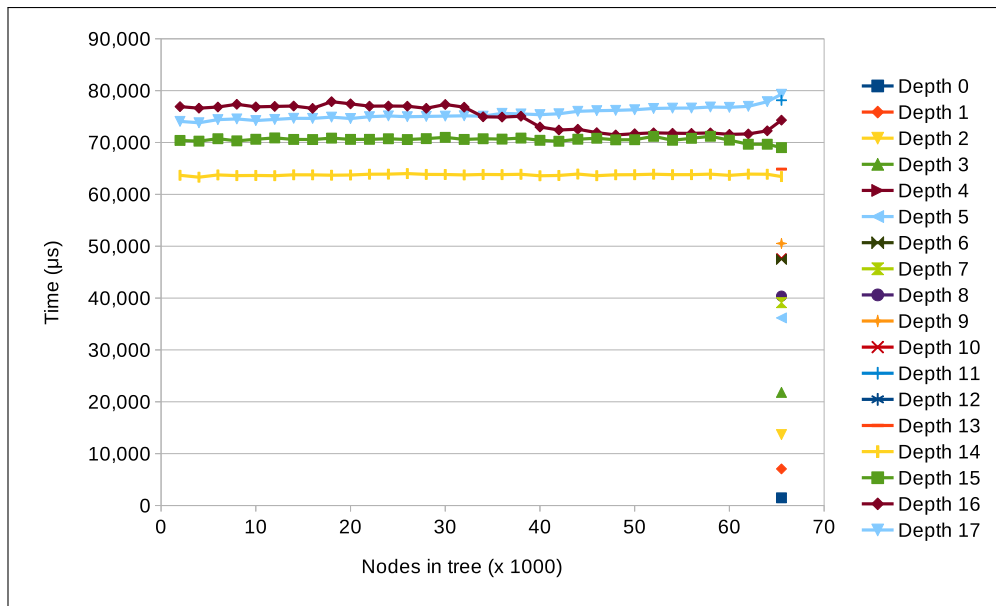
The *remove()* mean time of a population of 2000 elements. The tree is less populated on the right of the chart and more populated on the left.

Figure 63 – *remove()* average depth by nodes in a sparse vEB($2^{2^{17}}$).

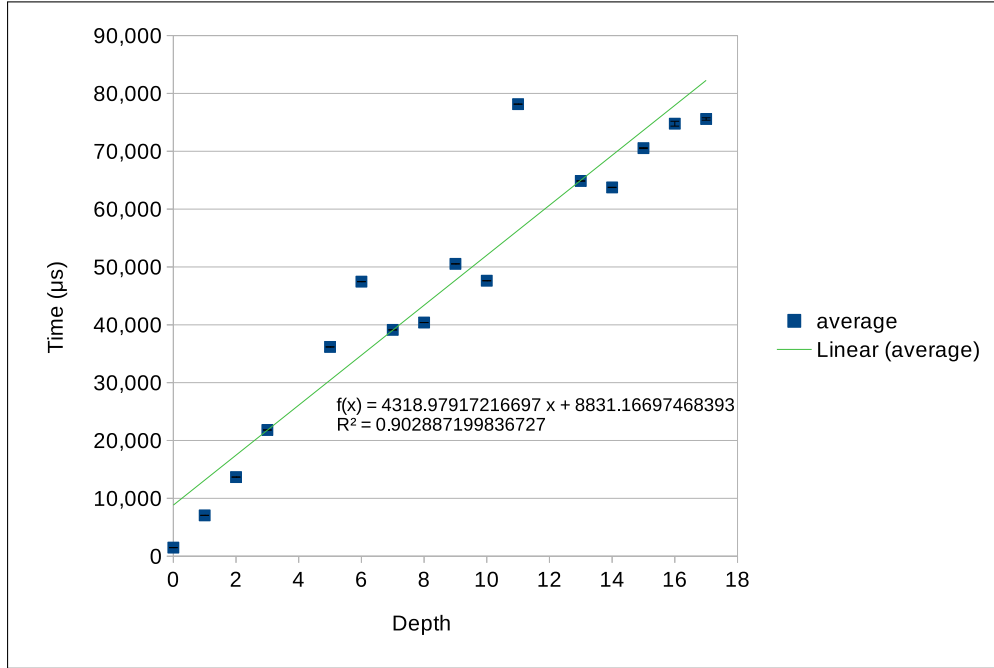
The *remove()* average depth of a population of 2000 elements. The tree is less populated on the right of the chart and more populated on the left.

Figure 64 – *remove()* depths count, of last 2k elements, by nodes in a sparse $vEB(2^{2^{17}})$.

A counting of depths taken by *remove()* operations of a population of 2000 elements. The tree is right populated on the left of the chart and more populated on the left.

Figure 65 – *remove()* depths time by nodes in a sparse $vEB(2^{2^{17}})$.

The *remove()* mean time, by depth, of a population of 2000 elements. The tree is less populated on the right of the chart and more populated on the left.

Figure 66 – *remove()* depth average time of a sparse vEB($2^{2^{17}}$).The average time taken by *remove()* operations per depth.

Like in dense trees, average time taken by *remove()* operations (Figure 62) is almost a mirror of *insert()* operations (Figure 42). The same happens regarding the average *depth* in Figures 63 and 43, and Figures 64 and 44. And also with the average time taken by *depth* (Figures 65 and 45).

In this case, because the tree is filled at the left, deleting nodes will start having to delete summaries at the left, so, more the tree is empty, more it will have to delete summaries down at the left (Algorithm 5 line 21).

Unfortunately we don't know how to explain why the average time taken by *remove()* operations at *depth* 16 decreases while slightly increases for other *depths* (Figure 65).

5.2.3 Experiments: last analysis

As we can see from all average time by *depths* (Figures 21, 26, 31, 36, 41, 46, 51, 56, 61, 66), they are linear as it ought to be. Notice $depth = \lg \lg U$, this together with the analysis we just did in the previous section, give us confidence that our distributed implementation holds $\lg \lg U$ time cost.

That said, let's "shoot in the foot". Remember the van Emde Boas recursion is $T(2^m) = T(2^{m/2}) + O(1)$ and it gives $O(\lg \lg U)$ as we transcribed in Section 3.1.

This work has been floating between theory and practice. It is theoretical work brought into practice with several aspects of both sides. Definitely our proposed distributed

vEB holds its $O(\lg \lg U)$ in theoretical field. But it may not be true in practical field, or at least very hard to see due to huge constants. That's because the computer power may not be enough to deal with it.

The statement on previous paragraph seems to be a contradiction as we just showed practical experiments proving it holds $O(\lg \lg U)$ for a huge vEB($2^{2^{17}}$). But it is not, and there are two reasons for that.

The First is the option "force_maxsize" (Appendix A), that was set to "true". We did it to make our analysis easier. In a practical final product it will be set to "false", to improve performance, and would affect the results, specially considering packages with timeouts/retry. Remember we have discarded samples with timeouts/retry from our analysis.

The second is related to the time taken by primary logic operation on keys, for instance, bitwise operations 'AND, OR, XOR, LSHIFT and RSHIFT'. These operations takes $O(1)$ time because they map directly to one cycle CPU instruction. Suppose we are working with an ARM7, it has bitwise operations for keys up to vEBt(2^{32}). Past that, it has to loop on remaining bytes to complete the operation. We have executed an extra experiment to show time taken by XOR operation on keys of different sizes using a Intel(R) Core(TM) i7-5500U CPU ⁶, capable of AVX2 instructions and therefore 256-bits operations. Please see Figures 67 and 68 for the results.

On this extra experiment we tested the time taken to find a key in the Registry (See Section B), this is basically the time to calculate a hash (Appendix D). The same analysis applies comparing keys or calculating "high" and "low" values.

As you can see on the graphs, its curve starts getting higher after *depth* 8 which is 256-bit (See Figure 67).

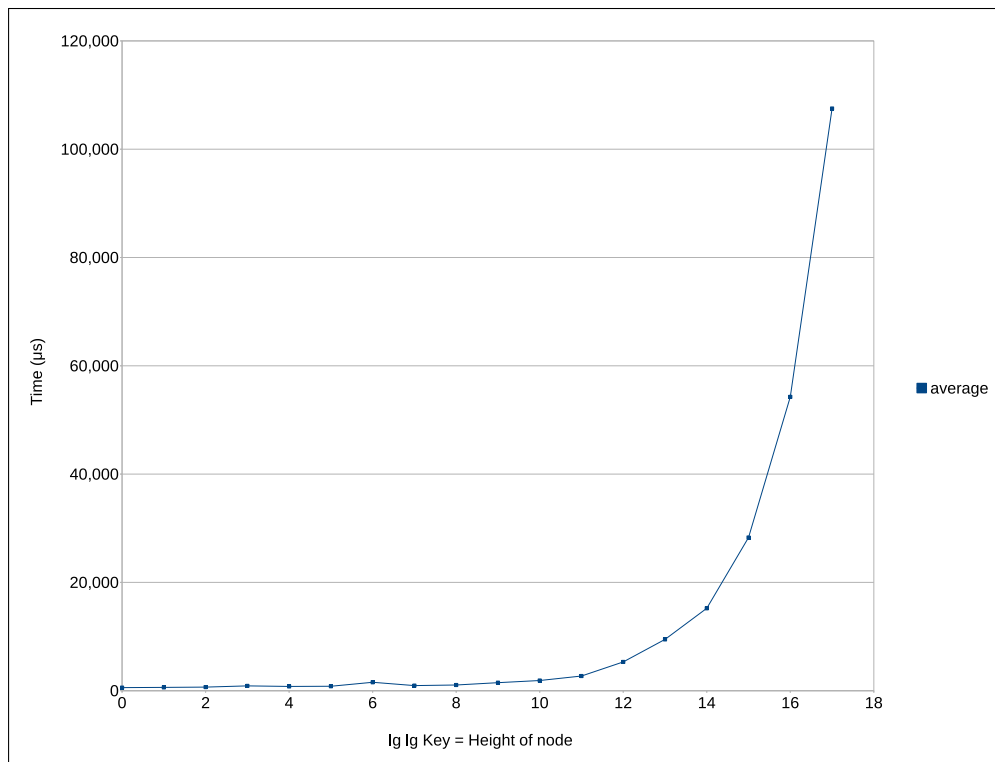
Bringing such cost into our recursion, it becomes $T(2^m) = T(2^{m/2}) + \Theta(\lg 2^m)$. And solving this recursion will end up with $\Theta(\lg U \lg \lg U)$.

That's said, we are still very comfortable with it, because this analysis just comes with a lot of preciousity and any algorithm out there, will also have its $O(1)$ comparisons transformed into $O(\lg U)$ time.

Why haven't we seen such effects on our experiments? That's because the time to calculate hashes ranges from 250 nanoseconds (not considering measurement interferences) to 50 micro-seconds and that just gets lost in fluctuations of RPCs that takes milliseconds

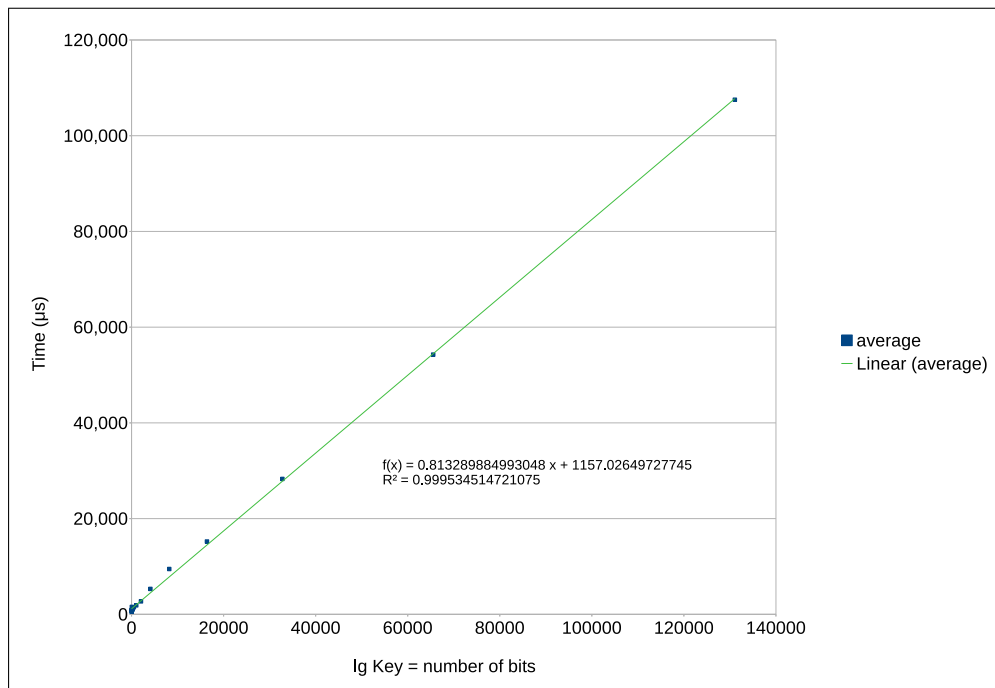
Finally, regarding the time to expand the tree from height vEB(2^m) to vEB(2^{m*2}) it is exactly the same cost of vEB(2^m).remove(vEB(2^m).min()) as we can see in Algorithm 8 and there is no need to further experiment.

⁶ <https://ark.intel.com/products/85214/Intel-Core-i7-5500U-Processor-4M-Cache-up-to-3_00-GHz>

Figure 67 – Time taken by registry.find() on 2000 keys for each *height*.

A registry.find() operation basically performs a XOR on the bytes of the key.

Figure 68 – Time taken by registry.find() on 2000 keys for keys of different sizes in bits.



A registry.find() operation basically performs a XOR on the bytes of the key.

6 Conclusion and Future work

It is hard to predict the behavior of a sparse vEB tree, specially because it is very sensitive to the chosen input data. Either way, when a huge tree eventually becomes dense, we really do expect it to work like the small dense tree shown in our experiments. For instance, insert and remove operations will become faster and search, successor and predecessor operations will become slower, but they will be limited by high and upper bounds.

The main contributions of this work are:

- We brought van Emde Boas tree, that performs *insert*, *successor*, *predecessor*, *search* and *remove* operations in $O(\lg \lg U)$ time, to the world of distributed big data structures. Our distributed vEBt solution overcomes the memory limitation of the original vEB tree, and seems to be a serious candidate to beat nowadays big data structures that performs same operations, or a subset of it, in $O(\lg n)$ time cost;
- We showed how to expand a vEB tree from 2^m universes into $2^{m \times 2}$ universes in $O(\lg \lg 2^m)$ time cost. And brought into discussion why not to expand to 2^{m+1} universes;
- We showed how to map every single node of a vEB tree into a Global Unique Id (GUID). And our GUID remains valid even if the tree expands;
- We showed how to implement a Distributed vEB tree on top of any distributed protocol as long as it supports multicast or broadcast;
- We designed and implemented a hand-crafted minimalist distributed protocol on top of UDP, and proved by experiments in local network it holds $O(\lg \lg U)$ time cost;
- We brought into discussion how a vEB tree is populated and behaves in sparse or dense trees. It is still a superficial discussion but we have not seen such discussion out there;
- We designed an architecture that is very suitable for research and experiment. Making it easy to change the police how nodes are created, *e.g.* if it is distributed or not, or even if it is a vEB node or some other structure that supports the same set of operations. The architecture also makes it easy to replace the underlying distributed protocol without affecting any other code of the solution. And finally, makes it easy to collect and save statistics from several different threads without changing methods interfaces.

6.1 Future work

- Optimizations:
 - Try to remove the “max” element from the inner trees in the same way as “min”. It would avoid having to deal with duplicated data across the network. This may also may speed up some operations;
 - Find out a good compression algorithm for serialized data. This could improve network communication;
 - Run tree requests concurrently, inspired in (KULAKOWSKI, 2013)(WANG; LIN, 2007), and try to make it $O(1)$ amortized time cost;
 - Use basic type integers (u64, u16 and u8) when the tree is at level of universe 2^{64} and lower;
 - Use fine optimizations like static polymorphism;
 - Miscellaneous improvements like document the code and write a user manual;
- Make a pseudo local version of vEB tree, changing the Linux (or others open source O.S.) kernel to supply memory from the network transparently to the process. How the vEB will benefit from OS caching mechanism? Will it have few cache-misses? In addition it is a very challenging solution that could be used every where else;
- Try to make an implementation of a vEB that accept string as keys;
- Implement and run experiment with a consistent hash Cheater;
- Investigate if we could use a consistent hash (KARGER et al., 1997) to replace our solution based in multicast;
- Network and Distributed System:
 - Make it more robust protocol;
 - Try to make a protocol that not need to rewind all RPC calls across hosting nodes. for instance, the *search* operation could return directly to the root when it hit the base case of the recursion;
 - Implement package framing to allow send more than 65k in a single message, or even work with IPv6 jumbogram packages allowing to send up to 4 GB UDP packages;
 - Tryout different protocols like CoAP, MQTT, MPI, UDT and even TCP;
 - Make experiments on WANs or Internet;

- Implement a protocol that can change dynamically based on QoS or some metrics. That would automatically choose the best underlying protocol based on the network conditions;
- Make it possible to negotiate with the peer what protocol and parameters to use. That is, negotiate capabilities;
- Add multicast TTL/HOPS to the parameter options and test with machines on Internet;
- Implement the lower communication class to make the tree work thorough infinite band (CHU; KASHYAP, 2006);
- Make experiments with link aggregation ¹;
- Compare Power Consumption using different approaches;
- Make a thin wrapper over other tree implementations, like Btree, and test its performance. For example, a Btree could be used at level 2^{32} and lower;
- Design and implement a definitive NoSQL solution:
 - Handle transactions, load balance, redundancy and fault tolerance;
 - Develop binding for other programming languages
- Benchmarking:
 - Handle transactions, redundancy and fault tolerance;
 - Compare performance against Redis, Google BigTable, Apache HBase, etc ...;
 - Use test benchmarks (SEN; FARRIS; GUERRA, 2013) out there for performance and correctness;
- Mastering the van Emde Boas structure:
 - Make an in depth study of tree behavior for several different pattern of data and predict lower and higher bounds;
 - Develop a tool to visualize the tree on a 3D virtual world, using OpenGL or Unreal Engine. The tool could would be rendered in realtime, by collecting data just like the “cheater”. In addition it could be drawn together with realtime viewer of some statistics and graphics. Also could allow parameters to be changed in realtime. This solution would be very useful for didactics and to study the behavior of data structure on different data;

¹ <<http://linux-ip.net/html/ether-bonding.html>>

Bibliography

AGUILERA, M. K.; GOLAB, W.; SHAH, M. A. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, VLDB Endowment, v. 1, n. 1, p. 598–609, ago. 2008. ISSN 2150-8097. Available from Internet: <http://dx.doi.org/10.14778/1453856.1453922>. Cited in page 58.

BELLO-ORGAZ, G.; JUNG, J. J.; CAMACHO, D. Social big data: Recent achievements and new challenges. *Information Fusion*, Elsevier BV, v. 28, p. 45–59, mar 2016. Available from Internet: <http://dx.doi.org/10.1016/j.inffus.2015.08.005>. Cited in page 26.

BOAS, P. E.; KAAS, R.; ZIJLSTRA, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, Springer Nature, v. 10, n. 1, p. 99–127, dec 1976. Available from Internet: <http://dx.doi.org/10.1007/BF01683268>. Cited 2 times in pages 22 and 25.

BOAS, P. van E. Preserving order in a forest in less than logarithmic time. In: *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1975. (SFCS '75), p. 75–84. Available from Internet: <http://dx.doi.org/10.1109/SFCS.1975.26>. Cited 2 times in pages 22 and 25.

BOAS, P. van E. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, Elsevier BV, v. 6, n. 3, p. 80–82, jun 1977. Available from Internet: [http://dx.doi.org/10.1016/0020-0190\(77\)90031-X](http://dx.doi.org/10.1016/0020-0190(77)90031-X). Cited 2 times in pages 22 and 25.

BORMAN, S. D. D.; HINDEN, R. *IPv6 Jumbograms*. [S.l.], 1999. Available from Internet: <https://tools.ietf.org/html/rfc2675>. Cited in page 65.

CHEN, H.; CHIANG, R. H. L.; STOREY, V. C. Business intelligence and analytics: From big data to big impact. *MIS Q.*, Society for Information Management and The Management Information Systems Research Center, Minneapolis, MN, USA, v. 36, n. 4, p. 1165–1188, dez. 2012. ISSN 0276-7783. Available from Internet: <http://dl.acm.org/citation.cfm?id=2481674.2481683>. Cited in page 28.

CHEN, S. et al. Towards scalable and reliable in-memory storage system: A case study with redis. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016. Available from Internet: <https://doi.org/10.1109/trustcom.2016.0255>. Cited in page 21.

CHU, J.; KASHYAP, V. *Transmission of IP over InfiniBand (IPoIB)*. [S.l.], 2006. Available from Internet: <https://tools.ietf.org/html/rfc4391>. Cited in page 115.

COMMUNITY cleverness required. *Nature*, Springer Nature, v. 455, n. 7209, p. 1–1, sep 2008. Available from Internet: <http://dx.doi.org/10.1038/455001a>. Cited in page 21.

CORMEN, T. H. et al. *Errata to Introduction to Algorithms, Third Edition*. <http://www.cs.dartmouth.edu/~thc/clrs-bugs/bugs-3e.php>. [Online; accessed 24-December-2016]. Cited in page 53.

CORMEN, T. H. et al. *Introduction to Algorithms, 3rd Edition* (MIT Press). The MIT Press, 2009. ISBN 0262033844. Available from Internet: <https://www.amazon.com/Introduction-Algorithms-3rd-MIT-Press/dp/0262033844%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262033844>. Cited 11 times in pages 22, 25, 31, 32, 34, 36, 37, 38, 39, 40, and 53.

COULOURIS, G. et al. *Distributed Systems: Concepts and Design*. Pearson, 2011. Available from Internet: <https://www.amazon.com/Distributed-Systems-Concepts-George-Coulouris-ebook/dp/B00ALTSBAQ%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3DB00ALTSBAQ>. Cited 2 times in pages 68 and 71.

DEDE, E. et al. MARISSA: MapReduce implementation for streaming science applications. In: *2012 IEEE 8th International Conference on E-Science*. Institute of Electrical and Electronics Engineers (IEEE), 2012. Available from Internet: <http://dx.doi.org/10.1109/eScience.2012.6404432>. Cited in page 28.

DEMAINE, E. D. Cache-oblivious algorithms and data structures. In: *Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS*. University of Aarhus, Denmark, 2002. Available from Internet: <http://erikdemaine.org/papers/BRICS2002/paper.pdf>. Cited 2 times in pages 22 and 25.

EGGERT, L.; FAIRHURST, G. *Unicast UDP Usage Guidelines for Application Designers*. [S.l.], 2008. Available from Internet: <https://tools.ietf.org/html/rfc5405>. Cited in page 44.

ELDAWY, A.; MOKBEL, M. F. The era of big spatial data. In: *2015 31st IEEE International Conference on Data Engineering Workshops*. Institute of Electrical and Electronics Engineers (IEEE), 2015. Available from Internet: <http://dx.doi.org/10.1109/ICDEW.2015.7129542>. Cited in page 21.

ELSHAWI, R. et al. Big graph processing systems: State-of-the-art and open challenges. In: *2015 IEEE First International Conference on Big Data Computing Service and Applications*. Institute of Electrical and Electronics Engineers (IEEE), 2015. Available from Internet: <http://dx.doi.org/10.1109/BigDataService.2015.11>. Cited in page 26.

HASHEM, I. A. T. et al. The rise of “big data” on cloud computing: Review and open research issues. *Information Systems*, Elsevier BV, v. 47, p. 98–115, jan 2015. Available from Internet: <http://dx.doi.org/10.1016/j.is.2014.07.006>. Cited in page 21.

HEFFNER, M. M. J.; CHANDLER, B. *IPv4 Reassembly Errors at High Data Rates*. [S.l.], 2007. Available from Internet: <https://tools.ietf.org/html/rfc4963>. Cited in page 44.

HINDEN, R.; DEERING, S. *Internet Protocol Version 6 (IPv6) Addressing Architecture*. [S.l.], 2003. Available from Internet: <https://tools.ietf.org/html/rfc3513>. Cited in page 54.

HINDEN, R.; HABERMAN, B. *Unique Local IPv6 Unicast Addresses*. [S.l.], 2005. Available from Internet: <https://tools.ietf.org/html/rfc4193>. Cited in page 54.

- IBM. What is big data? Available from Internet: <http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>. Cited 3 times in pages 21, 26, and 28.
- KARGER, D. et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1997. (STOC '97), p. 654–663. ISBN 0-89791-888-6. Available from Internet: <http://doi.acm.org/10.1145/258533.258660>. Cited 2 times in pages 69 and 114.
- KERNEL.ORG. *Kernel IPv6 variables*. <https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt>. [Online; accessed 24-December-2016]. Cited in page 59.
- KITCHIN, R. The real-time city? big data and smart urbanism. *SSRN Electronic Journal*, Elsevier BV. Available from Internet: <http://dx.doi.org/10.2139/ssrn.2289141>. Cited in page 28.
- KULAKOWSKI, K. A concurrent van emde boas array as a fast and simple concurrent dynamic set alternative. *Concurrency and Computation: Practice and Experience*, Wiley-Blackwell, v. 26, n. 2, p. 360–379, jan 2013. Available from Internet: <https://doi.org/10.1002/cpe.2995>. Cited 4 times in pages 22, 62, 68, and 114.
- LABRINIDIS, A.; JAGADISH, H. V. Challenges and opportunities with big data. *Proc. VLDB Endow.*, VLDB Endowment, v. 5, n. 12, p. 2032–2033, ago. 2012. ISSN 2150-8097. Available from Internet: <http://dx.doi.org/10.14778/2367502.2367572>. Cited in page 21.
- LANARI, R. From ERS-1 TO SENTINEL-1: A big data challenge for 25 years of DInSAR observations. In: *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. Institute of Electrical and Electronics Engineers (IEEE), 2015. Available from Internet: <http://dx.doi.org/10.1109/IGARSS.2015.7326054>. Cited in page 28.
- LEVIN, M. A.; WANDERER, J. P.; EHRENFELD, J. M. Data, big data, and metadata in anesthesiology. *Anesthesia & Analgesia*, Ovid Technologies (Wolters Kluwer Health), v. 121, n. 6, p. 1661–1667, dec 2015. Available from Internet: <http://dx.doi.org/10.1213/ANE.0000000000000716>. Cited in page 28.
- LI, T. et al. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Institute of Electrical and Electronics Engineers (IEEE), 2013. Available from Internet: <https://doi.org/10.1109/2Fipdps.2013.110>. Cited in page 69.
- MAITREY, S.; JHA, C. Handling big data efficiently by using map reduce technique. In: *2015 IEEE International Conference on Computational Intelligence & Communication Technology*. Institute of Electrical and Electronics Engineers (IEEE), 2015. Available from Internet: <http://dx.doi.org/10.1109/CICT.2015.140>. Cited in page 28.
- MATSUMOTO, M.; NISHIMURA, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, ACM, New York, NY, USA, v. 8, n. 1, p. 3–30, jan. 1998. ISSN 1049-3301. Available from Internet: <http://doi.acm.org/10.1145/272991.272995>. Cited in page 76.

MAURO, A. D.; GRECO, M.; GRIMALDI, M. What is big data? a consensual definition and a review of key research topics. In: . AIP Publishing, 2015. Available from Internet: <http://dx.doi.org/10.1063/1.4907823>. Cited in page 21.

MAYER-SCHÖNBERGER, V.; CUKIER, K. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. Eamon Dolan/Mariner Books, 2014. ISBN 0544227751. Available from Internet: <http://www.amazon.com/Big-Data-Revolution-Transform-Think/dp/0544227751%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0544227751>. Cited in page 28.

MERVIS, J. Agencies rally to tackle big data. *Science*, American Association for the Advancement of Science (AAAS), v. 336, n. 6077, p. 22–22, apr 2012. Available from Internet: <http://dx.doi.org/10.1126/science.336.6077.22>. Cited in page 21.

MONTEITH, S. et al. Big data are coming to psychiatry: a general introduction. *Int J Bipolar Disord*, Springer Science + Business Media, v. 3, n. 1, sep 2015. Available from Internet: <http://dx.doi.org/10.1186/s40345-015-0038-9>. Cited in page 28.

NARTEN E. NORDMARK, W. S. T.; SOLIMAN, H. *Neighbor Discovery for IP version 6 (IPv6)*. [S.l.], 2007. Available from Internet: <https://tools.ietf.org/html/rfc4861>. Cited in page 59.

PERRONS, R. K.; MCAULEY, D. The case for “n«all”: Why the big data revolution will probably happen differently in the mining sector. *Resources Policy*, Elsevier BV, v. 46, p. 234–238, dec 2015. Available from Internet: <http://dx.doi.org/10.1016/j.resourpol.2015.10.007>. Cited in page 28.

PRESS G. \$16.1 billion big data market: 2014 predictions from IDC and IIA. 2016. Available from Internet: <http://www.forbes.com/sites/gilpress/2013/12/12/16-1-billion-big-data-market-2014-predictions-from-idc-and-ia/#245d88a858ea>. Cited 2 times in pages 27 and 28.

RANI, G.; KUMAR, S. Hadoop technology to analyze big data. *International Journal of Engineering Development and Research (IJEDR)*, v. 3, n. 4, p. 949–952, dec 2015. ISSN 2321-9939. Available from Internet: <http://www.ijedr.org/papers/IJEDR1504167.pdf>. Cited in page 21.

SEN, R.; FARRIS, A.; GUERRA, P. Benchmarking apache accumulo BigData distributed table store using its continuous test suite. In: *2013 IEEE International Congress on Big Data*. IEEE, 2013. Available from Internet: <https://doi.org/10.1109/bigdata.congress.2013.51>. Cited 2 times in pages 22 and 115.

SINTEF. Big data, for better or worse: 90two years. 2013. Available from Internet: www.sciencedaily.com/releases/2013/05/130522085217.htm. Cited 3 times in pages 21, 26, and 28.

TANENBAUM, A. S.; WETHERALL, D. J. *Computer Networks*. Pearson, 2012. Available from Internet: <https://www.amazon.com/Computer-Networks-Andrew-S-Tanenbaum-ebook/dp/B006Y1BKGC%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3DB006Y1BKGC>. Cited 4 times in pages 41, 43, 44, and 65.

U.S., Canada Collaborate on Big Data in ASD Research. *The ASHA Leader*, American Speech Language Hearing Association, v. 20, n. 12, p. 16, dec 2015. Available from Internet: <http://dx.doi.org/10.1044/leader.NIB4.20122015.16>. Cited in page 28.

WANG, H.; LIN, B. Pipelined van Emde Boas tree: Algorithms, analysis, and applications. In: *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. Institute of Electrical & Electronics Engineers (IEEE), 2007. Available from Internet: <http://dx.doi.org/10.1109/INFCOM.2007.303>. Cited 4 times in pages 22, 62, 68, and 114.

WANG, Z.; YU, Q. Privacy trust crisis of personal data in China in the era of big data: The survey and countermeasures. *Computer Law & Security Review*, Elsevier BV, v. 31, n. 6, p. 782–792, dec 2015. Available from Internet: <http://dx.doi.org/10.1016/j.clsr.2015.08.006>. Cited in page 26.

WARD, J. S.; BARKER, A. *Undefined By Data: A Survey of Big Data Definitions*. 2013. Cited in page 21.

ZHENG, R. et al. SKVM: Scaling in-memory key-value store on multicore. In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. Institute of Electrical and Electronics Engineers (IEEE), 2015. Available from Internet: <http://dx.doi.org/10.1109/ISCC.2015.7405580>. Cited in page 26.

Appendix

APPENDIX A – Program Options

The testing program can receive the following command line options:

- port - The Multicast port number
- multicast_group - The Multicast address
- role - The role of the application (node).
 - "root" - The application hold the tree root.
 - "client" - The application is just a client to a root. (not implemented)
 - "node" - The application will host nodes.
 - "cheater" - The application will be a Cheater.
- timeout - Timeout in milliseconds to wait for Acks
- jump_multi - Used to calculate long waits
- retries - Number of retransmissions in case doesn't receive an acknowledgment.
- enough_servers_available - Max number of server to randomly choose to create a node. The client node will wait "timeout" until "enough_servers_available" host servers respond before choose between them;
- memory_threshold - The minimal amount of free memory required to host a vEB node.
- udp_buf_size - The size of the UPD message (id + key + metadata = $16384 * 2 + 1024$)
- thread_pool_size - The initial size of threads in pool to handle received messages;
- log_level - The verbosity of log
 - "trace" - Very high verbose messages that generates a lot out information to help developer in troubleshooting/debugging tasks;
 - "debug" - Debugging messages for developers;
 - "info" - Most relevant output messages for user;
 - "warning" - Warning messages that can't indicate potential problems;
 - "error" - Error messages that application can't recover from.

- `root_uuid` - Unique GUID of the Root tree (when "role" option is set to "root")
- `force_maxsize` - If true all packages (excepts some types of acknowledgments) are set to "udp_buf_size";
- `statistics.summary_only` - If true, print only statistics summary, otherwise prints all collected entries;
- `multicast_loopback` - If true, can't send/receive its own messages
- `no_hosting` - If true, will host vEB nodes. (can only be used with "role" of to "root" or "node");
- `self_cheating` - Useful when developing using a stand-alone machine.
- `test.run` - If true will run tests, only make sense with "role" of "root" or "client";
- `test.maxbits` - The test generates number randomly. This is the number of bits used to represent such numbers;
- `test.mode` - The type of the test that will be executed
 - "performance" - To test performance
 - "correctness" - To test correctness
- `has_cheater` - If true, will send a `answer.alloc_cheater` message and wait for cheater AckF;
- `service` - Usually it will be true for "node" and "cheater" applications, for root, it true, the application exists finishes after the test in done.

APPENDIX B – Software Architecture

The software architecture must be designed to satisfy the following software requirements. The ones listed first have priority over the next ones:

1. The architecture must allow the implementation of this research in time;
2. The architecture shall be as extensible as possible to make it easy to implement most of the future works presented in Conclusion chapter;
3. Make it possible to write code for experiments described in this document;
4. Make it easy to collect statistics from the experiments;
5. It must be easy to configure the arrangement of Nodes on the network. As an example, in one experiment we could make all 2^{16} nodes local, on another experiment we may want to have local nodes only if 30% of RAM memory available, and in another experiment we may want to always make nodes remote, so there is always an RPC between nodes on the tree;
6. Ultimate performance, while speed has a bit of priority over memory;
7. Code must run on Linux, if possible on Windows and Android too;
8. It must be possible to compile the code from an IDE or command line;
9. Code and comments must all be written in English to make it easier for others researchers to engage;
10. the code must be developed in C or C++;
11. whenever suitable use Design Patterns.

With the requirements in mind, let's see how our software architecture evolved by looking some class diagrams.

In Figure 69 we depict the most important class of our Tree, the trees themselves.

- ITree - The interface that defines all operations any tree must implement;
- IVebGlobal - The interface any vEB tree must implement. Notice it is just the ITree interface plus an "expanded()" method that is used when expanding the universe of our whole tree;

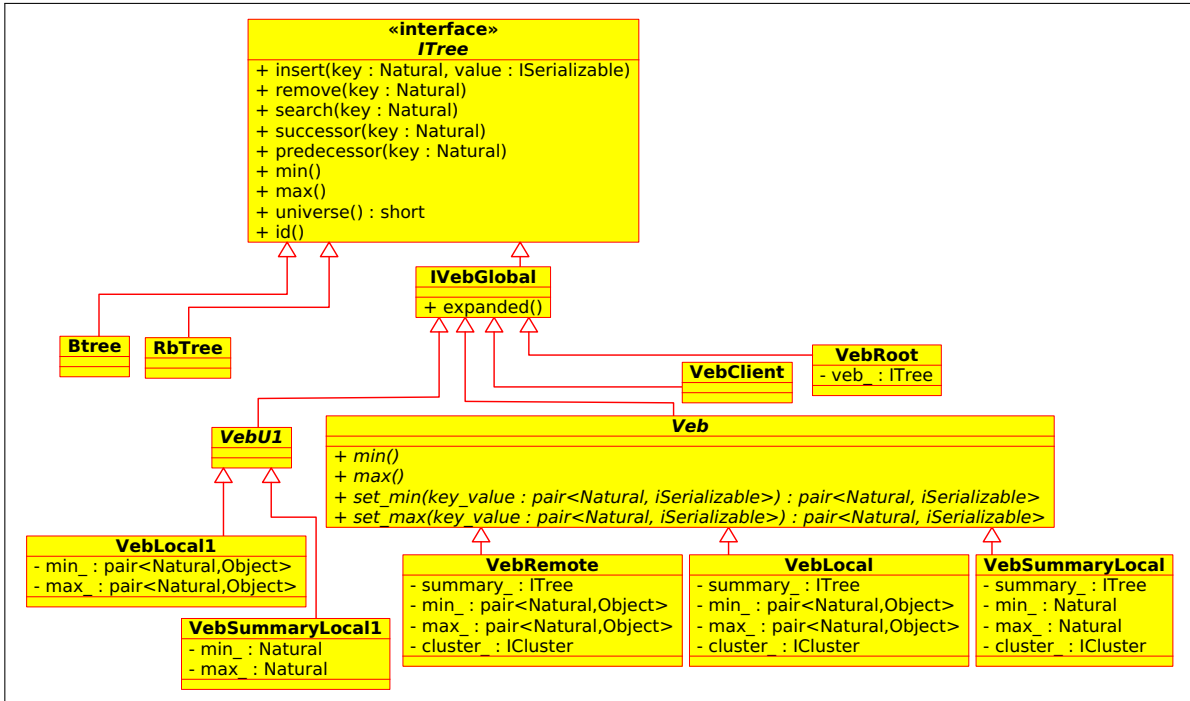


Figure 69 – Trees class diagram.

- Veb - It is, together with VebU1, the most important classes. It is an abstract class that implements all vEB algorithms, described in Section 3.1 of this document;
- VebU1 - Like "Veb", but this class implements the algorithms vEB tree with universe size of 2, while Veb implements trees where universe size is greater than 2;
- VebClient - It is a vEB that doesn't implement any real vEB logic, it simply just call a RPC method on a real tree somewhere else. So, this class depends on a underlying RPC mechanism we will see soon in this section;
- VebRoot - This class also doesn't implement any vEB algorithms. It uses the Proxy Design Pattern to delegates methods the real vEB class, that can be any class that inherits from IVebGlobal; Actually, a VebRoot really implements one single method, "expand()", this method is called automatically by VebRoot, before a delegated call to "insert()", if the key value is greater than then vEB universe; In addition to that, VebRoot also collects statistics D;
- VebLocal - It is a regular implementation of a vEB tree, with a VebU1 or Veb summary and a local ClusterArray or ClusterHashTable (we will see our Cluster implementations on the next figure);
- VebSummaryLocal - It is like VebLocal but its elements doesn't hold satellite data;
- VebRemote - It is a Veb where its summary is either a VebU1 (only with universe is 2) or a VebClient and its cluster is a ClusterRemote.

- VebLocal1 - It is used to implement a local vEB of universe 2;
- VebSummaryLocal1 - It a VebLocal1 for summaries, i.e. a VebLocal1 without satellite data;
- BTree and RbTree - It implements a B-Tree and RedBlack-Tree respectively. Actually, they haven't been implemented, but they are here just to show that we could have such trees at almost any place of our solution where a ITree is used, for instance, summaries, and tree pointed by clusters;

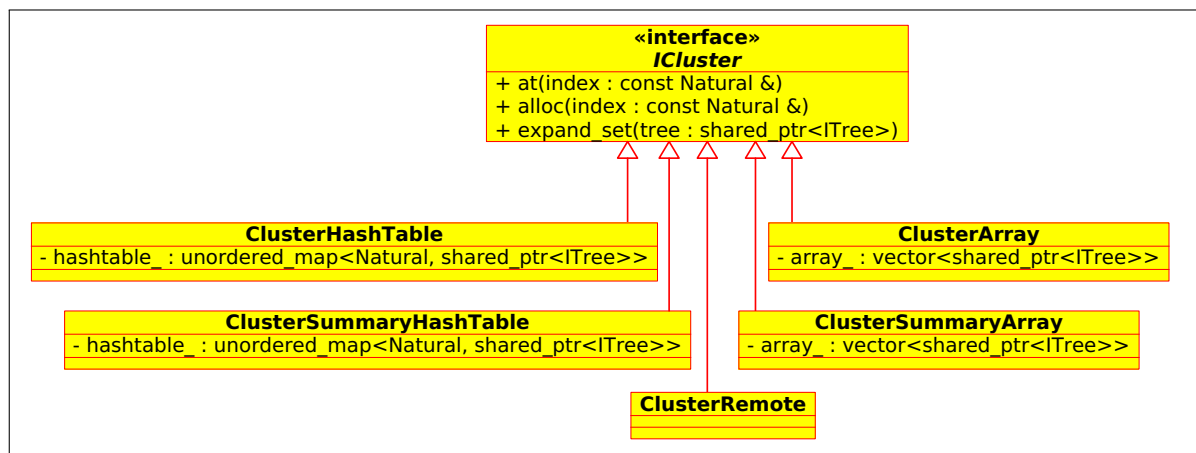


Figure 70 – Clusters class diagram.

Let's see the design of clusters shown in [Figure 70](#)

- ICluster - Cluster's interface;
- ClusterArray and ClusterSummaryArray - It is the regular cluster implemented as an array. Used by classes like VebLocal and VebSummaryLocal;
- ClusterHashTable and ClusterSummaryHashTable - It is a local cluster, but implemented as a hash table instead of an array. It has been developed and tested just to prove our architecture but wasn't really used in any of our experiments;
- ClusterRemote - It is the cluster instantiated by VebRemote, when elements of this cluster are accessed a RPC takes places;

Let's see the design of class involved in RPC shown in [Figure 71](#), [Figure 72](#) and [Figure 73](#).

- ITreeRpc - This interface is a copy of ITree interface. This is the interface used by VebClient to execute method methods;

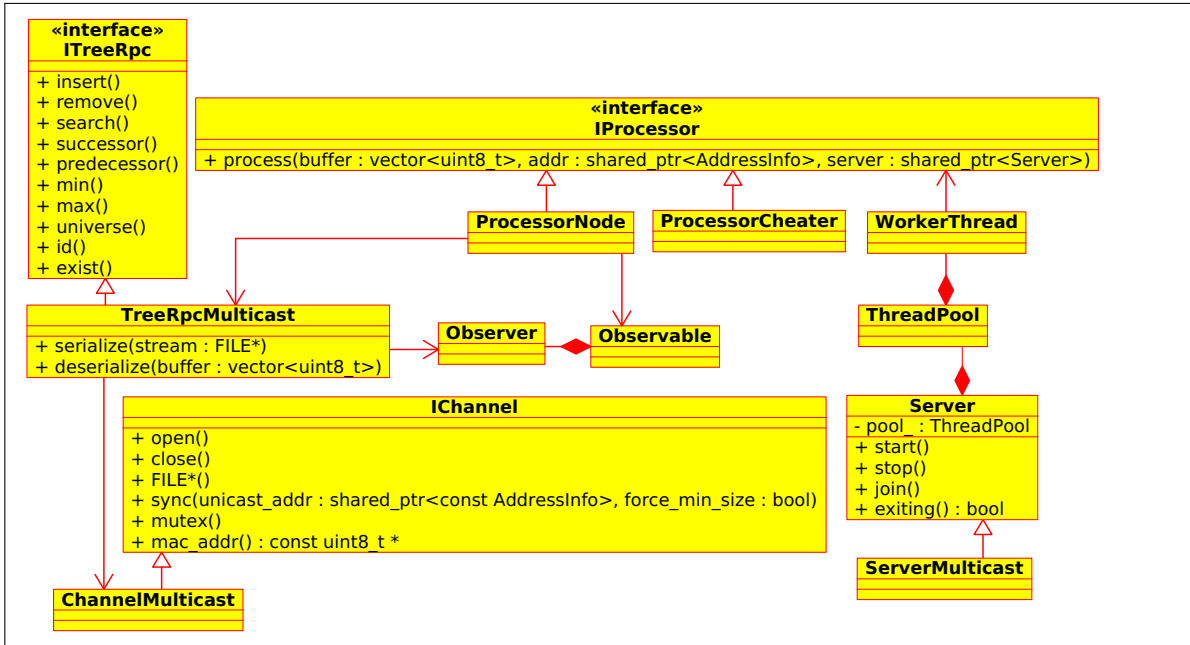


Figure 71 – RPC class diagram.

- **TreeRpcMulticast** - Is the class responsible to marshalling/unmarshalling a RPC call. This class also has some knowledge of its underlying channel, for instance ChannelMulticast, and implements all client RPC logic, like waiting and retries. This class is couple with ProcessorNode and ProcessorCheater;
- **IChannel** and **ChannelMulticast** - This class implements the client side communication channel. ChannelMulticast can be used to write UDP unicast and multicast messages. As an example, it could be implemented a ChannelTcp class that handles TCP messages, or ever ChannelRS232 that handles RS232 messages, but, in both cases, probably TreeRpcMulticast should be replaced for equivalent classes to optimize for underlying channel. It is also important to say that it is an unidirectional communication channel; This class, also works like a service that handle and synchronizes calls from multiple threads;
- **Server** and **ServerMulticast** - It is the peer class for IChannel and ChannelMulticast classes. Server classes is the host side communication channel. It is also important to say that it is an unidirectional communication channel. There will be only one instance of ChannelMulticast, and received raw messages are add to the ThreadPool to be processed;
- **ProcessorNode** - Once a message is received by the Server and added to the ThreadPool, an WorkerThread is created or awoken to handle the raw received buffer for processing. It then calls TreeRpcMulticast to parse the buffer, checks if Observable is there is an Observer waiting for such message, and if not, if handle the RPC call.

This class is also responsible to handle duplicated request messages. Usually it will be handling method calls and AckF messages;

- ProcessorCheater - It is like ProcessorNode but much more simply, just tracking method request and node creations.

Except ITreeRpc, Observable, Observer, WorkerThread and ThreadPool class in Figure 71, all other classes are couple, so if you need to implement another RPC mechanism probably you will have to implement them all.

Other classes that worths highlight are:

- Natural - This class encapsulates a GMP integer ¹ and has helper functions to handle huge number. In this research we are working with number from 0 to $2^{2^{17}} - 1$;
- Id - It has method to manipulate and map node Ids D;
- Transaction - This class uniquely identifies a RPC call in the whole system. Because of this class it is possible to respond/receive Acks for the right RPCs and to detect duplicated messages;
- Registry - It is basically a hash table of all node Ids hosted by a "node" machine;
- Factory - It is used to dynamically instantiate the right vEB, for instance, when a VebRoot is expanding or when handling the request to create a new node D;
- Properties - This class is in charge of parsing program input parameters A. A object of this class is instantiated as a Singleton Design Patterns, and several pieces of the program queries the parameters independently. We have decided to use Singleton over Dependency Injection to avoid overwhelm the code with everywhere passing a reference to the Properties object;

And last, but not least, Statistics class.

The Statistics class was designed to be used in several parts of the code, more than that, it may be used in reentrant code called simultaneous by several threads adding statistics data to completely unrelated tasks. For that, we had decided to use a sort of Singleton per thread approach.

When a thread decides it wants to collect statistics, it calls `include_thread_sample()` D. It creates an Object for that thread that holds all statistics collected to that thread. Once that thread is done, it calls `normalize_level()` D to calculate how many levels that

¹ <<https://gmplib.org/>>

method went down in the vEB tree, and finally calls “`Statistic::Statistic::add()`” to save it. Have a look in `VebRoot::insert()` [D](#) as an example.

On the host side, the `ProcessorNode` class also calls `include_thread_sample()` to start collecting statistics, the Veb trees call “`Statistic::thread_sample_set_level(universe_)`” to set the deepest level a method went in, then, the host side calls `get_thread_sample()` just before respond it back serialized to the client. Then finally, `TreeRpc` client side updates the statistics with peer collected statistics and with the number of timeouts and RPC calls [D](#) (lines 10, 28, 29, 117-119, 136-138). Also notice that, lines 28-29 executes in different a `WorkerThread` than line 10, it is allowed as long as the thread running line 10 doesn’t finish before threads running lines 28-29, and that will be the case, because the “observer” handle (line 17) will remove itself from the `Observable` when its destructor is called.

Now let’s revisit the initial requirements and double check if our architecture deals with it.

1. The architecture must allow the implementation of this research in time;

It was required 3 more months to finish our research, but we believe it did so, I guess we implemented the minimalist software able to run the experiments. We have tried simpler solution but none could survive to our experiments.

2. The architecture shall be as extensible as possible to make it easy to implement most of the future works presented in Conclusion chapter;

Yes, the RPC classes are separated, the `Factory` class allows to create very flexible tree topology, and other classes are encapsulated and not dependent on RPC or vEB classes. Due to the `ITree` interface we can virtually have any node implemented by any `Tree`. Also, the Id mapping makes it possible to create a Network-Agnostic solution.

3. Make it possible to write code for experiments described in this document;

Yes, as we did so.

4. Make it easy to collect statistics from the experiments;

Yes, we did so. We developed an easy to use API plus a module save statistics in csv format just ready to be imported by tools like LibreOffice Calc. See [Figure 80](#).

5. It must be easy to configure the arrangement of Nodes on the network. As an example, in one experiment we could make all 2^{16} nodes local, on another experiment we may want to have local nodes only if 30% of RAM memory available, and in another experiment we may want to always make nodes remote, so there is always an RPC between nodes on the tree;

Yes, it is possible due to `Factory` class.

6. Ultimate performance, while speed has a bit of priority over memory;

To be honest, in order to comply with the previous requirements and due to time restrictions we haven't really look at this very deeply. You could have used some low latency C++ programming techniques here but we didn't by the time of this writing.

7. Code must run on Linux, if possible on Windows and Android too

We have only tested on Linux. Due to uncommon characteristics to this software, we decided to implement the RPC module using BSD sockets. So, it won't compile in Windows. Probably the RPC module could be written using Boost library but it will left for future work.

8. it must be possible to compile the code from an IDE or command line

Yes, we are compiling it using CMake, using both QT Creator IDE and command line.

9. code and comments must all be written in English to make it easier for others researchers to engage;

Yes.

10. the code must be developed in C or C++;

Yes, C++, Boost, GMP.

11. whenever suitable use Design Patterns.

Yes, we have Singleton, Proxy, Observer, Factory and Thread Pool Design Partners, but to be honest we hadn't too much time to really consider more than it, there may be some places we could apply Design Patterns for better software quality, probably a good time for it would be when adopting C++ low latency techniques.

The software was a way very complex to implement. Node are running several threads concurrently, their access tree Register, send/receive messages simultaneous from different source. All of this together with the complexity of dealing with huge 131072 bit number and the complexity of implementing a UPD based RPC mechanism implemented, i.e. implement retransmission, duplication detection, congestion and flow control on unicast and multicast messages. Because of this, to make it work we have the following additional techniques or tools:

- We customized Boost log library to fill our needs. See a typical log output in Figure 74;
- tcpdump was heavily used. See Figures 75 and 76;

- Scripts to notify when some test finishes. See Figure 77;
- Valgring ² memory leak detection tool used. See Figure 78. At the beginning we didn't used it because using Modern C++ pretty much eliminates leak problems, but we are using a C library, lib GMP, and thanks to Valgrind we found a leak in a particular situation using lib GMP and fixed it;
- Even tough there is only one developer, due to the complexity of the code, using a decent source control tool was crucial. We have used git on a free on-line repository Bitbucket ³. See Figure 79.

In this chapter we have highlighted the most important features of the software architecture used in this project. The complete source code can be found in <https://bitbucket.org/dveb/dveb>, you can also send an email to the author "*Edgard Lima* <*edgard.lima@gmail.com*>" asking repository access.

² <http://valgrind.org/>

³ <https://bitbucket.org/>

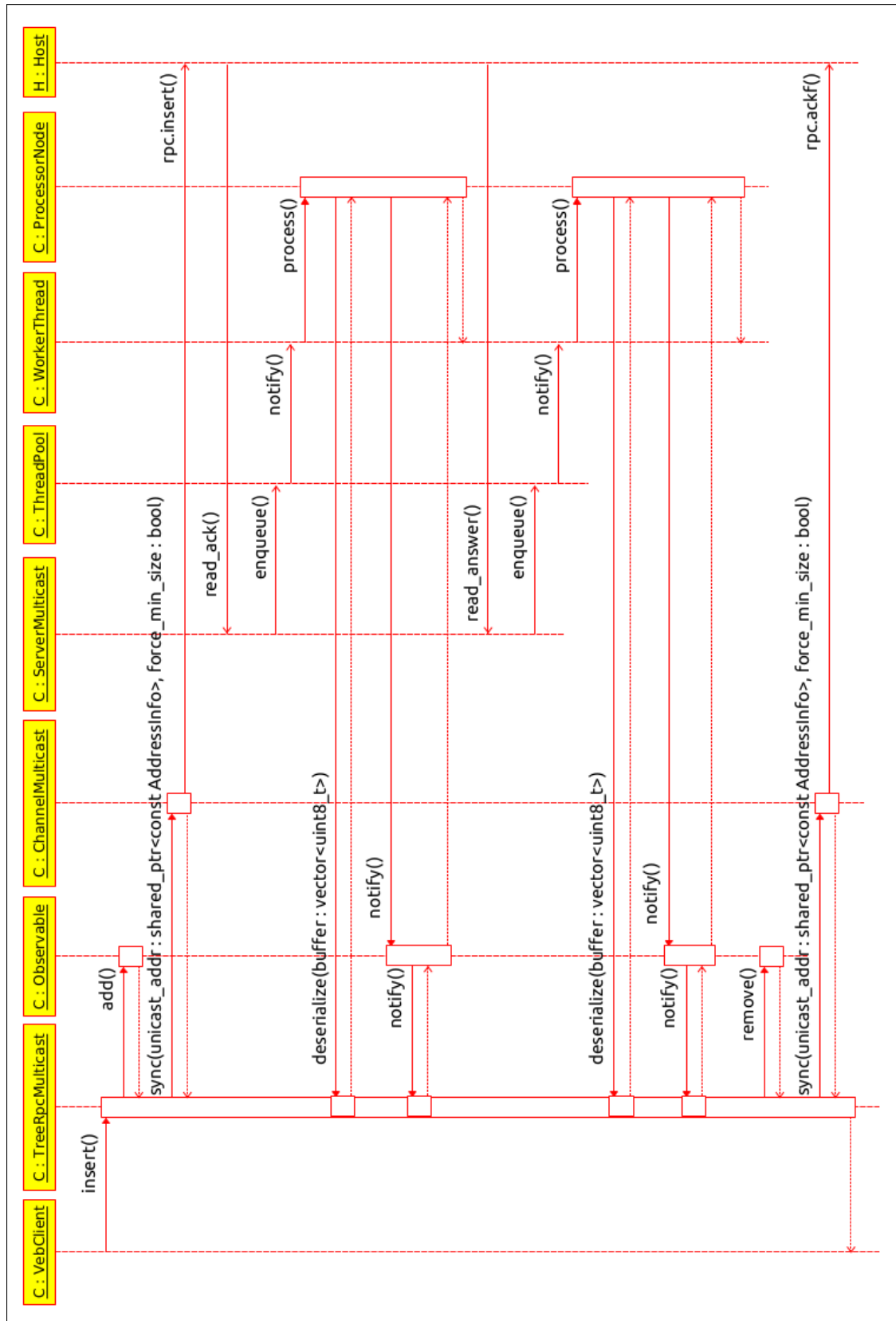


Figure 72 – RPC Client PoV.

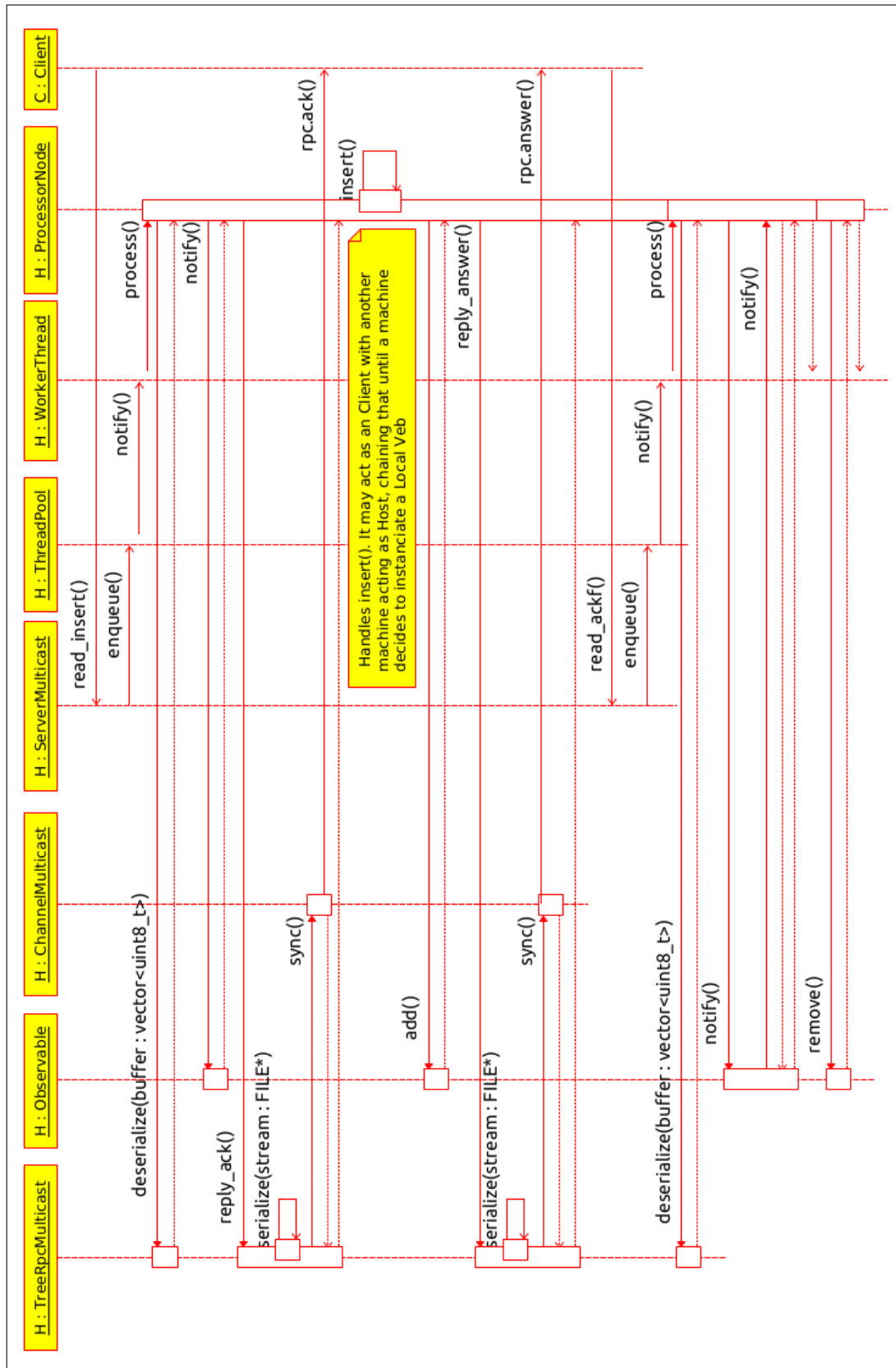


Figure 73 – RPC Host PoV.

Figure 74 – Typical log output.

```

edgard@edgard-Aspire-E5-573G: ~/Documents/Projects/Mestrado/Projects/dveb/workspace/build
2017-Mar-30 09:41:05.821376[7f1bce749740] <info> Port: 34343
2017-Mar-30 09:41:05.821439[7f1bce749740] <info> System Memory: 8.286.810.112
2017-Mar-30 09:41:05.821638[7f1bce749740] <info> Available Memory: 4.716.613.632
2017-Mar-30 09:41:05.821784[7f1bce749740] <info> Role: root
2017-Mar-30 09:41:05.821842[7f1bce749740] <warning> Memory threshold is 0%. At least 15% is recommended.
2017-Mar-30 09:41:05.822406[7f1bcac9d700] <trace> Worker waiting for Task
2017-Mar-30 09:41:05.822458[7f1bce749740] <trace> SND_BUFF 268435456
2017-Mar-30 09:41:05.822566[7f1bc3fff700] <trace> Worker waiting for Task
2017-Mar-30 09:41:05.822706[7f1bce749740] <info> Owning root 0
2017-Mar-30 09:41:05.822772[7f1bce749740] <debug> Starting server.
2017-Mar-30 09:41:05.822701[7f1bcb49e700] <trace> Worker waiting for Task
2017-Mar-30 09:41:05.822848[7f1bce749740] <trace> RCV_BUFF 268435456
2017-Mar-30 09:41:05.822871[7f1bcb9cf700] <trace> Worker waiting for Task
2017-Mar-30 09:41:05.822971[7f1bce749740] <debug> Starting test.
2017-Mar-30 09:41:05.822971[7f1bca49c700] <trace> Server Selecting
Running 1 test case...
2017-Mar-30 09:41:05.823484[7f1bce749740] <debug> test_tree_correctness_big_no_check
2017-Mar-30 09:41:05.823614[7f1bce749740] <info> preparing input data...
2017-Mar-30 09:41:05.823709[7f1bce749740] <info> input data ready
2017-Mar-30 09:41:05.823773[7f1bce749740] <info> Performance test
2017-Mar-30 09:41:05.823842[7f1bce749740] <info> kInsert - 1/10
2017-Mar-30 09:41:05.823935[7f1bce749740] <trace> Key under test (kInsert) 16 7e89
2017-Mar-30 09:41:05.824209[7f1bce749740] <trace> Observer 0x843710 created for kAnswerMin
2017-Mar-30 09:41:05.824295[7f1bce749740] <trace> Observer 0x8437f0 created for&& kAck
2017-Mar-30 09:41:05.824364[7f1bce749740] <trace> Observer 0x8439a0 created for&& kNack
2017-Mar-30 09:41:05.824455[7f1bce749740] <trace> Sending method kMin with transaction 2.5c:c9:d3:5f:8a:ad: retry 0
2017-Mar-30 09:41:05.824524[7f1bce749740] <trace> Sending to 225.0.0.37:10118
2017-Mar-30 09:41:05.824649[7f1bce749740] <trace> 33792 bytes sent
2017-Mar-30 09:41:05.824711[7f1bce749740] <trace> Universe 1 Id 1_0-natural_invalid Transaction 2.5c:c9:d3:5f:8a:ad
: Method kMin to 225.0.0.37:34343 waiting for ack
2017-Mar-30 09:41:05.826864[7f1bce749740] <trace> last_step_incremented to 1
2017-Mar-30 09:41:05.826939[7f1bce749740] <trace> Sending method kMin with transaction 2.5c:c9:d3:5f:8a:ad: retry 1
2017-Mar-30 09:41:05.827009[7f1bce749740] <trace> Sending to 225.0.0.37:10118
2017-Mar-30 09:41:05.827119[7f1bce749740] <trace> 33792 bytes sent
2017-Mar-30 09:41:05.827186[7f1bce749740] <trace> Universe 1 Id 1_0-natural_invalid Transaction 2.5c:c9:d3:5f:8a:ad
: Method kMin to 225.0.0.37:34343 waiting for ack
2017-Mar-30 09:41:05.831360[7f1bce749740] <error> Universe 1 Id 1_0-natural_invalid Transaction 2.5c:c9:d3:5f:8a:ad
: Method kMin got no ack after last timeout of 4 ms
2017-Mar-30 09:41:05.831626[7f1bce749740] <trace> Observer 0x8439a0 removed of kNack
2017-Mar-30 09:41:05.831884[7f1bce749740] <trace> Observer 0x8437f0 removed of kAck
2017-Mar-30 09:41:05.833478[7f1bce749740] <trace> Observer 0x843710 removed of kAnswerMin
2017-Mar-30 09:41:05.833633[7f1bce749740] <fatal> Except with i = 0. We got no Ack
unknown location(0): fatal error in "test_tree_correctness_big_no_check": std::runtime_error: We got no Ack

*** 1 failure detected in test suite "Master Test Suite"
2017-Mar-30 09:41:05.833783[7f1bce749740] <info> Printing statistics.
2017-Mar-30 09:41:05.833839[7f1bce749740] <info> Experiment finished in 10.813 us
2017-Mar-30 09:41:05.837503[7f1bce749740] <info> Joining server.
2017-Mar-30 09:41:05.837671[7f1bca49c700] <trace> Server Select done. rsel 1

```

The first field is the 'timestamp' with micro-seconds precision, the second field is the 'thread id', and the third field is the 'severity'. All of this, including coloring by severity level, are done automatically. The application only provides the fourth field, the 'message', with the indication of the severity, just using conventional C++ insertion operator ("<<").

Figure 75 – TCP dump.

```

edgard@edgard-Aspire-E5-573G: ~/Documents/Projects/Mestrado/Projects/dweb/workspace/build
192.168.0.100.34112 > 192.168.0.100.34343: [bad udp cksum 0x8258 -> 0xda19!] UDP, length 3
8
0x0000: 0000 0304 0006 0000 0000 0000 0000 0800 .....
0x0010: 4500 0042 5ffb 4000 4011 5897 c0a8 0064 E..B_.@.X....d
0x0020: c0a8 0064 8540 8627 002e 8258 0400 0000 ...d.@.'...X....
0x0030: 2786 0000 0000 0000 0000 0000 7b73 061a '{s..
0x0040: 0000 1c39 4754 8656 0000 0000 0100 0000 ...9GT.V.....
0x0050: 0000
16:52:22.040530 Out 1c:39:47:54:86:56 (oui Unknown) ethertype IPv4 (0x0800), length 1516: (tos
0x0, ttl 1, id 33158, offset 0, flags [+], proto UDP (17), length 1500)
192.168.0.100.34112 > 225.0.0.37.34343: UDP, bad length 33792 > 1472
0x0000: 0004 0001 0006 1c39 4754 8656 0000 0800 .....9GT.V....
0x0010: 4500 05dc 8186 2000 0111 7059 c0a8 0064 E.....pY...d
0x0020: e100 0025 8540 8627 8408 b3fa 0900 0000 ...%.@.'.....
0x0030: 2786 0000 0000 0000 0000 0000 7b7b 061a '{.....{..
0x0040: 0000 1c39 4754 8656 1000 0000 00e0 ff01 ...9GT.V.....
0x0050: 2820 0000 0000 0000 04a8 0f00 0001 1000 (.....
16:52:22.040533 Out 1c:39:47:54:86:56 (oui Unknown) ethertype IPv4 (0x0800), length 1516: (tos
0x0, ttl 1, id 33158, offset 1480, flags [+], proto UDP (17), length 1500)
192.168.0.100 > 225.0.0.37: udp
0x0000: 0004 0001 0006 1c39 4754 8656 0000 0800 .....9GT.V....
0x0010: 4500 05dc 8186 20b9 0111 6fa0 c0a8 0064 E.....o....d
0x0020: e100 0025 0000 0000 0000 0000 0000 0000 ...%.@.'.....
0x0030: 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0040: 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0050: 0000 0000 0000 0000 0000 0000 0000 0000 .....
16:52:22.040535 Out 1c:39:47:54:86:56 (oui Unknown) ethertype IPv4 (0x0800), length 1516: (tos
0x0, ttl 1, id 33158, offset 2960, flags [+], proto UDP (17), length 1500)
510436,47 99%

```

The blue square highlights the first 96 bytes of a UDP package containing the marshelled RPC call showed in Figure 76. For instance, “0900 0000” is the insert() method call, “2786” is the port number of the sender it will be replied to, the four next 0000’s are the tree uuid, “0000 7b7b” is the transaction sequential number, “061a 0000” is the process pid, “1c39 4754 8656” is the mac address of the sender, “1000” is the universe, “00e0 ff01” is the node id (it appears on the first line of the Figure 76 because this is a stand-alone run with ‘self_cheater’ option set to true).

Figure 76 – Trace log of a Insert operation.

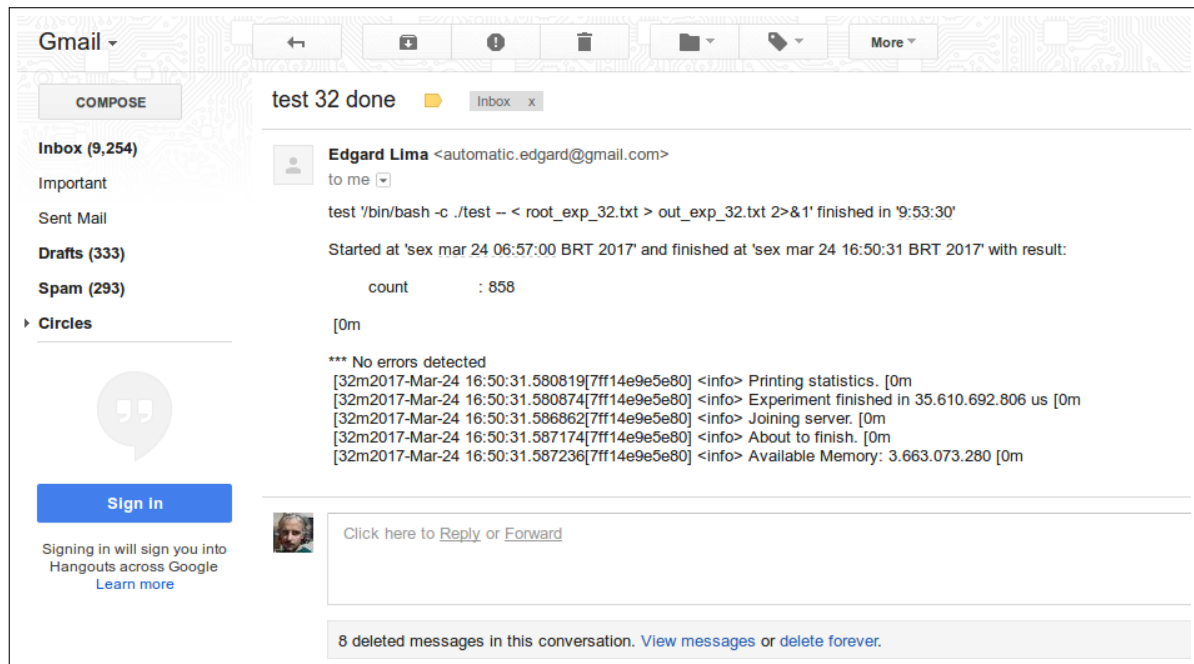
```

edgard@edgard-Aspire-E5-573G: ~/Documents/Projects/Mestrado/Projects/dveb/workspace/build
2017-Mar-24 16:52:22.040999[7f6fa67fc700] <trace> Universe 32 Id 6_1ffe0-natural_invalid Trans
action 7b7b.1c:39:47:54:86:56: Method kInsert to 225.0.0.37:34343 got ack but result. Lets wai
t for 3963618540 ms^[[0m
2017-Mar-24 16:52:22.041009[7f6f9d7fa700] <trace> Task started. Pool size is 0^[[0m
2017-Mar-24 16:52:22.041071[7f6faeee5700] <trace> Server reading^[[0m
2017-Mar-24 16:52:22.041144[7f6faeee5700] <trace> Server read -1 bytes^[[0m
2017-Mar-24 16:52:22.041111[7f6f9d7fa700] <trace> Received kInsert for 7b7b.1c:39:47:54:86:56:
Key 16 9e9b universe 16 from 192.168.0.100:16517^[[0m
2017-Mar-24 16:52:22.041207[7f6faeee5700] <trace> Server Selecting^[[0m
2017-Mar-24 16:52:22.041243[7f6f9d7fa700] <trace> Observer 0x7f6f5001daf0 created for&& kAckF^
[[0m
2017-Mar-24 16:52:22.041336[7f6f9d7fa700] <trace> We don't own node of 0 requested in 7b7b.1c:
39:47:54:86:56: for kInsert id 5_1ffe0-32_a80f0000_0^[[0m
2017-Mar-24 16:52:22.041413[7f6f9d7fa700] <trace> Replying kAck to transaction 7b7b.1c:39:47:
54:86:56: to 192.168.0.100:10118^[[0m
2017-Mar-24 16:52:22.041481[7f6f9d7fa700] <trace> Sending to 192.168.0.100:10118^[[0m
2017-Mar-24 16:52:22.041560[7f6faeee5700] <trace> Server Select done. rsel 1^[[0m
2017-Mar-24 16:52:22.041560[7f6f9d7fa700] <trace> 38 bytes sent^[[0m
2017-Mar-24 16:52:22.041624[7f6faeee5700] <trace> Server reading^[[0m
2017-Mar-24 16:52:22.041632[7f6f9d7fa700] <trace> Observer 0x7f6f5001daf0 removed of kAckF^[[0
m
2017-Mar-24 16:52:22.041683[7f6faeee5700] <trace> Server read 38 bytes^[[0m
223429,60 99%

```

Notice some data printed here in Figure 75.

Figure 77 – Automated e-mail.



Email notifying the experiment "32" has finished. The experiment took 9h 53 min 30 seconds without errors.

Figure 78 – Valgrind.

```

edgard@edgard-Aspire-E5-573G: ~/Documents/Projects/Mestrado/Projects/dveb/workspace/build
edgard@edgard-Aspire-E5-573G:~/Documents/Projects/Mestrado/Projects/dveb/workspace/build$
valgrind --leak-check=full --show-leak-kinds=definite ./test -- < root.txt
==7938== Memcheck, a memory error detector
==7938== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==7938== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==7938== Command: ./test --
==7938==
Warning: 'self_cheating' is on
WARNING: Loopback doesn't use UDP checksum! Is it required ???
Warning: 'self_cheating' is on
WARNING: Loopback doesn't use UDP checksum! Is it required ???
2017-Mar-30 09:22:40.581967[40625c0] <warning> Memory threshold is 0%. At least 15% is recommended.
Running 1 test case...

*** No errors detected
==7938==
==7938== HEAP SUMMARY:
==7938==     in use at exit: 186,954 bytes in 577 blocks
==7938==   total heap usage: 84,808 allocs, 84,231 frees, 182,009,873 bytes allocated
==7938==
==7938== LEAK SUMMARY:
==7938==    definitely lost: 0 bytes in 0 blocks
==7938==    indirectly lost: 0 bytes in 0 blocks
==7938==    possibly lost: 3,200 bytes in 10 blocks
==7938==    still reachable: 183,754 bytes in 567 blocks
==7938==                  of which reachable via heuristic:
==7938==                      newarray      : 352 bytes in 1 blocks
==7938==    suppressed: 0 bytes in 0 blocks
==7938== Reachable blocks (those to which a pointer was found) are not shown.
==7938== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==7938==
==7938== For counts of detected and suppressed errors, rerun with: -v
==7938== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
edgard@edgard-Aspire-E5-573G:~/Documents/Projects/Mestrado/Projects/dveb/workspace/build$

```

Valgrind reporting there is no memory leaks.

Figure 79 – Bitbucket git repository.

Author	Commit ID	Message	Date
Edgard Lima	1b3900a	fix compilation	2017-03-19
Edgard Lima	2b474e0	insert optimization	2017-03-19
Edgard Lima	9fb021f	print summary	2017-03-19
Edgard Lima	896e23d	fix help	2017-03-18
Edgard Lima	e6755ab	Now select and successor run each stats collected	2017-03-18
Edgard Lima	7b1f8d2	better statistics	2017-03-18
Edgard Lima	023c2c1	shuffle when testing sequential input	2017-03-18
Edgard Lima	b538bfb	deploy cmd	2017-03-18
Edgard Lima	66241db	property service	2017-03-18
Edgard Lima	6e08462	small fix	2017-03-18
Edgard Lima	b21abba	save several statistics	2017-03-18
Edgard Lima	9a636e2	save statistics in file	2017-03-17
Edgard Lima	333892d	flow control	2017-03-16

Few entries on commit history.

Figure 80 – Statistic.

Method	mean	count	stdev	error	col1	col2	col3	col4
1	303	1	0	0	377	1	0	420
2	1514	1	0	0	1543	1	0	1506
3	2507	1	0	0	2423	2	0	2763
4	3716	2	18	13	4414	3	1130	3312
5	5829	5	2026	906	6091	5	1303	5650
6					5635	5	2039	6239
7					4652	5	494	2585
8					5277	7	1944	1293
9					3418	3	62	
10					4499	4	2247	
11					3504	1	0	
12					2723	1	0	
13					2763	1	0	
14					3312	2	12	
15					3350	5	106	
16					3716	2	18	
17					4414	3	1130	
18					5635	5	2039	
19					4652	5	494	
20					5277	7	1944	
21					3418	3	62	
22					4499	4	2247	
23					3504	1	0	
24					2723	1	0	
25					2763	1	0	
26					3312	2	12	
27					3350	5	106	
28					3716	2	18	
29					4414	3	1130	
30					5635	5	2039	
31					4652	5	494	
32					5277	7	1944	
33					3418	3	62	
34					4499	4	2247	
35					3504	1	0	
36					2723	1	0	
37					2763	1	0	
38					3312	2	12	
39					3350	5	106	
40					3716	2	18	
41					4414	3	1130	
42					5635	5	2039	
43					4652	5	494	
44					5277	7	1944	
45					3418	3	62	
46					4499	4	2247	
47					3504	1	0	
48					2723	1	0	
49					2763	1	0	
50					3312	2	12	
51					3350	5	106	
52					3716	2	18	
53					4414	3	1130	
54					5635	5	2039	
55					4652	5	494	
56					5277	7	1944	
57					3418	3	62	
58					4499	4	2247	
59					3504	1	0	
60					2723	1	0	
61					2763	1	0	
62					3312	2	12	
63					3350	5	106	
64					3716	2	18	
65					4414	3	1130	
66					5635	5	2039	
67					4652	5	494	
68					5277	7	1944	
69					3418	3	62	
70					4499	4	2247	
71					3504	1	0	
72					2723	1	0	
73					2763	1	0	
74					3312	2	12	
75					3350	5	106	
76					3716	2	18	
77					4414	3	1130	
78					5635	5	2039	
79					4652	5	494	
80					5277	7	1944	
81					3418	3	62	
82					4499	4	2247	
83					3504	1	0	
84					2723	1	0	
85					2763	1	0	
86					3312	2	12	
87					3350	5	106	
88					3716	2	18	
89					4414	3	1130	
90					5635	5	2039	
91					4652	5	494	
92					5277	7	1944	
93					3418	3	62	
94					4499	4	2247	
95					3504	1	0	
96					2723	1	0	
97					2763	1	0	
98					3312	2	12	
99					3350	5	106	
100					3716	2	18	

For each method an individual file is saved just ready to be imported.

APPENDIX C – Reproducing the experiments

C.1 General preparation

The steps listed in this subsection apply to all experiments. Specific customizations will be described at the individual experiments subsections.

We will use the machines listed in Table 10 for all experiments. They are connected to a DLINK DGS-1210-28 Gigabit switch ¹ running factory settings. And all the bellow preparations step must be taken in each of them.

Table 10 – Machines

Machine	RAM	CPU	IP	NIC
mustang	4 GB	i7 870 @ 2.93GHz	192.168.1.7	Realtek RTL8111/8168/8411 PCIe GB (rev 03)
camaro	6 GB	i7 940 @ 2.93GHz	192.168.1.15	Realtek RTL8111/8168/8411 PCIe GB (rev 03)
ferrari	16 GB	i7-3770 @ 3.40GHz	192.168.1.3	Realtek RTL8111/8168/8411 PCIe GB (rev 09)
lamborghini	24 GB	i7 X 980 @ 3.33GHz	192.168.1.5	Realtek RTL8111/8168/8411 PCIe GB (rev 03)
bugatti	32 GB	i7-4820K @ 3.70GHz	192.168.1.9	Realtek RTL8111/8168/8411 PCIe GB (rev 09)
maserati	42 GB	i7-4820K @ 3.70GHz	192.168.1.11	Realtek RTL8111/8168/8411 PCIe GB (rev 09)

List of machine used during experiments.

All machines are running Debian 9.0 Stretch and g++ 6.3.0.

First of all, make sure all machines have the latest version of code and compile it.

```
git pull
mkdir build_b && cd build_b && cmake -DCMAKE_BUILD_TYPE=Release
../bigdata && make VERBOSE=1
cd ..
mkdir build_t && cd build_t && cmake -DCMAKE_BUILD_TYPE=Release
../tests && make VERBOSE=1
```

Now, increase kernel's sockets queues and buffers with the following commands:

¹ <https://dlink.com.br/sites/default/files/product_download/dgs-1210-28_c1_datasheet_01hq_pt_01_0.pdf>

```
sudo su -

sysctl -w net.core.wmem_max=134217728
sysctl -w net.core.rmem_max=134217728
sysctl -w net.ipv4.udp_mem=1638400 1638400 1638400
sysctl -w net.core.somaxconn=4096
sysctl -w net.core.netdev_max_backlog=262144
sysctl -w net.core.optmem_max=134217728
sysctl -w net.ipv4.udp_rmem_min=65535
sysctl -w net.ipv4.udp_wmem_min=65535
```

Those values are a probably an overkill. We have set it because at some earlier stage in the development we had problems with packets arriving on the machine and showed up on tcpdump, but not read on the process. That is because the default socket is very small for our needs. We are pretty much flooding the machines with 48K bytes UDP packets, then the socket buffer attached to the process gets full and start dropping packages. On later versions of our implementation we developed a congestion/flow control on our protocol, probably the default values are still small for our needs but we don't probably need such overkill. Further analysis on this will be left for future research with protocols.

Now, stop the linux Graphical service to release extra memory and cpu, and disable memory swapping to avoid another big source of uncertainty. See the commands in the grayed box bellow.

```
sudo su -

/etc/init.d/gdm stop; /etc/init.d/lightdm stop; /etc/init.d/gdm3 stop;
/etc/init.d/x11-common stop

swapoff -a && sh -c sync && sh -c 'echo 3 > /proc/sys/vm/drop_caches'
```

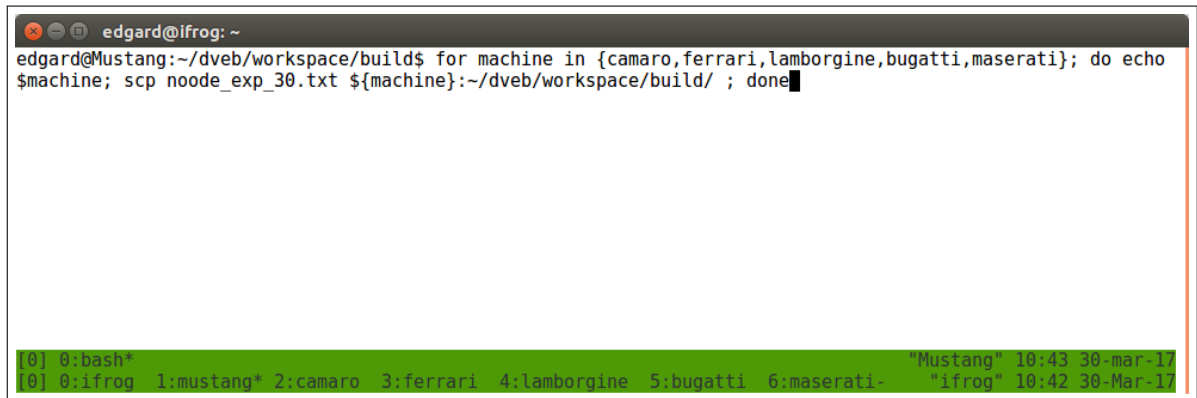
For all experiment there are configurations files to be used. One way to edit those files other than copy paste from here is;

```
./test help > name.conf
```

Then, edit the generated file according and execute with:

```
./test -- < name.conf
# or, to save the output
./test -- < name.conf 2 > &1 | tee result.vt100
```

Additionally we also recommend using tmux on all machines like in Figure 81 because the chance to drop your ssh connection is high due to intensive network traffic.



```
edgard@ifrog: ~
edgard@Mustang:~/dveb/workspace/build$ for machine in {camaro,ferrari,lamborghini,bugatti,maserati}; do echo
$machine; scp noode_exp_30.txt ${machine}:/dveb/workspace/build/ ; done
[0] 0:bash* "Mustang" 10:43 30-mar-17
[0] 0:ifrog 1:mustang* 2:camaro 3:ferrari 4:lamborghini 5:bugatti 6:maserati- "ifrog" 10:42 30-Mar-17
```

Figure 81 – Tmux session.

Table 11 – Machines/Configuration

Machine	Configuration
mustang	root.conf
camaro	node.conf
ferrari	node.conf
lamborghini	node.conf
bugatti	node.conf
maserati	cheater.conf

Configuration for each machine.

C.2 Experiment 01

```
root.conf

port 34343
multicast_group "225.0.0.37"
role "root"
timeout 4
jump_multi 12
retries 8
enough_servers_available 1
memory_threshold 20
udp_buf_size 33792
thread_pool_size 12
log_level "info"
root_uuid 0
force_maxsize true
statistics {
    summary_only false
} multicast_loopback false
no_hosting true
self_cheating false
tests {
    run true
    maxbits 16
    mode "performance"
}
has_cheater true
service false
```


node.conf

```
port 34343
multicast_group "225.0.0.37"
role "node"
timeout 4
jump_multi 12
retries 8
enough_servers_available 1
memory_threshold 20
udp_buf_size 33792
thread_pool_size 12
log_level "info"
root_uuid 0
force_maxsize true
statistics {
    summary_only false
} multicast_loopback false
no_hosting false
self_cheating false
test {
    run false
    maxbits 16
    mode "performance"
}
has_cheater true
service true
```

```
cheater.conf

port 34343
multicast_group "225.0.0.37"
role "cheater"
timeout 4
jump_multi 12
retries 8
enough_servers_available 1
memory_threshold 20
udp_buf_size 33792
thread_pool_size 12
log_level "info"
root_uuid 0
force_maxsize true
statistics {
    summary_only false
} multicast_loopback false
no_hosting true
self_cheating false
tests {
    run false
    maxbits 16
    mode "performance"
}
has_cheater true
service true
```

C.3 Experiment 02

For this experiment use the configuration files of [Experiment 01](#) just change the value of "tests.maxbits" to 131072:

```
root.conf

test {
    run true
    maxbits 131072
    mode "performance"
}
```

node.conf

```
tests {  
    run false  
    maxbits 131072  
    mode "performance"  
}
```

cheater.conf

```
test {  
    run false  
    maxbits 131072  
    mode "performance"  
}
```

C.4 Correctness test

Use the same configuration files from [Experiment 01](#) except for the value of "test.mode" to "correctness"

Repeat the test use the same configuration files from [Experiment 02](#) except for the value of "test.mode" to "correctness"

root.conf

```
test {  
    run true  
    maxbits 16  
    mode "correctness"  
}
```

node.conf

```
test {  
    run false  
    maxbits 16  
    mode "correctness"  
}
```

cheater.conf

```
test {  
  run false  
  maxbits 16  
  mode "correctness"  
}
```

APPENDIX D – Source Code

id.cc

```
bool Id::is_summary(const Id & id, bool check_parents) {
    uint32_t level = id.y_ >> 27;
    uint32_t smask = id.y_ & 0x7FFFFFFF;

    if (check_parents) {
        return smask != 0;
    }

    return ((1u << (level - 1u)) & smask);
}

Id Id::calculate_id(bitscnt_t universe) {
    Id id;
    uint32_t level = bigdata::log2(universe) + 1;
    uint32_t smask = 0x00;

    id.y_ = (level << 27) | smask;

    return id;
}

Id Id::calculate_child_id(const Id & parent, bool summary
↪ , const Natural & index) {
    Id id;
    uint32_t parent_level = parent.y_ >> 27;
    uint32_t parent_smask = parent.y_ & 0x7FFFFFFF;

    if (parent_level == 1) {
        throw std::runtime_error("Parent is already 2^1.
↪ You are trying to calculate the child's Id!
↪ ");
    }
}
```

```

uint32_t level = parent_level - 1;
uint32_t smask = parent_smask;

if (summary) {
    smask |= 1 << (level - 1);
    id.id_ = parent.id_;
} else {
    id.id_ = parent.id_ | (index << (1u << (level -
    ↪ 1)));
}

id.y_ = (level << 27) | smask;

return id;
}

```

veb_root.cc

```

void VebRoot::expand(const Natural & key) {
    while (key.bits() > universe_) {
        auto start = std::chrono::steady_clock::now();
        auto sample = statistic::include_thread_sample();
        auto aux_universe = universe_;
        auto time_couter = std::shared_ptr<void>(NULL, [&
        ↪ start,&sample, aux_universe](void*){
            if (sample != NULL && false == std::
            ↪ uncaught_exception()) {
                auto duration = std::chrono::
                ↪ duration_cast<std::chrono::
                ↪ microseconds>(std::chrono::
                ↪ steady_clock::now() - start);

                sample->normalize_level(aux_universe);
                sample->duration = duration.count();
                statistic::Statistic::add(*sample,
                ↪ TreeRpc::kVeb_expanded);
            }
        });
    }
}

```

```

        bitscnt_t new_universe = (universe_ > 0) ?
            ↪ universe_ << 1 : 1;
        std::shared_ptr<IVebGlobal> new_veb;
        auto temp_id = std::make_shared<Id>(Id::
            ↪ calculate_id(new_universe));
        auto newtuple = factory_->create(uuid_,
            ↪ new_universe, temp_id, server_, Factory::
            ↪ kTreeAuto, Factory::kInstanceClient);

        new_veb = std::dynamic_pointer_cast<IVebGlobal>(
            ↪ std::get<2>(newtuple));
        new_veb->search(Natural(0)); // make sure is
            ↪ created on peer

        if (veb_ != NULL) {
            new_veb->expanded(veb_);
        }

        if (std::get<1>(newtuple) == Factory::
            ↪ kInstanceLocal) {
            registry_->sign(uuid_ , *new_veb->id(),
                ↪ new_veb);
        }

        veb_.swap(new_veb);
        universe_ = new_universe;
        std::swap(id_, temp_id);
    }
}

std::pair<bool, std::shared_ptr<ISerializable>> VebRoot::
    ↪ insert(const Natural & key, std::shared_ptr<
    ↪ ISerializable> value) {

    if (!key.is_valid()) return {};
    if (veb_ == NULL || key.bits() > universe_) {
        expand(key);
    }
}

```

```

auto start = std::chrono::steady_clock::now();
auto sample = statistic::include_thread_sample();
auto time_couter = std::shared_ptr<void>(NULL, [this
    ↪ ,&start,&sample](void*){
    if (sample != NULL && false == std::
        ↪ uncaught_exception()) {
        auto duration = std::chrono::duration_cast<
            ↪ std::chrono::microseconds>(std::chrono
            ↪ ::steady_clock::now() - start);

        sample->normalize_level(universe_);
        sample->duration = duration.count();
        statistic::Statistic::add(*sample, TreeRpc::
            ↪ kInsert);
    }
});

return veb_->insert(key, value);
}

```

factory.cc

```

std::tuple<Factory::TreeType, Factory::TreeInstanceType,
    ↪ std::shared_ptr<ITree> >
Factory::create(tree_uuid_t uuid, bitscnt_t universe, std
    ↪ ::shared_ptr<const Id> id, std::shared_ptr<Server>
    ↪ server, TreeType tree_type, TreeInstanceType
    ↪ instance_type) {
    if (tree_type == kTreeAuto) {
        tree_type = kTreeVeb;
    }
    if (instance_type == kInstanceAuto) {
        instance_type = kInstanceClient;
    }
    if (instance_type == kInstanceClient) {
        auto tree_rpc = std::make_shared<TreeRpcMulticast
            ↪ >(universe, server, uuid, id, false);
    }
}

```



```

    auto new_tree = std::make_shared<veb::VebClient>(
        ↪ universe, id, tree_rpc);
    return std::make_tuple(tree_type, instance_type,
        ↪ new_tree);
} else if (instance_type == kInstanceRemote) {
    if (universe == 1) {
        auto summary = Id::is_summary(*id);
        if (summary) {
            return std::make_tuple(tree_type,
                ↪ instance_type, std::make_shared<veb
                ↪ ::VebU1SummaryLocal>(universe, id))
                ↪ ;
        } else {
            return std::make_tuple(tree_type,
                ↪ instance_type, std::make_shared<veb
                ↪ ::VebU1Local>(universe, id));
        }
    } else {
        auto new_tree = std::make_shared<veb::
            ↪ VebRemote>(universe, id, uuid, server,
            ↪ this->shared_from_this());
        return std::make_tuple(tree_type,
            ↪ instance_type, new_tree);
    }
} else if (instance_type == kInstanceLocal) {
    auto summary = Id::is_summary(*id);
    if (summary) {
        if (universe == 1) {
            return std::make_tuple(tree_type,
                ↪ instance_type, std::make_shared<veb
                ↪ ::VebU1SummaryLocal>(universe, id))
                ↪ ;
        } else {
            return std::make_tuple(tree_type,
                ↪ instance_type, std::make_shared<veb
                ↪ ::VebSummaryLocal>(universe, id));
        }
    } else {

```

```

        if (universe == 1) {
            return std::make_tuple(tree_type,
                                   ↪ instance_type, std::make_shared<veb
                                   ↪ ::VebU1Local>(universe, id));
        } else {
            return std::make_tuple(tree_type,
                                   ↪ instance_type, std::make_shared<veb
                                   ↪ ::VebLocal>(universe, id));
        }
    }
}
return {};
}

```

statistic.hh

```

struct Sample : public ISerializable {
    // ...
    void zero() {
        duration = 0;
        rmsg_all = 0;
        rmsg_timeout = 0;
        deeper_level = UINT32_MAX;
    }

    void update(const Sample & sample) {
        rmsg_all += sample.rmsg_all;
        rmsg_timeout += sample.rmsg_timeout;
        if (sample.deeper_level < deeper_level) {
            deeper_level = sample.deeper_level;
        }
    }

    void set_level(bitscnt_t universe) {
        if (universe < deeper_level) {
            deeper_level = universe;
        }
    }
}

```

```

    void normalize_level(bitscnt_t root_universe) {
        auto base_level = bigdata::log2(root_universe);
        deeper_level = bigdata::log2(deeper_level);
        if (base_level >= deeper_level) {
            deeper_level = base_level - deeper_level;
        }
    }
    // ...
};

```

statistic.cc

```

static __thread Sample * temp_sample_ = NULL;

std::shared_ptr<Sample> include_thread_sample() {
    if (temp_sample_ != NULL) {
        return get_thread_sample();
    }
    temp_sample_ = new Sample();
    return std::shared_ptr<Sample>(temp_sample_, [](
        ↪ Sample*)
    {
        delete temp_sample_;
        temp_sample_ = NULL;
    });
}

std::shared_ptr<Sample> get_thread_sample() {
    if (temp_sample_ != NULL) {
        return std::shared_ptr<Sample>(temp_sample_, [](
            ↪ Sample*){}));
    }
    return nullptr;
}

void thread_sample_set_level(bitscnt_t universe) {
    auto sample = get_thread_sample();
}

```

```

        if (sample != NULL) {
            sample->set_level(universe);
        }
    }

    void thread_sample_update(const Sample & sample) {
        auto s = get_thread_sample();
        if (s != NULL) {
            s->update(sample);
        }
    }

    std::unordered_map<pthread_t, Statistic> Statistic::
        ↪ registry_;
    std::mutex Statistic::registry_mutex_;

```

tree_rpc_multicast.cc

```

1 std::tuple<bool, Natural, std::shared_ptr<ISerializable>,
    ↪ bitscnt_t>
2 TreeRpcMulticast::run_method(TreeRpc::Method method,
    ↪ TreeRpc::Method answer, const Natural * key, const
    ↪ ISerializable * value, const Id * expand_id, std::
    ↪ shared_ptr<const AddressInfo> unicast_addr) const {
3     volatile bool ack_received = false;
4     volatile bool got_exception = false;
5     volatile bool got_non_exist = false;
6     volatile bool got_answer = false;
7     std::tuple<bool, Natural, std::shared_ptr<
    ↪ ISerializable>, bitscnt_t> ret_value;
8     std::condition_variable cond;
9     std::mutex mtwait;
10    auto sample = statistic::get_thread_sample();
11
12    {
13        std::shared_ptr<ObserverHandle<TreeRpc::Method>>
            ↪ observer;
14        std::shared_ptr<ObserverHandle<TreeRpc::Method>>

```

```

    ↪ observer_ack;
15    std::shared_ptr<ObserverHandle<TreeRpc::Method>>
    ↪ observer_nack;
16
17    observer = server_->add(answer, [this, &mtwait, &
    ↪ cond, &ret_value, &got_answer, &sample](const
    ↪ TreeRpc::Method &method, const
    ↪ ServerDispatchArgs & args) -> bool {
18        std::unique_lock<std::mutex> lkwait(mtwait);
19        bool handled = false;
20        if (method == TreeRpc::kControlForceStop) {
21            cond.notify_all();
22            return false;
23        }
24        try {
25            if (transaction_ == args.transaction) {
26                handled = true;
27                ret_value = std::make_tuple(args.has,
    ↪ std::move(args.key), nullptr,
    ↪ args.universe);
28                if (sample != NULL && args.sample !=
    ↪ NULL) {
29                    sample->update(*args.sample);
30                }
31                got_answer = true;
32                cond.notify_all();
33
34                // TODO: put on thread pool ?
35                BOOST_LOG_TRIVIAL(trace) << "Sending_
    ↪ method_" << TreeRpc::kAckF << "
    ↪ _with_transaction_" << args.
    ↪ transaction;
36                this->reply(args.uuid, args.addr,
    ↪ TreeRpc::kAckF, args.
    ↪ transaction, true, nullptr,
    ↪ nullptr, nullptr);
37            }
38        } catch(const std::runtime_error & e) {

```

```

39         std::cerr << e.what() << std::endl;
40     } catch(...) {
41     }
42     return handled;
43 });
44
45 observer_ack = server_->add(kAck, [this,&mtwait,&
    ↪ cond,&ack_received](const TreeRpc::Method &
    ↪ method, const ServerDispatchArgs & args) ->
    ↪ bool {
46     std::unique_lock<std::mutex> lkwait(mtwait);
47     bool handled = false;
48     if (method == TreeRpc::kControlForceStop) {
49         cond.notify_all();
50         return false;
51     }
52     try {
53         if (transaction_ == args.transaction) {
54             if (false == ack_received) {
55                 cond.notify_all();
56             }
57             handled = true;
58             ack_received = true;
59         }
60     } catch(const std::runtime_error & e) {
61         std::cerr << e.what() << std::endl;
62     } catch(...) {
63     }
64     return handled;
65 });
66
67
68 observer_nack = server_->add(kNack, [this,&mtwait
    ↪ ,&cond,&got_exception,&got_non_exist](const
    ↪ TreeRpc::Method &method, const
    ↪ ServerDispatchArgs & args) -> bool {
69     std::unique_lock<std::mutex> lkwait(mtwait);
70     bool handled = false;

```

```

71         if (method == TreeRpc::kControlForceStop) {
72             cond.notify_all();
73             return false;
74         }
75         try {
76             if (transaction_ == args.transaction) {
77                 handled = true;
78                 got_exception = args.has;
79                 got_non_exist = !got_exception;
80                 cond.notify_all();
81                 if (got_exception) {
82                     BOOST_LOG_TRIVIAL(warning) << "
                        ↪ Got_a_Nack_exception";
83                 } else {
84                     BOOST_LOG_TRIVIAL(trace) << "Got_
                        ↪ a_Nack_not_exist";
85                 }
86             }
87         } catch(const std::runtime_error & e) {
88             std::cerr << e.what() << std::endl;
89         } catch(...) {
90             }
91         return handled;
92     });
93
94     {
95         std::unique_lock<std::mutex> lkwait(mtwait,
            ↪ std::defer_lock);
96         std::unique_lock<std::recursive_mutex>
            ↪ lk_channel((server_>channel())>mutex
            ↪ (), std::defer_lock);
97
98         lkwait.lock();
99
100        for(auto retries=0; retries <= Properties::
            ↪ get_instance()->retries(); retries++) {
101            lkwait.unlock();
102            lk_channel.lock();

```

```

103         process(method, retries == 0, key, value,
           ↪ expand_id);
104 BOOST_LOG_TRIVIAL(trace) << "Sending_
           ↪ method_" << method << "_with_"
           ↪ transaction_" << transaction_ << "_"
           ↪ retry_" << retries;
105 (server_>channel())>sync(unicast_addr);
106 lk_channel.unlock();
107 lkwait.lock();
108
109 if (sample != NULL) {
110     sample->rmsg_all++;
111 }
112
113 if (got_non_exist == true) {
114     break;
115 }
116
117 if (got_exception) {
118     BOOST_LOG_TRIVIAL(error) << "Universe
           ↪ _" << universe_ << "_Id_" <<
           ↪ id_ << "_Transaction_" <<
           ↪ transaction_ << "_Method_" <<
           ↪ method << "_peer_Nack_exception
           ↪ ";
119     throw std::runtime_error("Peer got an
           ↪ _exception");
120 }
121 if (got_non_exist) {
122     BOOST_LOG_TRIVIAL(trace) << "Universe
           ↪ _" << universe_ << "_Id_" <<
           ↪ id_ << "_Transaction_" <<
           ↪ transaction_ << "_Method_" <<
           ↪ method << "_peer_Nack_doesnt_"
           ↪ exists";
123     throw exception_peer_not_exist();
124 }
125

```



```

126         if (ack_received == false) {
127             if (retries > 0) {
128                 if (sample != NULL) {
129                     sample->rmsg_timeout++;
130                 }
131             }
132             BOOST_LOG_TRIVIAL(trace) << "Universe
↪ _" << universe_ << "_Id_" <<
↪ id_ << "_Transaction_" <<
↪ transaction_ << "_Method_" <<
↪ method << "_waiting_for_ack";
133             cond.wait_for(lkwait, std::chrono::
↪ milliseconds(Properties::
↪ get_instance()->timeout()));
134             if (server_->exiting()) {
135                 throw std::runtime_error("Server_
↪ is_exiting");
136             }
137         } else {
138             break;
139         }
140
141         if (got_non_exist == true) {
142             break;
143         }
144
145         if (got_exception) {
146             BOOST_LOG_TRIVIAL(error) << "Universe
↪ _" << universe_ << "_Id_" <<
↪ id_ << "_Transaction_" <<
↪ transaction_ << "_Method_" <<
↪ method << "_peer_Nack_exception
↪ ";
147             throw std::runtime_error("Peer_got_an
↪ _exception");
148         }
149         if (got_non_exist) {
150             BOOST_LOG_TRIVIAL(trace) << "Universe

```

```

        ↪ " " << universe_ << "Id" <<
        ↪ id_ << "Transaction" <<
        ↪ transaction_ << "Method" <<
        ↪ method << "peerNackdoesnt"
        ↪ exists";
151     throw exception_peer_not_exist();
152 }
153 if (ack_received) {
154     break;
155 }
156 }

157
158 if (ack_received == false) {
159     BOOST_LOG_TRIVIAL(trace) << "Universe"
        ↪ << universe_ << "Id" << id_ << "
        ↪ Transaction" << transaction_ << "
        ↪ Method" << method << "gotnoack"
        ↪ ;
160     throw exception_peer_timedout("Wegotno
        ↪ Ack");
161 }
162
163 lkwait.unlock();
164 observer_ack.reset();
165 lkwait.lock();
166
167
168 if (got_answer == false) {
169     std::cv_status waitres;
170
171     if (got_exception) {
172         BOOST_LOG_TRIVIAL(error) << "Universe
            ↪ " << universe_ << "Id" <<
            ↪ id_ << "Transaction" <<
            ↪ transaction_ << "Method" <<
            ↪ method << "peerNackexception
            ↪ aftertheack";
173         throw std::runtime_error("Peergotan

```

```

    ↪ "exception");
174 }
175 if (got_non_exist) {
176     BOOST_LOG_TRIVIAL(error) << "Universe
        ↪ " << universe_ << "Id_" <<
        ↪ id_ << "Transaction_" <<
        ↪ transaction_ << "Method_" <<
        ↪ method << "peer_Nack_doesnt_
        ↪ exists_after_the_ack";
177     throw std::runtime_error("Weird.
        ↪ Peer_sent_ack_but_answer_
        ↪ Actually_we_got_'got_non_exist'
        ↪ after_an_Ack");
178 }
179
180 uint32_t expected_jumps = universe_ > 1 ?
    ↪ log2(universe_) : 1;
181 expected_jumps *= Properties::
    ↪ get_instance()->jump_multi(); //
    ↪ let's assume there might be a
    ↪ missing at each level
182 uint32_t timeout = Properties::
    ↪ get_instance()->timeout() *
    ↪ expected_jumps;
183 timeout *= Properties::get_instance()->
    ↪ retries() + 1;
184
185 BOOST_LOG_TRIVIAL(trace) << "Universe_"
    ↪ << universe_ << "Id_" << id_ << "
    ↪ Transaction_" << transaction_ << "
    ↪ Method_" << method << "got_ack_but
    ↪ result_Let's_wait_for_" <<
    ↪ timeout << "ms";
186 waitres = cond.wait_for(lkwait, std::
    ↪ chrono::milliseconds(timeout));
187 if (server_->exiting()) {
188     throw std::runtime_error("Server_is_
        ↪ exiting");

```

```

189     }
190
191     if (got_answer == false) {
192         BOOST_LOG_TRIVIAL(error) << "Universe
            ↳ " << universe_ << "Id" <<
            ↳ id_ << "Transaction" <<
            ↳ transaction_ << "Method" <<
            ↳ method << "longwaitfinished_
            ↳ without_answer_and_with" << (
            ↳ std::cv_status::timeout ==
            ↳ waitres ? "timeout" : "result")
            ↳ ;
193     if (got_exception) {
194         BOOST_LOG_TRIVIAL(error) << "
            ↳ Universe" << universe_ <<
            ↳ "Id" << id_ << "
            ↳ Transaction" <<
            ↳ transaction_ << "Method"
            ↳ << method << "peerNack
            ↳ exceptionafterlongwait";
195         throw std::runtime_error("Peer_
            ↳ got_an_exception");
196     }
197     if (got_non_exist) {
198         BOOST_LOG_TRIVIAL(error) << "
            ↳ Universe" << universe_ <<
            ↳ "Id" << id_ << "
            ↳ Transaction" <<
            ↳ transaction_ << "Method"
            ↳ << method << "peerNack
            ↳ doesnt_exists_after_long_
            ↳ wait";
199         throw std::runtime_error("Weird.
            ↳ Peer_sent_ack_but_answer_
            ↳ Actually_we_got_
            ↳ got_non_exist'after_anAck
            ↳ and_longwait");
200     }

```

```

201         BOOST_LOG_TRIVIAL(error) << "Universe
        ↳ _" << universe_ << "_Id_" <<
        ↳ id_ << "_Transaction_" <<
        ↳ transaction_ << "_Method_" <<
        ↳ method << "_after_long_wait,_"
        ↳ got_nothing";
202         throw std::runtime_error("Peer_ack_
        ↳ and_then_not_responded");
203     }
204 }
205 }
206 }
207
208     return ret_value;
209 }

```

flowcontrol.cc

```

1 namespace bigdata {
2
3 static volatile int32_t last_step_ = 0;
4 static int32_t consecutive_ok_ = 0;
5 static std::mutex mutex_;
6
7 static std::mt19937 generator_;
8
9 uint32_t FlowControl::timeout() {
10     std::lock_guard<std::mutex> lkg(mutex_);
11     static auto initied = false;
12     if (!initied) {
13         struct timeval tv;
14         gettimeofday(&tv, NULL);
15         generator_.seed(getpid() ^ tv.tv_usec);
16
17         initied = true;
18     }
19
20     uint32_t timeout = Properties::get_instance()->

```

```

    ↪ timeout();
21
22     if (last_step_ > 0) {
23         timeout *= 1 << last_step_;
24         uint32_t val = generator_() % (((timeout / 2) +
    ↪ 1) | 1) ;
25         timeout += val;
26     }
27
28     return timeout;
29 }
30
31 void FlowControl::timeout_feedback(bool timeout) {
32     std::lock_guard<std::mutex> lkg(mutex_);
33
34     if (timeout) {
35         if (last_step_ < Properties::get_instance()->
    ↪ retries()) {
36             last_step_++;
37             BOOST_LOG_TRIVIAL(trace) << "last_step_␣
    ↪ incremented␣to␣" << last_step_;
38         }
39         consecutive_ok_ = 0;
40     } else {
41         consecutive_ok_++;
42         if (consecutive_ok_ >= 4) {
43             consecutive_ok_ = 4;
44             if (last_step_ > 0) {
45                 last_step_--;
46                 BOOST_LOG_TRIVIAL(trace) << "last_step_␣
    ↪ decremented␣to␣" << last_step_;
47                 consecutive_ok_ = 0;
48             }
49         }
50     }
51 }
52
53 uint32_t FlowControl::long_timeout(bitscnt_t universe) {

```

```

54     uint32_t expected_jumps = universe > 1 ? log2(
        ↪ universe) : 1;
55     expected_jumps *= Properties::get_instance()->
        ↪ jump_multi(); // let's assume there might be a
        ↪ missing at each level
56
57     auto max_step = 1 << Properties::get_instance()->
        ↪ retries();
58     uint32_t timeout = Properties::get_instance()->
        ↪ timeout() * max_step;
59     timeout += ((timeout / 2) + 1) | 1;
60     timeout *= Properties::get_instance()->retries()+1;
61     timeout *= expected_jumps;
62
63     return timeout;
64 }
65
66 }

```

test.cc

```

1 template <class T>
2 static void test_sanity(std::vector<T> & tdata) {
3     int32_t i=0;
4     const int32_t print_each = 100;
5
6     BOOST_LOG_TRIVIAL(info) << "Correctness_ test";
7
8     for (auto v : tdata) {
9         if (quitting_) return;
10        if ((i++ % print_each) == print_each - 1) {
11            std::cout << std::endl;
12            BOOST_LOG_TRIVIAL(info) << "inserting_" << i
                ↪ << '/' << tdata.size();
13        } else {
14            std::cout << '.' << std::flush;
15        }
16        auto h = std::get<0>(root_->insert(bigdata::

```

```

    ↪ Natural(v), {}));
17     BOOST_CHECK_MESSAGE(h == false, v.to_string() + "
    ↪ ␣(insert)found");
18 }
19 std::cout << std::endl;
20
21 std::sort(tdata.begin(), tdata.end());
22
23 BOOST_LOG_TRIVIAL(info) << "search";
24
25 for (auto v=0u; v < tdata.size(); v++) {
26     if (quitting_) return;
27     if (v % print_each == print_each - 1) {
28         std::cout << std::endl;
29         BOOST_LOG_TRIVIAL(info) << "search␣" << v <<
    ↪ ' / ' << tdata.size();
30     } else {
31         std::cout << ' . ' << std::flush;
32     }
33     auto k = root_->search(bigdata::Natural(tdata[v])
    ↪ ).first;
34     BOOST_CHECK_MESSAGE(k == true, tdata[v].to_string
    ↪ () + "␣not␣found");
35 }
36 std::cout << std::endl;
37
38 BOOST_CHECK_MESSAGE(root_->min().first == tdata[0],
    ↪ root_->min().first.to_string() + "␣(min)!=" +
    ↪ tdata[0].to_string());
39 BOOST_CHECK_MESSAGE(root_->max().first == tdata[tdata
    ↪ .size()-1], root_->max().first.to_string() + "␣
    ↪ (max)!=" + tdata[tdata.size()-1].to_string());
40
41 BOOST_LOG_TRIVIAL(info) << "predecessor";
42
43 for (auto v=1u; v < tdata.size(); v++) {
44     if (quitting_) return;
45     if (v % print_each == print_each - 1) {

```



```

46         std::cout << std::endl;
47         BOOST_LOG_TRIVIAL(info) << "predecessor_" <<
           ↪ v << '/' << tdata.size();
48     } else {
49         std::cout << '.' << std::flush;
50     }
51     bigdata::Natural p = root_->predecessor(bigdata::
           ↪ Natural(tdata[v])).first;
52     BOOST_CHECK_MESSAGE(p == tdata[v-1], p.to_string
           ↪ () + "_("predecessor)!=_" + tdata[v-1].
           ↪ to_string());
53 }
54 std::cout << std::endl;
55
56 BOOST_LOG_TRIVIAL(info) << "successor";
57
58 for (auto v=0u; v < tdata.size()-1; v++) {
59     if (quitting_) return;
60     if (v % print_each == print_each - 1) {
61         std::cout << std::endl;
62         BOOST_LOG_TRIVIAL(info) << "successor_" << v
           ↪ << '/' << tdata.size();
63     } else {
64         std::cout << '.' << std::flush;
65     }
66     bigdata::Natural s = root_->successor(bigdata::
           ↪ Natural(tdata[v])).first;
67     BOOST_CHECK_MESSAGE(s == tdata[v+1], s.to_string
           ↪ () + "_("successor)!=_" + tdata[v+1].
           ↪ to_string());
68 }
69 std::cout << std::endl;
70
71 BOOST_LOG_TRIVIAL(info) << "deleting/searching";
72
73 std::random_shuffle(tdata.begin(), tdata.end());
74
75 for (auto v=0u; v < tdata.size(); v++) {

```

```

76     if (quitting_) return;
77     if (v % print_each == print_each - 1) {
78         std::cout << std::endl;
79         BOOST_LOG_TRIVIAL(info) << "deleting/
           ↳ searching_" << v << '/' << tdata.size()
           ↳ ;
80     } else {
81         std::cout << '.' << std::flush;
82     }
83     root_->remove(bigdata::Natural(tdata[v]));
84     auto k = root_->search(bigdata::Natural(tdata[v])
           ↳ ).first;
85     if (k) {
86         root_->remove(bigdata::Natural(tdata[v]));
87         k = root_->search(bigdata::Natural(tdata[v])
           ↳ ).first;
88     }
89     BOOST_CHECK_MESSAGE(k == false, tdata[v].
           ↳ to_string() + " not found");
90 }
91 std::cout << std::endl;
92
93 BOOST_CHECK_MESSAGE(root_->min().first.is_valid() ==
           ↳ false, root_->min().first.to_string() + " (min)
           ↳ not empty");
94 BOOST_CHECK_MESSAGE(root_->max().first.is_valid() ==
           ↳ false, root_->max().first.to_string() + " (max)
           ↳ not empty");
95 }
96
97 static void test_performance(bigdata::TreeRpc::Method
           ↳ method, const std::unordered_set<bigdata::Natural>
           ↳ & keys, int32_t index, const int32_t
           ↳ num_collect_statistic) {
98     const int print_header_each = 100;
99
100     BOOST_LOG_TRIVIAL(info) << "Performance test";
101

```

```

102     auto it = std::next(keys.begin(), index);
103
104     for(auto count=0; count < num_collect_statistic && it
105         ↪ != keys.end(); it++, count++) {
106         while(paused_) {
107             if (quitting_) return;
108             usleep(100 * 1000);
109         }
110         if (quitting_) return;
111
112         if (std::distance(keys.begin(), it) %
113             ↪ print_header_each == 0) {
114             std::cout << std::endl;
115             BOOST_LOG_TRIVIAL(info) << method << "□-□" <<
116                 ↪ std::distance(keys.begin(), it) + 1 <<
117                 ↪ "/" << keys.size();
118         } else {
119             std::cout << ' ' << std::flush;
120         }
121         BOOST_LOG_TRIVIAL(trace) << "Key□under□test□(" <<
122             ↪ method << "□" << *it;
123
124         switch (method) {
125         case bigdata::TreeRpc::kInsert:
126             root_>insert(*it, {});
127             break;
128         case bigdata::TreeRpc::kSearch:
129             root_>search(*it);
130             break;
131         case bigdata::TreeRpc::kSuccessor:
132             root_>successor(*it);
133             break;
134         case bigdata::TreeRpc::kPredecessor:
135             root_>predecessor(*it);
136             break;
137         case bigdata::TreeRpc::kRemove:
138             root_>remove(*it);
139             break;
140         }
141     }

```

```

135         default:
136             throw std::runtime_error("Not testing method
137                                     ↪ " + bigdata::TreeRpc::to_string(method)
138                                     ↪ );
139     }
140 }
141
142 std::cout << std::endl;
143 BOOST_LOG_TRIVIAL(info) << method << " From " <<
144     ↪ index + 1 << " to " << std::distance(keys.begin
145     ↪ (), it);
146 }
147
148 static void test() {
149     bigdata::statistic::Statistic::include();
150     std::unordered_set<bigdata::Natural> keys;
151     const int32_t numinsert = 65535; // UINT64_MAX;
152     const int32_t num_collect_statistic = 2000;
153     BOOST_LOG_TRIVIAL(debug) << "
154     ↪ test_tree_correctness_big_no_check";
155
156     bigdata::Natural::Seed seed;
157     int32_t i = 0;
158
159     try {
160         while(keys.size() < numinsert) {
161             if (quitting_) return;
162             keys.insert(bigdata::Natural::random(seed,
163                 ↪ bigdata::Properties::get_instance()->
164                 ↪ test_maxbits()));
165         }
166
167         BOOST_LOG_TRIVIAL(info) << "preparing input data
168         ↪ ...";
169
170         if (bigdata::Properties::get_instance()->
171             ↪ test_mode() == "correctness") {
172             std::vector<bigdata::Natural> correctness(

```

```

        ↪ keys.size());
164     std::copy(keys.begin(), keys.end(),
        ↪ correctness.begin());
165     BOOST_LOG_TRIVIAL(info) << "input_data_ready"
        ↪ ;
166     test_sanity(correctness);
167     return;
168 }
169
170 BOOST_LOG_TRIVIAL(info) << "input_data_ready";
171
172 for(int32_t i=0, s=0; i < static_cast<int32_t>(
    ↪ keys.size()); i += num_collect_statistic, s
    ↪ ++) {
173     int32_t n = num_collect_statistic;
174     if (n > static_cast<int32_t>(keys.size()) - i
        ↪ ) {
175         n = static_cast<int32_t>(keys.size()) - i
            ↪ ;
176     }
177     if (n > 0) {
178         test_performance(bigdata::TreeRpc::
            ↪ kInsert, keys, i, n);
179         test_performance(bigdata::TreeRpc::
            ↪ kSuccessor, keys, i, n);
180         test_performance(bigdata::TreeRpc::
            ↪ kPredecessor, keys, i, n);
181         test_performance(bigdata::TreeRpc::
            ↪ kSearch, keys, i, n);
182
183         std::cout << bigdata::statistic::
            ↪ Statistic::get();
184         bigdata::statistic::Statistic::get().save
            ↪ (s+1);
185         bigdata::statistic::Statistic::zero();
186     }
187 }
188

```

```

189     for(int32_t i=0, s=0; i < static_cast<int32_t>(
        ↪ keys.size()); i += num_collect_statistic, s
        ↪ ++) {
190         int32_t n = num_collect_statistic;
191         if (n > static_cast<int32_t>(keys.size()) - i
        ↪ ) {
192             n = static_cast<int32_t>(keys.size()) - i
        ↪ ;
193         }
194         if (n > 0) {
195             test_performance(bigdata::TreeRpc::
        ↪ kRemove, keys, i,
        ↪ num_collect_statistic);
196
197             std::cout << bigdata::statistic::
        ↪ Statistic::get();
198             bigdata::statistic::Statistic::get().save
        ↪ (s+1);
199             bigdata::statistic::Statistic::zero();
200         }
201     }
202
203 } catch (const std::runtime_error & e) {
204     BOOST_LOG_TRIVIAL(fatal) << "Except_with_i=" <<
        ↪ i << "." << e.what();
205     throw e;
206 } catch (...) {
207     BOOST_LOG_TRIVIAL(fatal) << "Except_with_i=" <<
        ↪ i << ".";
208     throw;
209 }
210 }

```

natural.hh

```

1 namespace std
2 {
3     template<>

```

```

4     struct hash<bigdata::Natural>
5     {
6         size_t operator()(bigdata::Natural const& natural
            ↪ ) const
7         {
8             size_t hash = 0x00;
9             if (natural.value_ != 0) {
10                size_t count = mpz_size(natural.value_ ->
                    ↪ n_);
11                const mp_limb_t * limbs = mpz_limbs_read
                    ↪ (natural.value_ -> n_);
12                for (auto c=0u; c < count; c++) {
13                    uint64_t limb = limbs[c];
14                    boost::hash_combine(hash, limb & 0
                        ↪ 0xFFFFFFFF);
15                    boost::hash_combine(hash, (limb >>
                        ↪ 32) & 0xFFFFFFFF);
16                }
17            }
18            return hash;
19        }
20    };
21 }

```

veb.cc

```

1 std::tuple<bool, std::shared_ptr<ISerializable>, bool>
    ↪ Veb::insert(const Natural & key, std::shared_ptr<
    ↪ ISerializable> value) {
2
3     statistic::thread_sample_set_level(universe_);
4     // BOOST_LOG_TRIVIAL(trace) << std::hex << "Inserting
    ↪ " << key << " at " << *id_ << std::dec << "("
    ↪ << universe_ << ")";
5
6     if (UNLIKELY(key.bits() > universe_)) {
7         throw std::runtime_error("k.bits()=" + std::
            ↪ to_string(key.bits()) + " is bigger than

```

```

    ↪ universe=2^" + std::to_string(universe_));
8   }
9   if (UNLIKELY(!key.is_valid())) {
10      throw std::runtime_error("Key is not valid.");
11   }
12
13   std::pair<bool, std::shared_ptr<ISerializable>>
    ↪ ret_data;
14
15   if (min().first.is_valid() == false) {
16      set_min(std::make_pair(key, value));
17      set_max(min());
18      return {};
19   }
20
21   if (key == min().first) {           // allow replace
22      ret_data = set_min(std::make_pair(key, value));
23      if (key == max().first) {
24         set_max(min());
25      }
26      return std::make_tuple(ret_data.first, ret_data.
    ↪ second, true);
27   }
28
29   Natural high;
30   Natural low;
31
32   if (key < min().first) {
33      high = min().first.high(universe_ >> 1);
34      low = min().first.low(universe_ >> 1);
35      set_min(std::make_pair(key, value));
36   } else {
37      high = key.high(universe_ >> 1);
38      low = key.low(universe_ >> 1);
39   }
40
41   bool there_was_something = false;
42   std::tie(ret_data.first, ret_data.second,

```



```
        ↪ there_was_something) = cluster()->insert(high,
        ↪ low, value);
43  if (false == there_was_something) {
44      summary()->insert(high, {}); // if this happen,
        ↪ insert bellow will be constant-time (null
        ↪ min)
45  }
46
47  if (key >= max().first) {
48      set_max(std::make_pair(key, value));
49  }
50
51  return std::make_tuple(ret_data.first, ret_data.
        ↪ second, true);
52 }
```