Master’s Thesis

Automated massive deployment of bitcoin private network nodes for network simulation

Student: Napoleon-Christos Oikonomou [AEM: 16]
napoleonoikon@gmail.com

Supervisors:
Professor Athena Vakali
PhD Candidate Georgios Vlahavas
Acknowledgments

As the end of this thesis also marks the end of my postgraduate studies, I feel the need and the obligation to thank some people who played a catalytic role in this accomplishment.

First of all, I would like to thank my supervisor Professor Athena Vakali, for her support but also for the confidence she has shown in me.

I would also like to extend my heartfelt greetings to PhD Candidate Georgios Vlahavas, for the excellent, in my opinion, co-operation and guidance that he provided throughout my work.

Thanks to Elena, Pavlos and Mary for the mental, and oftentimes physical, shelter that they provided, when it was needed the most.

Lastly, I want to thank my family for the continued support all these years, without whom, I would not be the man I am today.

Thessaloniki, February 2020,

Napoleon.
Automated massive deployment of bitcoin private network nodes for network simulation

Abstract

In 2008, a groundbreaking invention based on peer-to-peer networks came to light, Bitcoin. Bitcoin, a decentralized crypto-currency and its accompanying architecture, the blockchain, which proposed a new and revolutionary way to distribute information, in bitcoin’s particular case liquid assets, quickly caught the attention not only of researchers, software engineers and computer scientists, but also of the general public.

Since then, a lot of research is being conducted on how different fields can leverage the blockchain, to improve their products and technologies. As such, new blockchains are deployed continuously, in an effort to discover those specific structures and parameters, that would produce the best results for each field. Alas, the deployment of blockchains is a rather tedious and complex process, as one can easily discover.

In this Master’s Thesis, work has been done in an effort to create an infrastructure that, optimistically, will assist in the deployment of blockchains, by automating various necessary processes. Given the vast adoption of Bitcoin versus other blockchain technologies, it seemed logical that this infrastructure should be a tool focused on the rapid deployment of bitcoin blockchains.

Specifically, by leveraging the advantages of container-based virtualization, a software tool dubbed btc-network has been developed, whose main purpose is to abstract away complex procedures in the deployment step, and provide users not only with single-command deployments, but also, with the elimination of as much time-consuming overhead, as possible.

Napoleon-Christos Oikonomou
napoleonoikon@gmail.com
School of Informatics, Faculty of Sciences
Aristotle University of Thessaloniki, Greece
February, 2020
Αυτοματοποίηση της μαζικής εγκατάστασης κόμβων ιδιωτικών δικτύων bitcoin με σκοπό την προσομοίωση δικτύου

Το 2008, μια πρωτοποριακή εφεύρεση βασισμένη σε δίκτυα peer-to-peer ήρθε στο φως, το Bitcoin. Το Bitcoin, ένα αποκεντρωμένο κρυπτονόμισμα αλλά και η αρχιτεκτονική του, το blockchain, το οποίο πρότεινε έναν νέο και εντελώς διαφορετικό τρόπο διανομής της πληροφορίας, γρήγορα κέρδισε την προσοχή όχι μόνο ερευνητών, μηχανικών λογισμικού και επιστημόνων της Πληροφορικής, αλλά και του ευρέος κοινού.

Από τότε, διεξάγεται εντατικά έρευνα σχετικά με τον τρόπο με τον οποίο διάφορα πεδία μπορούν να εκμεταλλευτούν το blockchain, με σκοπό να βελτιώσουν τα δικά τους προϊόντα και τεχνολογίες. Ως εκ τούτου, νέα blockchain αναπτύσσονται συνεχώς, σε μια προσπάθεια εύρεσης των συγκεκριμένων αρχιτεκτονικών και παραμέτρων, που θα παράγουν τα καλύτερα δυνατά αποτελέσματα για κάθε τομέα. Δυστυχώς, εύκολα μπορεί κάποιος να ανακαλύψει ότι η ανάπτυξη και η εγκατάσταση ενός blockchain, είναι μια αρκετά κουραστική και σύνθετη διαδικασία.

Σκοπός αυτής της εργασία είναι η δημιουργία μιας υποδομής που, δυνητικά, θα βοηθήσει στην ανάπτυξη και εγκατάσταση blockchains, αυτοματοποιώντας διάφορες απαραίτητες διαδικασίες. Δεδομένης της τεράστιας υποδομής του Bitcoin έναντι άλλων blockchain τεχνολογιών, κρίθηκε σκόπιμο να εισάγεται μια υποδομή αυτή να εισάγει μια εργαλείο που εστιάζει συγκεκριμένα στην ταχεία ανάπτυξη και εγκατάσταση blockchain του Bitcoin. Αξιοποιώντας τα πλεονεκτήματα της eikονικοποίησης σε container, αναπτύχθηκε ένα εργαλείο λογισμικού που ονομάζεται btc-network. O κύριος σκοπός του εργαλείου αυτού είναι η αφαίρεση σύνθετων διαδικασιών από το βήμα ανάπτυξης blockchain, ώστε να παρέχει στους χρήστες του τη δυνατότητα να εγκαθιστήσουν πλήρη δίκτυο κόμβων bitcoin με την εκτέλεση μιας μονάχας εντολής.

Αρχικά, το κομμάτι του συστήματος με το οποίο αλληλεπιδρά ο χρήστης είναι μια Διεπαφή Γραμμής Εντολών (Command Line Interface), η οποία είναι υπεύθυνη για την συλλογή όλων των απαραίτητων δεδομένων εισόδου ώστε να δημιουργηθεί αρχικά ένα δίκτυο blockchain. Η εισαγωγή δεδομένων γίνεται τόσο μέσω από μια
σειρά ερωτήσεων που παρουσιάζονται στο χρήστη, όσο και μέσω αρχείων JSON, όταν η απαιτούμενη πληροφορία αφορά τόσο μεγαλύτερη σε όγκο εισαγωγή δεδομένων, όσο και πληροφορία που πρέπει να έχει συγκεκριμένη δομή. Για παράδειγμα, μέσω αυτής της διεπαφής, ο χρήστης ενημερώνει το σύστημα τόσο για την τοποθεσία των συσκευών που επιθυμεί να έχουν ρόλο κόμβου στο δίκτυο — δίνοντας τις διευθύνσεις IP αυτών —, όσο και για τις παραμέτρους που επιθυμεί να έχει το δίκτυο που θα δημιουργηθεί.

Στη συνέχεια, το σύστημα εκτελεί όλες τις απαραίτητες διαδικασίες με σκοπό την πλήρη εγκατάσταση του δικτύου. Για να το κάνει αυτό, χρησιμοποιεί εικονικοποίηση σε επίπεδο λειτουργικού συστήματος μέσω της τεχνολογίας Docker. Πιο συγκεκριμένα, σε κάθε δυνητικό κόμβο, δημιουργεί απομονωμένες Περιοχές Χρήστη (User Spaces) που ονομάζονται containers. Κάνοντάς το αυτό, καταφέρνει αφενός να μπορεί να εγκαταστήσει το απαραίτητο λογισμικό σε οποιοδήποτε μηχάνημα, ανεξαρτήτως του λειτουργικού συστήματος του, αλλά και της υπάρχουσας κατάστασης του (λ.χ. ποια πακέτα λογισμικού υπάρχουν ήδη εγκατεστημένα σε αυτό) και αφετέρου, να παρέχει δυκλείδες ασφαλείας στο σύστημα, αφού το δίκτυο δεν έχει πρόσβαση σε άλλους πόρους του συστήματος στο οποίο φιλοξενείται.

Το container που δημιουργείται σε κάθε κόμβο, αποτελεί ουσιαστικά όλο το απαραίτητο λογισμικό για να λειτουργήσει ο κόμβος μέσα στο δίκτυο. Έχοντας ως βάση ένα ειδικά κατασκευασμένο container που έχει δημιουργηθεί εκ των προτέρων, το οποίο περιέχει τον πηγαίο κώδικα του Bitcoin σε εκτελέσιμη μορφή, καθώς και διάφορα εργαλεία με σκοπό την βελτιστοποίηση, χρονικά και χωρικά, διαφόρων διαδικασιών, το btc-network δημιουργεί κάθε φορά ένα container, ειδικά κατασκευασμένο ανάλογα με τις απαιτήσεις του χρήστη. Ακόμη παρέχει το χρήστη διάφορες άλλες αυτοματοποιήσεις, όπως η παύση ή η επανένταξη ενός κόμβου στο δίκτυο, εκτελώντας κάθε φορά μόνο μια εντολή, κρύβοντας όλη την υπάρχουσα πολυπλοκότητα. Στο τέταρτο Κεφάλαιο της παρούσας εργασίας περιγράφονται αναλυτικά όλες οι λειτουργίες που παρέχει η υποδομή που αναπτύχθηκε, καθώς και το τρόπος χρήσης του.

Σημαντική απαίτηση κατά την ανάπτυξη του εργαλείου ήταν η επεκτασιμότητα. Συγκεκριμένα, δεδομένου ότι η παρούσα υλοποίηση βασίστηκε σε containers και όχι σε λογισμικό που εκτελείται στο αληθινό λειτουργικό σύστημα ενός μηχανήματος, υπήρξε ιδιαίτερη μέριμνα ώστε να μην υπάρχουν αυξημένες
απαιτήσεις, τόσο υπολογιστικών, όσο και δικτυακών πόρων. Για το σκοπό αυτό
ekteleísthikan diáfora peirámata schetíká me tìn epistasia'mótita kai tìn tachúthta
epikoinwínias twn kómbwn sto díktuo. Ta apotelesmata autón parousiáizontai sto
pémpto Keftáliaio tis parousias ergasiaís. Ópws faíneita kai ekeí, h chrísi
eikóniKO-poíhshs, dén dhmiourghí káneva próblhma sto díktuo pou prokúptei, to
opóio katalígei na éinai klównos enós pragmatikou diktúou.

Evn katakleíði, h upodómi pou dhmiourghíteke sta pláisia autís tis
ergasiaís, apotelei èna ergaleió ikánó na ekteleósei òles tis aparáítítites enérgeièes
 gia tìn dhmiourghía diktúwn Bitcoin se eláchisto chróno, me tò mikrótero dynató
kóstos, tósso gia to chrísi, ósso kai gia ta mikhánìmata pou òa apoteleóson toues
kómbous. Málista, dedoménon tou tròpou eisagwghís plhróforwv, kathistá tì
diádikasia dhmiourghías enós diktúou me osodhípote megálh paramtropoíhsh, mia
diádikasia h opóa éinai éukolo na ekteleóste akómy kai apó chrístes pou dén échoun
megálh episthmovnikó upórbathro sthn anáptuvh lýgmikou kai sthn episthímí tis
Plhróforikhís.

Napoleón-Xhrístos Oikonoímov
napoleonoikon@gmail.com

tmíxh plhróforikhís, scholh òstikwn episthmovn
aristotelíxio panepisthmio thessalonikhís, elláda
febrouárhíos, 2020
Table of Contents

Acknowledgments ................................................................................................................. 2
Abstract ................................................................................................................................. 3

Αυτοματοποίηση της μαζικής εγκατάστασης κόμβων ιδιωτικών δικτύων bitcoin με σκοπό την προσομοίωση δικτύου ............................................................................. 4
Table of Contents ...................................................................................................................... 7
List of Figures .......................................................................................................................... 9
List of Tables ........................................................................................................................... 11

Chapter 1 Introduction .......................................................................................................... 12
  1.1 Subject of Master’s Thesis ............................................................................................... 12
  1.2 Background and Discussion on Problems Addressed ...................................................... 13
  1.3 Novelties and Contributions .......................................................................................... 14
  1.4 Document Structure ....................................................................................................... 16

Chapter 2 Background and Related Work .......................................................................... 17
  2.1 Background ...................................................................................................................... 17
     2.1.1 Client-Server Model ................................................................................................. 17
     2.1.2 Peer-to-Peer Network .............................................................................................. 18
     2.1.3 Blockchain ................................................................................................................ 20
     2.1.4 Bitcoin and Proof of Work Consensus ..................................................................... 21
     2.1.5 JavaScript and Node.js ............................................................................................ 23
     2.1.6 JSON (JavaScript Object Notation) ......................................................................... 24
     2.1.7 YAML (YAML Ain’t Markup Language) ................................................................... 25
     2.1.8 Operating-System-Level Virtualization ................................................................... 26
     2.1.9 Docker and Docker Compose ................................................................................... 27
  2.2 Related Work .................................................................................................................... 28
     2.2.1 On the Security and Performance of Proof of Work Blockchains .............................. 28
     2.2.2 Tampering with the Delivery of Blocks and Transactions in Bitcoin ....................... 30
     2.2.3 Using Docker in High Performance Computing Applications .................................. 31
     2.2.4 Connection of Present Work with Previous Works and Implementations .............. 33

Chapter 3 Requirements, Structure and Design ............................................................... 36
  3.1 System Requirements ....................................................................................................... 36
  3.2 System Architectural Structure and Design .................................................................... 38
  3.3 Network Type and Mode ................................................................................................. 40
Chapter 4 Implementation ................................................................. 43
  4.1 Base Docker Image of Bitcoin Core ........................................... 43
  4.2 Docker Compose Configuration Files ....................................... 44
  4.3 Consensus Parameters Configuration Files ................................. 45
  4.4 Command Line Interface ......................................................... 46
    4.4.1 User Input ........................................................................ 46
    4.4.2 Required Files ................................................................ 46
    4.4.3 Required Commands ....................................................... 49
  4.5 Ease of Initial Installation ........................................................ 49

Chapter 5 Experimentation and Usage .............................................. 51
  5.1 Multi-Node Network with a Single Host ..................................... 51
    5.1.1 Experiment Setup ............................................................ 53
    5.1.2 Results Acquired ............................................................. 55
      5.1.2.1 Without Throttling .................................................... 56
      5.1.2.2 Throttling Download & Upload Speed to 63 KB/s ............. 57
      5.1.2.3 Throttling Download & Upload Speed to 5 KB/s ............. 58
  5.2 Multi-node network with Multiple Hosts .................................... 59
    5.2.1 Experiment Setup ............................................................ 59
    5.2.2 Results Acquired ............................................................. 60
    5.2.3 Relevant Source Code ....................................................... 61

Chapter 6 Conclusions and Future Work ........................................... 62
  6.1 Conclusions ............................................................................. 62
  6.2 Suggestions for Future Work .................................................... 63
    6.2.1 Graphical User Interface ................................................... 64
    6.2.2 Support for Network-wide commands ................................. 64
    6.2.3 Support for Different Blockchains ...................................... 65
    6.2.4 Experiment Continuation .................................................. 65

Appendix ......................................................................................... 66
  A. Local Compilation & Usage ........................................................ 66
    A.1 Manual Installation ............................................................. 66
    A.2 Creation Of A Binary Executable Bundle ............................... 66
    A.3 Commonly Used Commands ................................................ 67
  B. Command Line Interface UI ....................................................... 68
    B.1 Network Creation ............................................................... 68

References ...................................................................................... 71
List of Figures

FIGURE 1: CLIENT-SERVER MODEL .......................................................................................... 18
FIGURE 2: UNSTRUCTURED peer-TO-PEER NETWORK ............................................................. 19
FIGURE 3: FORMATION OF A BLOCKCHAIN ............................................................................ 21
FIGURE 4: DATA STRUCTURE OF BLOCKS IN THE LEDGER .................................................... 22
FIGURE 5: PROOF-OF-WORK ILLUSTRATION ....................................................................... 23
FIGURE 6: JSON DOCUMENT .................................................................................................. 25
FIGURE 7: YAML DOCUMENT .................................................................................................. 25
FIGURE 8: EXAMPLE OF FULL SYSTEM VIRTUALIZATION ......................................................... 27
FIGURE 9: DOCKER API WRAPPING OF THE LINUX KERNEL ..................................................... 28
FIGURE 10: GERVAILS, KARAME ET AL. QUANTITATIVE FRAMEWORK ................................... 29
FIGURE 11: EXAMPLE ADVERSARY ATTACK IN GERVAILS, RITZDORF ET AL. ......................... 31
FIGURE 12: CONTAINER-BASED & SUPERVISOR-BASED DISTRIBUTED PROGRAMMING ........... 32
FIGURE 13: MEMORY USAGE COMPARISON OF CHUNG ET AL. ............................................... 32
FIGURE 14: CPU PERFORMANCE COMPARISON OF CHUNG ET AL. ....................................... 32
FIGURE 15: ARCHITECTURE OF NETWORK FROM BTC-NETWORK ......................................... 38
FIGURE 16: EXAMPLE USAGE OF BTC-NETWORK ..................................................................... 39
FIGURE 17: DOCKER COMPOSE CONFIGURATION YAML PRODUCED BY BTC-NETWORK ........... 44
FIGURE 18: EXAMPLE CONFIGURATION JSON REQUIRED BY BTC-NETWORK ....................... 45
FIGURE 19: INITIAL SCREEN OF BTC-NETWORK ..................................................................... 46
FIGURE 20: FLOW FOR CREATING A NEW NETWORK WITH BTC-NETWORK ......................... 48
FIGURE 21: NON-INTERACTIVE USAGE OF BTC-NETWORK ..................................................... 49
FIGURE 22: ONE-CLICK INSTALL SCRIPT OF BTC-NETWORK & PREREQUISITES ....................... 50
FIGURE 23: NODE INFORMATION JSON FOR MULTI-NODE, SINGLE-HOST NETWORK ............... 53
FIGURE 24: EXAMPLE BLOCK DOCUMENT IN MONGODB ....................................................... 54
FIGURE 25: EXAMPLE LOG OF THE ARRIVAL OF A NEW BLOCK .............................................. 55
FIGURE 26: NO THROTTLING - ARRIVAL TIME PER BLOCK ..................................................... 56
FIGURE 27: 63 KB/S THROTTLING - ARRIVAL TIME PER BLOCK .............................................. 57
FIGURE 28: 5 KB/S THROTTLING - ARRIVAL TIME PER BLOCK ................................................. 59
FIGURE 29: MULTI-NODE, MULTI-HOST - ARRIVAL TIME PER BLOCK ..................................... 61
FIGURE 30: EXAMPLE INSTRUCTIONS FOR LOCAL INSTALLATION ....................................... 66
FIGURE 31: EXAMPLE COMMAND TO START A NODE VIA SSH ............................................. 67
FIGURE 32: EXAMPLE COMMAND TO START SAVING LOGS TO A REMOTE NODE ......................... 67
FIGURE 33: EXAMPLE COMMAND TO STOP A NODE VIA SSH ........................................ 67
FIGURE 34: EXAMPLE COMMAND TO STOP SAVING LOGS ON A REMOTE NODE ................... 68
FIGURE 35: EXAMPLE COMMAND TO GET LOGS FROM A REMOTE LOG LOCALLY ................. 68
FIGURE 36: SCREEN TO PROVIDE NODE INFORMATION IN BTC-NETWORK ...................... 68
FIGURE 37: SCREEN TO PROVIDE BASE IMAGE INFORMATION IN BTC-NETWORK ................. 69
FIGURE 38: SCREEN TO PROVIDE BLOCKCHAIN PARAMETERS INFORMATION IN BTC-NETWORK [1] ... 69
FIGURE 39: SCREEN TO PROVIDE BLOCKCHAIN PARAMETERS INFORMATION IN BTC-NETWORK [2] ... 69
FIGURE 40: SCREEN TO PROVIDE OUTPUT INFORMATION IN BTC-NETWORK ....................... 70
FIGURE 41: CREATION SUCCESS MESSAGE IN BTC-NETWORK ........................................ 70
List of Tables

**TABLE 1:** CONNECTION OF PRESENT WORK WITH PREVIOUS WORKS AND IMPLEMENTATIONS .......................... 35

**TABLE 2:** EXPERIMENTS' SPECIFICATIONS ........................................................................................................... 52

**TABLE 3:** MULTI-NODE, SINGLE-HOST NO-THROTTLING RESULTS ................................................................ 56

**TABLE 4:** MULTI-NODE, SINGLE-HOST 63 KB/s THROTTLING RESULTS ................................................................. 57

**TABLE 5:** MULTI-NODE, SINGLE-HOST 5 KB/s THROTTLING RESULTS ................................................................. 58

**TABLE 6:** MULTI-NODE, MULTI-HOST RESULTS ........................................................................................................ 60
Chapter 1
Introduction

This chapter introduces the core theme that has concerned and motivated the research that was conducted in this work. Firstly, the subject of this work is presented by discussing some of the key aspects of blockchain technology, focusing on its novel ideas and their potential to revolutionize many aspects of software design and computer science in general. Continuing, one of the main problems that blockchain technology and the research that is done in the field is realized, namely; the importance of deploying new experimental networks constantly and the challenges that are presented in this process. Alongside these, the main ideas for tackling these challenges that this work encapsulates are presented. Moreover, in Section 1.3, the novelty and contribution of this work is discussed. To conclude, the last Section provides an outline of the overall structure of this document.

1.1 Subject of Master’s Thesis

Peer-to-peer networking is the idea of an application architecture where computing is distributed between peers where each one is responsible for a partition of the required tasks. Those peers are all equally privileged, without the need for central coordination by servers or stable hosts and are said to form a p2p (peer-to-peer) network of nodes [1].

The value of these kinds of networks has been long realized by programmers and computer scientists since their inception in 1969 [2], but they only found their way into main production-level applications since the start of the millennium [3]. This was mainly because of the complex infrastructure required to implement a network of peers and keep them all continuously in-sync.

In 2008, a groundbreaking invention based on peer-to-peer networks came to light, from an unknown person —or a group of people— called Satoshi Nakamoto, Bitcoin: A Peer-to-peer Electronic Cash System [4]. Bitcoin, a decentralized cryptocurrency and its accompanying architecture, the blockchain, which proposed a new
and revolutionary way to distribute information, in bitcoin’s particular case liquid assets, quickly caught the attention not only of researchers, software engineers and computer scientists, but also of the general public.

Since then, there is a very large amount of ongoing research into bitcoin and the blockchain architecture in general, where researchers are constantly trying to find optimizations upon its original implementation, in order to make blockchain find its place alongside the established currency and information-exchange systems worldwide.

Moreover, even though the primary use of blockchains currently, is as a distributed ledger for cryptocurrencies, there is ongoing research on ways to exploit this platform based on trustlessness for other everyday uses. A notable example is smart contracts, which are contracts written in computer code, where there is no need for human interaction for them to be executed and as such, can be enforced in total. Another field of interest, where the use of a peer-to-peer network could bring enormous benefits, is the supply chain. This fact, has already been observed by some of the largest multinational retail corporations, who are investing billions of dollars in research regarding the security and ease of use and deployment of small, private blockchains which will give them the ability to securely, openly and subjectively track their products [5].

As such, new blockchains with sometimes slightly tweaked parameters and other times fundamentally different ones are being deployed constantly, in an effort to test and find which one would suit best different groups of interested parties.

1.2 Background and Discussion on Problems Addressed

As mentioned previously, one of the main processes that researchers working in the field of blockchain technology stumble upon, is the continuous deployment of networks in order to test various hypotheses.

This is not an easy task to accomplish, mainly because it requires extensive knowledge both in computer software and in computer networking. Noteworthily, there are even some blockchain projects that charge users who want to deploy a custom implementation of their, generally free to use, software [6].
However, this also proves to be a cumbersome procedure even for researchers with the required background. That is because of one of the key elements of blockchain networks; the rules that each participant node abides to, have to be set beforehand. As such, in order to experiment with different network parameters, one has to deploy the network over and over, when in most cases, even a single deployment is a largely time-consuming process.

Moreover, this new research field brought together many ideas from computer networking, mathematics, game theory and cryptography, among others, which resulted in interest from researchers and scientists whose backgrounds do not always include the required knowledge for compiling, installing and in the end deploying a computer network, let alone a peer-to-peer network with its many intricacies and complexity.

To conclude, given the above, it easily becomes evident that two main caveats that need to be addressed in the field of blockchain research is the creation of automation tools that help in faster deployment of blockchain, but also the creation of tools that assist in the simplification of the blockchain deployment process. As explained in the rest of the document, a tool that merges the two aforementioned categories of tools is the main context of this thesis.

1.3 Novelties and Contributions

The purpose of this master’s thesis is the creation of a tool and an infrastructure which will potentially address the problems of deploying a blockchain and creating a peer-to-peer network, helping developers and researchers in testing their hypotheses quicker, easier and lifting them of the burden of having to acquire the required knowledge of building a peer-to-peer network.

The goal of this infrastructure will be to function as a toolbox of technologies, where one could simply specify the desired parameters, and have it automatically select the required tools and use them to deploy a blockchain with as little hustle as possible. Keeping in mind installation and compatibility problems, it seemed reasonable that this infrastructure would be cut off from any local software development and be a part of the overall ecosystem.
The rationale behind the development of this project consists of three main pillars.

First of all, as already mentioned, all the required tools for creating a network’s node are grouped together in a “black box” that is independent of the otherwise everchanging nature of the network and its nodes. That is, it is not dependent neither on the structure of the desired network nor on the architecture of its individual node.

Secondly, there is almost zero-configuration needed in order to perform the most usual actions. Actions like connecting a new node to the network or manipulate some parameters before redeploying a new network of nodes.

And lastly, there is an abstraction layer between the chain’s source code configuration format, so as there is not a need for the user to be familiar with more than one language or dialect.

The cornerstone of the implementation, which was dubbed *btc-network*, is the exclusive use of free open source software, a principle that is also respected by this work as well [7].

In order to ensure the effectiveness of the developed software, experiments were run to measure and calculate statistics about the time that is required for the network to be at an equilibrium. That is, every single node has the same information and considers the exact same chain as the longest valid chain. The benefits from these experiments were three-fold. First, they provided a use case to test whether the software that was created can in fact be used in a real-world setting. Secondly, to demonstrate the incorporation of *btc-network* in an experimentation flow. And lastly, to test whether a network with multiple peers, all running in the same host, can provide similar results to a real-world one. This could be proven valuable information for rapid testing and experimentation.
1.4 Document Structure

This document is comprised of 6 chapters in total, including the introductory chapter.

In **Chapter 2**, the necessary technological and theoretical background is presented, for which the reader should be aware of, in order to be able to fully understand the concepts and technologies referred to, in the chapters that follow. Also, previous attempts to address the problem at hand are being presented, alongside comments and remarks about elements from those attempts, that were adopted in this work. There is also a mention on how this work aspires to build upon previous attempts, and in some specific cases outperform them.

In **Chapter 3**, there is a detailed description of all the theoretical and practical requirements that the system must meet, as well as the reasoning behind the selection and design of the architectures and technologies that were used to satisfy them.

In **Chapter 4**, the tool’s implementation process is presented in detail, commenting on its source code and infrastructure in general. It also describes how to use it, including how to integrate it into an application and describe some examples, as well as how to expand and improve the project.

In **Chapter 5**, examples of the use of the end-system are presented, alongside some experiments that were conducted as well as the observations and results that were acquired from them.

In **Chapter 6**, there is a discussion on the conclusions that were made during implementation, and some of the author's thoughts on future improvement and use of this implementation.

Finally, in the Appendix of this document one can find detailed instructions on the commands and tools that are required to successfully execute and use the tool created.
Chapter 2
Background and Related Work

This chapter is divided into two sections. In the first section, an effort is being made to explain in detail all the necessary theoretical and technological background that the reader needs to understand, before continuing reading this work. Both the theoretical elements and the technologies used in the implementation of the thesis are analyzed. The second section lists some of the previous works that both formed the basis and were proved valuable sources of ideas for the software that was developed in this thesis.

2.1 Background

2.1.1 Client-Server Model

In the field of computer science, the client-server software architecture model is a common software development method, in which, a client (i.e. a piece of software) requests something —a resource, the results of a calculation—, and another piece of software, the server, responds by sending the requested payload to them. Each server can serve multiple clients and each client can request different resources from different servers.

The client and the server are separate entities and can be run either on different processes on the same hardware or more commonly, on different hardware/computers. The latter, requires the existence of a communication network, and with the capabilities offered by such a large network as the Word Wide Web, this application structure is today one of the most, if not the most, common methods of deploying and operating distributed systems, where client and server are considered different, but complimentary parts, of the same distributed application [8].

What becomes apparent in client-server model networks, is that they are most often designed in a fashion in which the server node operates as the center of the network, serving many different clients. This leads to a couple of noteworthy
throwbacks. Firstly, given that the server node is the only point of information and resources of the network, as the network grows, the server has to scale its computing power resources, in order to be able to keep track of the ever-increasing demand of the network. Secondly, in this architecture, the clients are not able to communicate with one another directly and are dependent on the master node serving as an intermediary for information exchange, which is a major flaw, because not only the whole network is dependent on a specific node not failing, but also there is the inherent risk of information and resource censorship from the server.

![Figure 1: Client-Server Model](image)

2.1.2 Peer-to-Peer Network

As stated in the previous chapter, peer-to-peer (p2p) networking is the idea of an application architecture where computing is distributed between peers where each one is responsible for a partition of the required tasks. Those peers are all equally privileged, without the need for central coordination by servers or stable hosts and are said to form a p2p network of nodes [1]. These networks differ from the much more common client-server model, in the sense that all peer nodes are equal and are interchangeably functioning as both clients and servers in the network.

Even though p2p networks are a superset of client-server model network and one can correctly describe the network of Figure 1 as a peer-to-peer network, traditionally, when one talks of a network as a peer-to-peer network, they refer to an unstructured peer-to-peer network. Unstructured p2p networks are a sub-category of
networks where there is not a particular structure imposed on them, but rather, the edges between nodes follow a random distribution, much like a random graph [9] [10].

Given their ungoverned and free structure, these networks provide some significant advantages. Because all resources and computing power are shared, when a new node attempts to join the network, they carry their resources and power with them, solving the scalability problem. Furthermore, they have a very high degree of network robustness, because there is not any node more significant of any other, and as such, they are not affected by large rates of connections of new nodes and disconnections of existing ones. This proves to be very important in our modern day and age when devices switch networks constantly (e.g. smartphones, computers, IoT etc.) [11]. Most notably, though, the absence of a centralized governing node, brings an absence of both the risk of the network collapsing after a centralized attack and the risk of censorship which is more and more important nowadays.

As expected, though, there is also a number of caveats that arise from this lack of structure, with their common denominator being the lack of a single source of truth, because, as stated above, there is not a centralized node that holds all the information. Most notably, an actor-node cannot be fully certain of the state of the network, because they can only interact with the whole network through their neighboring nodes, not having any guarantees that they will not act maliciously.

Also, when a peer queries for a resource, the whole network must be flooded with the query, which is obviously a computationally expensive operation that in many cases, particularly when a resource is rarely accessed may yield no results.

Below there is an illustration of these types of unstructured peer-to-peer networks, where it is apparent that the existing edges seem to be, and are in fact, random.

![Figure 2: Unstructured peer-to-peer Network](image)
2.1.3 Blockchain

A blockchain, as its name suggests, is a —usually continuously growing— list of records, called blocks, that are cryptographically linked with one another. Each block contains, among other application-specific information, the cryptographic hash of the previous block and transaction data. The cryptographic hash is the output produced from a one-way mathematical function, where the block is its input. The characteristics of this function that are relevant in this use case are that:

- It is deterministic, meaning that a block will always produce the same hash.
- It is practically infeasible to calculate the input block from a given hash value.
- It is also practically infeasible to find two different blocks that produce the same hash.
- The function is hyper-sensitive to initial conditions, meaning that a very slight change in a block, will produce a completely different hash. This is dubbed as the “avalanche effect” [12].

The transactional data each block contains, describe the changes that occurred since the previous block. As such, a blockchain is resilient to alteration of its data. This is because a —usually malicious— change in a block, would change its hash and thus the hash of every block that came after it, resulting in invalidating the whole chain.

The first ideas about a cryptographically secured chain of blocks was described as early as 1991 [13], where it was proposed as a system where document timestamps could not be tampered with. Later work incorporated hash trees to the design, which improved its efficiency by allowing sever document certificates to be grouped into one block. [14]

The first blockchain was eventually conceptualized by Satoshi Nakamoto in the original bitcoin whitepaper [4], to serve as a distributed ledger because, as they argued, by design, a blockchain can be considered secure, and exemplify a distributed computing system with high Byzantine fault tolerance, that is, a system in which consensus must be achieved between trustless parties. In the figure below, a blockchain formation can be seen, where with green appears the genesis block, that is a piece of information that all parties must agree that is true, with black the main chain and with purple, orphan blocks, blocks that describe an alternative, albeit valid history of transactions.
For each block to be accepted as a valid block, a list of requirements must be met. One of the key requirements is that it must contain specific information from the previous block that was already on the chain. What is obvious is that the first block of the chain (pictured in green below) cannot possibly follow that rule. As such, it is a unique case where all nodes agree that this check (and some other similar ones) could be skipped for this specific block.

However, even if a new block conforms to every rule necessary, it could still end up not being part of the chain. This could happen, for example, if, by the time a node creates such a block, the network has already created many more and as such, it is not part of the chain with the most work anymore, making it a stale block (pictured in purple).

![Figure 3: Formation of a Blockchain](image)

2.1.4 Bitcoin and Proof of Work Consensus

Bitcoin is a digital cryptocurrency invented in 2008 by Satoshi Nakamoto and started being used in 2009, when its source code was released, and the first bitcoins were created [4]. It uses a blockchain as its public ledger to record bitcoin transactions. This piece of information is distributed worldwide through an unstructured peer-to-peer network, called the bitcoin network [15] by peer nodes running a piece of specific bitcoin software [16]. Each block contains, alongside the information explained in Section 2.1.3 above, transactions between different users, in the form “payer X sends Y bitcoins to payee Z”.
Using a blockchain to record transactions and distributing it through a peer-to-peer network, makes bitcoin a decentralized digital currency without a central bank or single administrator that can be sent from user to user on the peer-to-peer bitcoin network without the need for intermediaries [17].

One of the most notable novelties of bitcoin, is the way of reaching consensus in an unstructured, trustless peer-to-peer network. As stated in Section 2.1.2, in a network like this, there is the caveat that because there is not a centralized authority, there can’t be a single source of truth, that each peer can trust. To overcome this, in the bitcoin network, each node keeps a full copy of the blockchain, which is updated by continuously broadcasting every new block through the whole network. Consensus is achieved by agreeing that the only valid chain is the one that took the most effort to build, which, in Bitcoin’s case, is the longest one (i.e. the black chain, when referencing Figure 3). Extending the chain is succeeded by a process called proof-of-work mining, through which it becomes disadvantageous for malicious parties to alter the chain with invalid data. Quoting Nakamoto, “The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains.” [4]. In essence, for a new block to be added to the chain, the producing node must first solve a very computationally expensive mathematical puzzle, paying with resources, electric
energy, etc. and thus, having no feasible gain in trying to trick the network. An illustration of a proof-of-work process can be seen below.

![Figure 5: Proof-of-Work Illustration](image)

2.1.5 JavaScript and Node.js

JavaScript is a high-level interpreted programming language. Although, like the majority of different programming languages, it is based on the C programming language, yet it is dynamic, weakly typed and uses functions as first-class citizens. It is a multi-paradigm language and therefore supports both functional and object-oriented, as well as prototype-based programming styles.

It was originally created by Brendan Eich in 1995 to add programs to the Netscape Navigator browser. It was created for client-side programming, with the user’s browser being the client. JavaScript is now widely used and supported by all popular browsers [18].

Because of its simple syntax and its many features, in the last decade it has become widely used for server-side programming as well. Several powerful programming environments have been created, most notably Node.js [19]. Since
2009, the CommonJS Project is responsible for language standardization, and making JavaScript a full-blown general-purpose programming language [20].

The work of this thesis was implemented in JavaScript because it was considered appropriate to implement a tool that helps users deploy a blockchain everywhere, with a language that can be run almost anywhere. Moreover, the number of open-source third party modules is larger than any other programming language and is growing with the fastest rate [21]. A main culprit for JavaScript’s very large adoption and reusability of modules, is npm. npm is a package manager for the JavaScript programming language. It is the default package manager of the Node.js environment and consists of a Command Line Interface and a large database of registry packages. Noteworthy is the fact that npm is the largest database of its kind. npm helps developers reuse code and integrate it into their applications. It also provides an easy way to download, install and execute various binaries, without having to manually install all their dependencies [22].

2.1.6 JSON (JavaScript Object Notation)

JSON is a lightweight data exchange template. Its major advantage against other standards (e.g. XML [23]), is that a JSON text, while it can be parsed and generated by the computer, it can very easily be read and written by humans. It is based on the ECMA-262 3rd Edition Standard of the JavaScript language.

JSON is a data-interchange standard that is completely independent of programming languages, but uses conventions that are familiar to developers of the C programming family, including C, C++, Java, JavaScript, Perl, Python, and many more. These properties make JSON an ideal data exchange standard format.

JSON is built on two structures:
• A collection of key-value pairs. In various programming languages, this is interpreted as an object, structure, dictionary, hash table, key list, or associative table.
• A sorted list of values. In most programming languages, this is interpreted as an array, list, or sequence.

These are, in essence, universal data structures that all modern programming languages support. The most basic formats a JSON text can take are object, table, and value. An object is a non-structured set of pairs of names / values, while a table is a
collection of values in a row. A value can be a string, a number, a status statement (true, false or null), an object, or a table [24]. An example of a JSON document is shown below:

```json
{
   "name": "John",
   "age": 30,
   "cars": {
      "car1": "Ford",
      "car2": "BMW",
      "car3": "Fiat"
   }
}
```

Figure 6: JSON Document

2.1.7 YAML (YAML Ain't Markup Language)

Similar to JSON, YAML is a human-readable, data-serialization language. It uses both Python-style indentation to indicate nesting, but also supports a much more compact way by using [] for lists and {} for objects, making it a superset of JSON [25]. As such, it supports natively encoding scalars, including objects, structures, dictionaries, lists, arrays, hash tables, key lists and or associative tables.

One major advantage over JSON, is that its ruleset of information representation makes it possible for YAML files to correctly be read from streams. An example of a YAML document is shown below:

```yaml
---
name: John
age: 30
cars:
   car1: Ford
car2: BMW
car3: Fiat
```

Figure 7: YAML Document
2.1.8 Operating-System-Level Virtualization

In the field of computing, the generic term virtualization, refers to the process of creating a virtual —as its name suggests—, rather than an actual version of something. Examples of this include virtual storage devices, virtual computer network and virtual computer hardware platforms in general. This act of dividing actual pieces of hardware into virtual machines started in the 1960s, where large organizations, most commonly universities, needed a way to divide system resources of their mainframe computers, in order for them to be used for different applications. Since then, the meaning of the term has broadened where nowadays it is most commonly referring to software resources virtualization [26] [27].

Operating-system-level virtualization refers to the paradigm of the kernel of an operating system to allow the coexistence of multiple, isolated from each other user-space instances, usually called containers. Much like different user accounts on a computer, where each user sees only their own workspace and experience this as the working station in its entirety, software programs that are executed inside a virtual operating system, can only see their container’s contents, resources and devices, in contrast to programs that are running, conventionally, on ordinary operating systems, which are aware of all available resources. This idea of creating self-contained machines, provides a number of advantages.

First of all, a program running inside a virtual container, while it expects to see the whole computer, in actuality, it can only view the container’s allocated resources, believing them to be all that is available. As it is easily evident, this provides a very strong extra layer of security. A malicious program running in a virtual environment will not be able neither to affect its host machine, nor to read, write or in any way alter data without the proper authorization.

Another major advantage is in sharing software between machines and users. By using —and sharing— a virtual machine, users are able to execute different computer programs, without having the burden of finding, understanding, downloading and installing all the —sometimes complex— required dependencies and interconnection it may have with the operating system and different programs. One can view this as sharing the entire computer, instead of just the executable program.
Figure 8: Example of full system virtualization

2.1.9 Docker\textsuperscript{1} and Docker Compose\textsuperscript{2}

Continuing on Operating-system-level virtualization, as described in Section 2.1.8, Docker is a set of, what is usually called platform as a service (PaaS) products, that leverage this type of virtualization to bootstrap software in packages called containers. As stated, these containers bundle their own software and configuration and are separate both from the hosting operating system and from each other. They can communicate with each other, though, through well-defined channels [28] [29]. This allows a user to run an application on a Container, which is similar to a Virtual Machine, making sure that all dependencies will be installed correctly, with no compatibility issues. A container can be executed there with the guarantee that the execution environment exposed to the application is the same in development, testing, and production. It defines an abstraction for over the Linux kernel in order to keep track of machine-specific settings and thus be able to run —unchanged— on many different machines, with many different configurations [28].

\textsuperscript{1} https://www.docker.com/
\textsuperscript{2} https://docs.docker.com/compose/
The technology and capabilities provided by Docker are used extensively in this work and are being described in much more detail in the following Chapters.

Docker Compose is a tool from the Docker suite of products, that is responsible for defining and executing multi-container applications. Through its configuration files and command line interface utility, it assists users to configure an application’s services and is responsible for the creation and start-up processes of containers. Like its companioning software, Docker, it is used extensively in the work and explained in more depth in the chapters that follow.

2.2 Related Work

2.2.1 On the Security and Performance of Proof of Work Blockchains

In this work by Gervais, Karame et al. [30], the authors propose a new quantitative framework to analyze the security, performance and overall consensus implications that arise when using various consensus and network parameters. Then, using that network, they try to devise optimal models for creating strategies to attack the network, and use that model as a subjective way to compare different networks.
As it can be seen from the figure above, the first half of the proposed framework consists of a Proof-of-Work (PoW) Blockchain. As such, to be able to evaluate the security and performance of various blockchain instantiations in a practical way, the authors constructed a simulator of the Bitcoin network.

Their simulator is comprised of provable mathematical equations that describe different parts of the network, whose parameters resulted from empirical values and observations gathered from the real bitcoin network and internet service providers. Nevertheless, given the enormous complexity and ever-changing nature of a network such as Bitcoin and the environment around it (e.g. people, economy and so on), the number of operations that can even be modeled, no matter how erroneously, is limited.

Recognizing that, the authors worked on replicating accurately, mainly the network aspects that affected their specific research questions, disregarding other, irrelevant to them operations (e.g. the propagation of transactions).

Continuing, they provide their results, that show —in a novel way—, how changes in just the network’s parameters, without tampering with neither its architecture nor with its consensus rules, can lead to significant improvements (they argue that Bitcoin’s transaction throughput could double). Noteworthily, none of these changes affects the network’s security, in any way.

What becomes apparent from this work, is the fact that the main blockchain networks cannot be dependent only on theoretical assumptions for their configurations, and there must be a phase before adaptation of a new change, where as many different variations as possible, are tested on chains as close as possible to the main one. To their defenses, Bitcoin developers do those tests on Bitcoin’s Testnet,
a network that is a clone of the main blockchain, but what they cannot test, is if a new change accompanied by an adjustment to previous changes could yield even better results.

2.2.2 Tampering with the Delivery of Blocks and Transactions in Bitcoin

In this work by Gervais, Ritzdorf et al. [31], the authors focus on the scalability measures taken by the Bitcoin network’s nodes and explore the possibility that a malicious actor could exploit these measures, in order to tamper with the security of the network.

Initially, they argue about which subset of the main measures that have been already taken by the network has the most importance, regarding security. They identify them to be, among others:

• The existence of an internal reputation management system, in order to combat broadcasting of ill-formed blocks and transactions.
• The existence of an advertisement-based information request system, in order to minimize the amount of information spread in the network.
• Static timeouts to prevent blocking, while tolerating different kinds of network latency.
• Logging of the order of the received transaction advertisements.

Continuing, they conduct different experiments in order to prove their hypothesis that the network can, indeed, be exploited, because of these measures. Noteworthily, they used a real network of five full nodes for their experiments. These nodes were scattered around the globe and as such provide a good approximation of the real network.

Concluding, they propose countermeasures, in order to enhance the security of the network, without negatively impacting its performance. These measures are listed below. What is a considerable advantage of these countermeasures, is, as one can easily observe, that they do not increase the complexity of the network;

• Convert Static Timeouts to Dynamic.
• Updating Block Advertisements; Do not send inv messages and keep track of block advertisers.
• Update Transaction Advertisements; Filter IP addresses and choose senders randomly.

The figure below provides an example attack, as described by the authors.

2.2.3 Using Docker in High Performance Computing Applications

In this work by Chung et al. [32], the authors explore and review the advantages and disadvantages of the two main methods of software virtualization, namely supervisor-based and container-based. In the latter, they focus exclusively on Docker, the main technology used in the field, which was described in detail in Section 2.1.11 and 2.1.12. Firstly, they discuss the different benefits and limitations of each virtualization technique and conclude that in general containerization performs significantly better, mainly because of the minimal overhead, when compared with virtual machines.

Continuing, they recognize that the main caveat of containerization is the complexity presented when one attempts to run distributed computing software using containers. To that end they propose a simple model that alleviates that complexity.
Then, they conduct different experiments executing both CPU intensive and memory intensive applications. As it can be seen from the figures below, the results show that containerization is a far better alternative to full virtual machines.

Figure 12: Container-based & Supervisor-based Distributed Programming

Figure 13: Memory Usage Comparison of Chung et al.

Figure 14: CPU Performance Comparison of Chung et al.
Concluding, they argue that it may be the case that when using a single bare-metal machine for hosting either Docker containers of full scale virtual machines, Docker prevails, but, when one attempts to take advantage of the portability of software that these options provide, creating a cluster of containers of VMs, there performance is on par. However, the much easier to use Docker, seems to be the overall better choice for, at least, CPU intensive applications.

2.2.4 Connection of Present Work with Previous Works and Implementations

Even though this work is not a direct continuation of any of the aforementioned works, nonetheless, it is a combination of all the elements that were considered appropriate by all other approaches.

Regarding the work of Gervais, Karame et al., it was the main inspiration for this present Thesis. As the authors recognized, albeit indirectly, there is a very important demand of having a way to deploy blockchains with different parameters, in order to better test if theoretical hypotheses of the network’s behaviors, are, indeed, correct. The creation of software that assists this, is the common ground between their work and this thesis. However, as it is easily apparent, this work tries to solve a very important caveat of the authors’ implementation, that they created software that assists fast deployments of simulation of networks, whereas this Thesis created software that assists in fast deployments of real networks. btc-network is not a panacea though, but mainly a supplementary tool. Using a simulator, one is able to create and test very large and complex configurations, without having the problem of resources and scalability, whereas using a real network, one can be more certain that there will be fewer variations from the main blockchain. Minimizing the overhead was one of the main concerns of this work and explanation on how it was addressed is discussed in detail in the chapters that follow.

Regarding the work of Gervais, Ritzdorf et al., this work borrowed the main research the authors did on the most important scalability measures taken by the network and future improvements. Given that it would be infeasible to create an exact replica of the real network, rapidly and continuously, this work uses a form of virtualization, as the one that explained in previous Sections, for the network’s nodes.
As such, there was a need to know if this virtualization would change the nature of the network in any way and also, if it would be a wise investment for the future, when new measures would be adopted by the main network. As it appears, luckily, this is not the case, and is safe to use the results that would be derived from experiments running on blockchains produced by btc-network. Also, it seems safe to use btc-network to create and run experiments even on the same machine, without having significant differences from a network where nodes are scattered geographically, except—obviously—the absence of latency and the very high bandwidth, both of which can be approximated with software, such as trickle, as detailed in the Chapters that follow.

Trickle is a highly configurable user space bandwidth manager. It works as a bandwidth shaper that enables the user to limit the bandwidth that a specific protocol is using so that they can either maintain multiple simultaneous connections and not end up in a traffic jam, or,—as used in the premise of this work—limit the available bandwidth an application can have [33].

Regarding the work of Chung et al., this was the main research that this thesis was depended upon, in order to, firstly, conclude if the choice of virtualization is valid and its main advantage, ease-of-use, does not become hindered by performance issues and, secondly, to choose between full scale virtual machine and container based virtualization. Interpreting the authors’ results, container-based virtualization was the architecture that was chosen. Moreover, this is on par with the overall consensus of the community, that container-based virtualization, and more specifically Docker, is a technology which lives up to its hype in recent years [34].

Finally, noteworthy is the fact that this work is an open source project, readily available to developers and researchers freely.

The table that follows, provides a surface summarization of the aforementioned ideas and approaches, as well as how they inspired this research.
<table>
<thead>
<tr>
<th>Literature</th>
<th>Subject</th>
<th>Key Ideas</th>
</tr>
</thead>
</table>
| [30]       | • Quantitative framework to analyze security, performance and overall implications when using various consensus and network parameters.  
            • Construction of a simulator of the Bitcoin network. | • Theoretical guarantees regarding changes’ results are not enough.  
            • There must be a test-phase in a as close to real network as possible, before adopting a new change.  
            • Author’s tool deploys *simulations* of networks, whereas this work deploys *real* networks. |
| [31]       | • Scalability measures taken by Bitcoin’s blockchain.  
            • Existence of ways to exploit these measures in order to tamper with the network’s security.  
            • Proposal of countermeasures to combat malicious authors. | • Network properties have a huge impact on security.  
            • The use of virtualization does not interfere in any way with countermeasures against malicious parties.  
            • A single-host network has almost identical properties with a multi-host one. |
| [32]       | • Exploration, review and comparison among supervisor-based and container-based software virtualization.  
            • In the case of container-based, the Docker technology is reviewed. | • The choice of virtualization is valid and its main advantage, ease-of-use, does not become hindered by performance issues.  
            • Choose between full scale virtual machine and container-based virtualization. |
Chapter 3
Requirements, Structure and Design

This chapter describes the theoretical, functional and nonfunctional software requirements of the system as well as the structure and design choices that were made in order to create the general structure of the tool that was developed in the context of this thesis. Their description is divided into three Sections, the first of which describes the requirements that the infrastructure deemed necessary to fulfill, explaining the reasoning behind their selection.

The second Section describes the architecture of the system, which, as mentioned in the previous chapter, is based heavily on virtualization and in particular, containerization. Moreover, the structure of the system developed is analyzed, both internally and from the user-facing side. In addition, the end of the chapter lists some of the key technologies/products used and presents the final structure that the system acquires by using them.

3.1 System Requirements

**btc-network** was implemented using the JavaScript programming language ecosystem and is designed for use by software developers and blockchain researchers. For this reason, throughout the whole design and implementation process, a basic knowledge and understanding of JavaScript, JSON and Bitcoin’s Core source code and logic, was taken for granted.

The system was designed in order to meet certain requirements, which include both operational and usage requirements. These requirements originated mainly from the main pain points of conducting blockchain related research, as described in Section 1.2, but also from general principles when designing computer software, for example ease of extendibility.

Regarding the operational requirements, these are analyzed as follows:
• Be in the form of a CLI, so as it does not depend on the existence of graphical hardware, and thus be able to run on machines remotely. Moreover, a Command Line Interface is lighter, faster and more easily adapted to already known usage patterns.

• The only required input for the deployment of a network, albeit a “vanilla” one, should be the list of the IP addresses, or, equivalently, the hostnames of the machines that the user desires to act as nodes.

• Provide the ability for users to give their own custom-made instance of a node, which may be different from the default.

• Provide the ability for users to give their desired consensus and network parameters (e.g. Block size, Network difficulty, etc.)

• Not be depended upon the Operating System or the Architecture of the machine that the user wants to turn into a node.

• Not be depended on an Internet Connection, but rather only on the existence of a local network between the nodes.

• Not require source code compilation in order to assist the rapid deployment of multiple networks.

• Provide security guarantees, in the sense that each node process should not be able to access other resources/files of the hosting machine.

• Provide a way to create a network with multiple nodes, on a single machine.

• Have as close to none as possible overheard, so as not to hinder network scalability.

• Handle all errors appropriately without threatening the integrity of the network.

Regarding the usage requirements, those include:

• Fast and easy replication of the source code on the local user’s environment.

• Ease of the learning process and usage of the software.

• Easy installation and integration into existing software projects.

• One-click connection and disconnection of nodes from the network.

• Structured, predefined way to provide network configurations.

• Guarantees for the smooth operation of the platform created on a permanent basis.
3.2 System Architectural Structure and Design

As described in previous Sections, the software that was created in the context of this thesis uses containerization at its core. In particular, Docker containers were used, in order to create bitcoin nodes seamlessly and with ease.

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. The Docker client and daemon communicate over a network interface.

As such, in this work, the proposed method of deploying a bitcoin network is to deploy a network of Docker containers that are using a predefined image of the required Bitcoin Software, configured appropriately. In particular, **btc-network**, in its generic use, creates each node as a container and then, guides the user on how to execute that container on their machines. To do so, it uses a custom-built image containing Bitcoin’s core software precompiled, as a basis, which is altered according to user specifications, appropriately. Noteworthily, the system is agnostic towards the host of each node. As such, in its generic use, the network can contain multiple nodes per hosting machine.

An example of the structure of such a network can be seen in the figure below. In this use case, a network of twelve nodes is deployed, using only three hosting machines.

![Figure 15: Architecture of network from btc-network](image)
Regarding the user-facing Command Line Interface, its main responsibility is to hide all of this design complexity from the user, acting as a wrapper for execution and handling of all the chosen configuration by them. It guides the user with a series of simple questions, asking about which operation they desire. For more complex questions (e.g. node IP addresses, custom network parameters etc.), it requires a JSON file as input, whose structure is documented and easily understood. An example demonstration of the full structure of this implementation is presented in the figure below.

![Diagram of btc-network](image)

*Figure 16: Example usage of **btc-network***

In other words, after gathering the required input from the user, **btc-network** creates the required image configuration and blockchain requirements that are specific for each node, in order for a full-blown network to be deployed.
3.3 Network Type and Mode

In general, each bitcoin node can operate in three different types of the network: the mainnet, the testnet and the regtest.

Mainnet is the original and main network for bitcoin transactions, where bitcoins have real economic value. It consists of trustless nodes whose owners are generally stranger to one another and as described previously, a new block occurs every ~10 minutes. In essence this is the blockchain peer-to-peer network that is detailed in Chapters 1 and 2.

The testnet is an alternative Bitcoin blockchain, to be used for testing. Testnet coins are separate and distinct from actual bitcoins and are never supposed to have any value. This allows application developers or bitcoin testers to experiment, without having to use real bitcoins or worrying about breaking the main bitcoin chain [35]. Thus, in order to ease development experience, testnet is periodically reset. At the time of writing, this has happened twice, and developers are now deploying on the third version of testnet\(^3\)\(^,\)\(^4\)\(^,\)\(^5\).

Contrary to the previous two network types, regtest —also known as Regression Test Mode— is a private network that a user can create at will, starting from its genesis block. The main difference from mainnet and testnet is that this chain provides each node with the ability to almost instantly generate a new block on demand for testing events and can create private bitcoins with no real-world value. As is evident this network provides complete control over the environment. Many developers consider regtest mode the preferred way to develop new applications [36].

As such, the nodes that are created by **btc-network**, are part of a private regtest peer-to-peer network.

---

\(^3\) https://github.com/bitcoin/bitcoin/commit/5cbf75324d1509a1262b65c5073314a4da3f6d77/
\(^4\) https://github.com/bitcoin/bitcoin/commit/98ba262a48b66cae8478525e809898512e997948/
\(^5\) https://github.com/bitcoin/bitcoin/commit/feeb761ba07af74a7cd78b8c8f7c2a961fd9ea1c/
3.4 Basic Technologies Used

As mentioned previously, both the Command Line Interface itself and the scripts that are responsible for input parsing, were created using the JavaScript language, running within the Node.js Engine. Moreover, every configuration file that was used from the system abided to either the JSON or YAML specification standards, as described in Chapter 2.

Two core third-party software modules were used for the present implementation of the system. The first of which is Inquirer.js, an open source framework, widely used in the Node.js ecosystem [37], which has been used as the mainstay for the created Command Line Interface, assisting in gathering user input through many, easily understood by the user, series of questions. The second is Docker Compose, which, as described in Section 2.1.10, served as a tool for defining and running multi-container Docker applications.

Docker Compose provided two noteworthy advantages. First of all, it caches the configuration that is used to create a container. As such, when a service that has not changed since its last use, is restarted, Docker Compose reuses the existing containers. Reusability of containers means that one can make changes to their environment very rapidly, with minimal—if any—recompilations and overhead. Secondly, it provided an easy way to link virtual storage volumes inside a container, to real volumes on the hosting machine, providing the user with the ability to read/edit all files used by the node, if they so desire. Also, when a node is restarted, it searches for any containers from previous runs, it copies the volumes from the old container to the new container and as such, ensures that any data that has been created in volumes is not lost.

Two important software products were also used for this infrastructure, nvm (Node Version Manager) [38], which was utilized to create a one-click install script that must be initially run on a machine in order to install the Node.js engine, necessary for execution on btc-network on any environment, and pkg [39], a tool that enables you to package Node.js projects into an executable that can be run even on devices without Node.js installed. Pkg was used to create ready-to-execute binaries for the most common operating system and architectures.
Although both of these seem to provide similar functionalities, their main differences come from the trade-offs in application size that comes with being agnostic of the execution environment when precompiling software.

Moreover, Dockerhub [40] and npm [41] were used to host and distribute all the necessary modules and base images, as described previously.

Lastly, for the experiments that were executed in Chapter 5, as detailed there, the Database MongoDB was used to store every information necessary.
Chapter 4
Implementation

This chapter describes in detail how the software is implemented, taking into account all the requirements outlined in Chapter 3. An important consideration during the implementation was the ease of adding new features and extending existing functionality. The Sections that follow, explain in detail each part of the implementation, the sum of which comprises the software tool that was developed in the context of this thesis.

4.1 Base Docker Image of Bitcoin Core

As explained previously, the core output of btc-network is a group of Docker containers, one for each node. To create each container, a base was created first, in order to be able to build upon that, recompiling only changed pieces of software.

This Docker Image contains the software Bitcoin Core [13] [42], which is necessary to run a full node. The node was built from source using the latest —openly available— source code and is updated regularly. It uses ccache [43], a compiler cache that speeds up recompilation, by caching previous compilations and detecting when the same compilation is being done again.

Given the requirement for cross-compatibility across different operating systems and architectures, it also encapsulates a full image of the Linux Ubuntu Operating System [44], the most popular Linux Distribution, which, fortunately only increases the image’s size by ~7%, which makes the image ~355 MB.

Noteworthily, the image contains the bare minimum software to run a bitcoin node. It does not include a wallet, tests, benchmarks or a Graphical User Interface.
4.2 Docker Compose Configuration Files

As mentioned previously, Docker Compose was used in order to define and execute the different containers. For it to work, a configuration file is required, written using the YAML standard, which was described in Chapter 2. A minimal configuration file can be seen in the figure that follows. Referring to this —minimal— example there are three noteworthy points explained below.

```
version: '3'
services:
  btc-node:
    build: .
    ports:
      - '18502:18444'
      - '18401:18443'
    expose:
      - '18444'
    volumes:
      - './data/btc-node:/root/btc-node'
    command:
      - '-conf=/root/.bitcoin/bitcoin.conf'
      - '-datadir=/root/btc-node'
      - '-addnode=192.0.2.0:18501'
```

*Figure 17: Docker Compose Configuration YAML produced by *btc-network*

First of all, it maps internal ports to user-defined ports of the hosting machine, to maximize configurability, but also to make it able to execute multiple nodes on the same machine.

Secondly, it links virtual storage volumes to real ones, providing the ability to view, edit and manipulate the blockchain’s files freely and easily, even if the node is stopped or deleted.

Thirdly, but most importantly, it builds the image that will be used to execute the container (*build: . command*), a wrapper image which extends the base one, that was mentioned in the previous section, by substituting the source code files that are
necessarily edited, to match the user’s needs. The existence of this wrapper image, alongside ccache is the cornerstone of rapid deployments.

Noteworthily, it also sets a restart policy that tries to continuously restart nodes in the event of sudden failure. However, this only happens if a node manages to initial function as expected for at least 10 seconds. This is done to prevent an always crashing node from going into a restart loop.

4.3 Consensus Parameters Configuration Files

In order to define various consensus and blockchain parameters a configuration JSON file is required from the user. This file is the one place of configuring the blockchain before deployment. A list of all the available options, alongside their default values is listed in the online documentation of btc-network. An example input file that overrides some of the options is depicted below. The two top level keys, namely `chainparamsCPP` and `consensusH`, refer to the two source code files of bitcoin’s core, that contain all configurations used by a node.

```
{
    "chainparamsCPP": {
        "nSubsidyHalvingInterval": 1000
    },
    "consensusH": {
        "COINBASE_MATURITY": 50
    }
}
```

*Figure 18: Example Configuration JSON required by btc-network*

---

4.4 Command Line Interface

As already mentioned, in order to hide all this complexity, and transform user needs into the corresponding pieces of software, a Command Line Interface was created. Its purpose is to first provide the user with an easy to understand interface, and then create and execute all the required software, in order to deploy a bitcoin network. Also, another advantage of having such a tool to one’s disposal, is the ability to execute many complex functionalities with a single command.

4.4.1 User Input

As described in Section 3.3, in order to gather user input, the software module Inquirer.js was used. With it, a series of questions was created in order to understand the user’s needs. The figure below depicts the initial screen the user is presented with, when they first execute the tool.

![Initial screen of btc-network](image)

Figure 19: Initial screen of btc-network

4.4.2 Required Files

The first, and most important, feature of the software that was created, is the automated creation of all the required configuration files, Docker compose YAMLs and images that are needed to deploy a full-scale network. As previously mentioned, in order to do so, the tool requires from the user some basic information:
• The .json file that contains information about the location of nodes (i.e. their IP and the desired port).
• If they want to use a different image as the base software of a node. The default one is explained in Section 4.1.1.
• If they want to provide a .json file with custom consensus parameters or use the default ones.
• Which folder to use in the executing machine to save all required data. Data like the configuration files, the blocks of the blockchain and so on.

These options are depicted in the flow diagram that follows. Screenshots from the Interface as presented in Appendix B.1.

After provided with all the required inputs, the software starts executing the necessary routines in order to create the required files, in their appropriate format by Docker. Noteworthily, in each step, the necessary tests are executed to determine if the answer is valid (e.g. the given paths exist, the given files have the required format etc.).

Continuing, the software firstly creates the required output folders and files, if they do not already exist and downloads the required base images. After that, a custom parser is executed in order to alter appropriately the source code of bitcoin, with the provided input.

Lastly, if all tests pass and all files were created successfully, the tool exits with a success message for the user.
Figure 20: Flow for creating a new network with *btc-network*
4.4.3 Required Commands

The other two functionalities provided by the software concern the starting and stopping of a node. Those can be configured either using the interface, similarly to what is shown in the figures above, or even in a non-interactive mode though single commands. Noteworthily, if any of these commands fails, the user is informed with an appropriate message and any other node in the network remains unaffected.

The documentation of using the software in this mode can easily be found in the tool, as shown in the figure that follows.

![Figure 21: Non-interactive usage of btc-network](image)

4.5 Ease of Initial Installation

The ease of installation and usage was one of the main concerns when developing the software for this thesis. Obviously, one cannot expect that there will not be any prerequisite work to run the software, but an effort has been made to not only keep this effort to a minimum, but also, eliminate it for each successive use of the software.
As mentioned below the only real requirement from **btc-network**, besides from the tool itself, is Docker. Docker is a widely popular product and as such, it does not seem unreasonable to expect that it is already been used by the majority of users.

Moreover, even though **btc-network** is distributed through precompiled binaries, it technically is ran inside the Node.js runtime. Node.js, as previously mentioned in Chapter 2, is very widely adopted and used, and as such, it would seem appropriate to rely on npm’s distribution of btc-network, which is a lot smaller in size.

Nevertheless, in the context of this thesis, a script has been created that checks and installs, if necessary, all the required pieces of software. For completeness, the one-click install command is presented in the figure below.

```
```

*Figure 22: One-click install script of btc-network & prerequisites*
Chapter 5
Experimentation and Usage

As can easily become apparent from the previous Section, the use of btc-network is pretty straightforward. However, its use cases have some tiny differences from the default deployment of a blockchain. As such in this Section, an effort has been made to provide explanations and descriptions on how to use the software developed, and, by the implementation of some experiments that were executed, provide insights about its overhead.

The main purpose of the experiments proposed below, was to measure and calculate statistics about the time that is required for the network to be at an equilibrium. That is, every single node has the same information and considers the exact same chain as the longest valid chain.

While the results below showcase mainly a proof of the arguments that were made previously about the selected infrastructure not interfering in any way with the networks performance, they also showcase, hopefully successfully, the main advantage of the tool that was created, which is the ease of use either when first deploying a new network, or when redeploying a network with modified parameters.

5.1 Multi-Node Network with a Single Host

This use case concerns the creation and deployment of a Multi-node network in a single machine. This is the most approachable way to conduct research on the bitcoin blockchain, because, as is evident, the resources that it requires are minimal. As explained in previous chapters, the network that is created in this case resembles strongly the real one, but with one big caveat; latency and bandwidth. To that end, the bandwidth manager trickle was used.

Different experiments were run, for different connection speeds. Each of those, alongside the reasoning for their selection is explained in the table that follows.
Table 2: Experiments’ Specifications

<table>
<thead>
<tr>
<th>Network Up/Down Speed</th>
<th>No. of Nodes</th>
<th>No. of Hosts</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>– (LAN)</td>
<td>9</td>
<td>1</td>
<td>As a baseline to test the similarity of a network in a single host with a real-world one.</td>
</tr>
<tr>
<td>63KB/s</td>
<td>9</td>
<td>1</td>
<td>The lowest estimation of internet connection speeds worldwide [45].</td>
</tr>
<tr>
<td>5KB/s</td>
<td>9</td>
<td>1</td>
<td>Equivalent of a Dial-Up connection in the 1990’s [46].</td>
</tr>
<tr>
<td>–</td>
<td>20</td>
<td>20</td>
<td>Real-world scenario.</td>
</tr>
</tbody>
</table>

Specifically, taking into account the range of internet connection speeds worldwide, which is among 0.5Mbps and 26.7Mbps [45], experiments were run for these two edge cases, as well as for speeds as low as 5KB/s, the equivalent of a Dial-Up connection in the 1990’s.

The reader should keep in mind that the nature of the blockchain that was deployed, has one crucial different rule. Instead of trying to keep the mining throughput to one block per almost ten minutes, like in the real network, this time, a block is mined almost instantly, as long as a node gets information of a previous block having been mined. That is, in each step of the expansion of the blockchain, not only there is only a single node that is responsible for creating a new block (chosen at random in each step), but also, it does not initiate the generation process until it acquires the most recent state of the blockchain. This is done in order to avoid unnecessary, for the sake of these experiments, CPU and I/O load. For example, if the longest chain in the network consists of ten blocks, but the node that was chosen as a miner has not yet been informed of it and only knows of a chain with five blocks, then, the block that it will generate will eventually be discarded. Of course, such a block would not hinder the results and so this was done purely as an effort to better manage the available resources for the experiments. Consequently, the time between the mining of two blocks, is generally in the order of seconds.

The setup for the experiment, alongside the explanation of its implementation and the results that were acquired, are presented below.
5.1.1 Experiment Setup

First of all, the network was deployed through **btc-network**, by using the configuration shown in the figure below. As it can be seen, it consists of nine full nodes, all running in the same machine. To indicate this more easily, the reserved `host.docker.internal` IP address was used (but it could just as valid be replaced by `localhost`, for example). Then, a custom base image was used, namely `iamnapo/btc-network:trickle`. This image adds the option to set the `BTC_SPEED` environment variable, in order to throttle download/upload speeds of the node.

![Figure 23: Node information JSON for Multi-node, Single-Host Network](image)

This number was chosen so as each node could be connected to eight different peers, which is not only the default, but also the recommended case in bitcoin’s peer-to-peer network. The network was deployed on a machine with a 4-core CPU clocked at 2.1GHz and 4GB of RAM. Also, noteworthy is the fact that the bare minimum requirements for deploying a single full node call for a machine with at least 512MB of RAM.

Having the network successfully deployed (using the **--run** command), a script was executed, that was responsible for mining new blocks.

---

9 https://bitcoin.stackexchange.com/a/8140
10 https://bitcoin.org/en/bitcoin-core/features/requirements
Specifically, its first responsibility is to create a random number of at most 50 transactions and add them to the mempool, through a node chosen at random. The mempool is a memory pool [47] saved in each node, that contains all unconfirmed bitcoin transactions. Each transaction saved in mempool wait to be included into the blockchain, in order for it to become confirmed.

Then, using the appropriate commands, it mines the block using a node chosen, again, at random, and saves some crucial information about it into a database.

In computer science, a database is any organized collection of data. A key feature of databases is that through their design and data hierarchy, they enable the user to quickly retrieve, write and update data.

The two major categories of database architectures are the traditional relational databases and the more modern non-relational databases. On a relational database, the data is organized into related tables and their management is done with queries, usually through SQL (structured Query Language), with the use of relational algebra [48].

Non-relational databases (commonly called NoSQL), on the other hand, use different models as a basis and are more suitable in applications that require storing and processing large volumes of data of different types within a short time.

In the present work, the non-relational database MongoDB was used. MongoDB was chosen because it has a very good interface with Node.js and because it stores its data in BSON format, which is the JSON binary representation [49].

An example document is shown in the figure below.

```
{
  arrivedAfterMillis: [],
  height: 2437,
  nTx: 19,
  minedAt: 1577794673529,
  blockHash: "34a72286fb01903f336d0118ff8fc5dc8b9f66af21f86ca2243256735c807cd5",
  millisToMine: 23.836994,
  minerNode: 1,
  createdAt: 1577794673545,
  updatedAt: 1577794673545
}
```

*Figure 24: Example Block document in MongoDB*
Notice that the height property refers to the height of the chain that this block belongs to, which is not necessarily the longest in the network. Given that this process is continually executed, choosing a different node each time, this could possibly lead to a number of stale blocks. To remedy that, an artificial delay has been added, which, however, is completely taken into account and thus eliminated from every single one calculation.

This process is repeated until 200 blocks have been added to the longest chain. Noteworthy is the fact that each node has the private key for a specific address, allowing circling through transactions, fees and mining rewards and thus resulting in a continuing stream of funds, to continue running such an experiment with an arbitrary number of blocks.

Afterwards, the log files from each node were gathered and with the use of another script, the arrivedAfterMillis was updated, with the difference in milliseconds from the time a block was created, to the time it was received in each node. An example of such a log is shown below.

![Example log of the arrival of a new block](image)

5.1.2 Results Acquired

After flooding the network with new blocks and transactions and having acquired the resulting times, the results that occurred are the ones that are presented below. Regarding system usage it was observed that CPU usage was ~10%, which was expected, given that when using Regression Testing Mode, there aren’t any particularly computationally expensive operations.

An interesting insight, though, was the RAM usage which fluctuated around ~80MB per node on all cases.
5.1.2.1 Without Throttling

This first case was used as a basis for comparing different speed bounds and to represent the theoretical best-case scenario. Given that all nodes are local to the same machine and are using the Local Loopback Address, where communication is near-instant, as it is only CPU bound [50]. The observed latency is in the order of \(\sim 0.1ms\).

<table>
<thead>
<tr>
<th>Metric</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Time</td>
<td>1 ms</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>319 ms</td>
</tr>
<tr>
<td>Mean Time</td>
<td>67 ms</td>
</tr>
<tr>
<td>Median Time</td>
<td>59 ms</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>50.37 ms</td>
</tr>
</tbody>
</table>

As expected, the minimum time is almost nonexistent, because the mining node was also one of the nodes in the network. Also, the average time for a block to arrive in every node is sub-second. The plot below shows the average arrival time for each succeeding block, as well as its respective minimum and maximum time of arrival in each node. As it can be observed, the network’s throughput is more than enough for congesting blocks, and as such, there does not appear to be any difference as the chain grows.

Figure 26: No Throttling - Arrival Time per block
5.1.2.2 Throttling Download & Upload Speed to 63 KB/s

To continue, with the use of trickle, the network’s speed was limited to 63 KB/s, which is almost the lowest available speed worldwide, as mentioned in the previous Section. The results are as follows.

Table 4: Multi-node, Single-Host 63 KB/s throttling Results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Results</th>
<th>Increase % from No-throttling network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Time</td>
<td>1 ms</td>
<td>±0%</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>554 ms</td>
<td>+74%</td>
</tr>
<tr>
<td>Mean Time</td>
<td>99 ms</td>
<td>+48%</td>
</tr>
<tr>
<td>Median Time</td>
<td>88 ms</td>
<td>+49%</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>72 ms</td>
<td>+44%</td>
</tr>
</tbody>
</table>

Even by keeping in mind that this network contains a very small number of peers, the results show —astonishingly— that, even with the slowest Internet connection speed available, the network does not appear to be affected significantly. The increase percentages may not be very small but, given that they are in the order of milliseconds, they are insignificant.

Figure 27: 63 KB/s throttling - Arrival Time per block
5.1.2.3 Throttling Download & Upload Speed to 5 KB/s

Next, throttling to 5 KB/s was used. The results are presented in the table and figure below. As stated previously, this was more of a fun use case to see if the use of blockchain would be feasible, at least network-bandwidth-wise in the age of Dial-Up Internet access.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Results</th>
<th>Increase % from No-throttling network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Time</td>
<td>1 ms</td>
<td>±0%</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>19.15 min</td>
<td>+360,088%</td>
</tr>
<tr>
<td>Mean Time</td>
<td>2.94 s</td>
<td>+4,288%</td>
</tr>
<tr>
<td>Median Time</td>
<td>101 ms</td>
<td>+71.2%</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>5.42 min</td>
<td>+645,522%</td>
</tr>
</tbody>
</table>

This time, the low speed took its toll on the block arrival performance. As it can be observed, the time to equilibrium is, on average, four orders of magnitude larger!

Also, noteworthy, when trying to flood the network with new blocks, some nodes (different ones for different repetitions of the experiment) seemed to cap at CPU usage > 100%, resulting in the machine becoming unresponsive. This usually happened after producing ~60 blocks. As such, for completeness of the experiments, this particular use case was conducted in a different machine, equipped with a 6-core CPU clocked at 2.6GHz and 16GB of RAM.

In regard to the figure below, it seems that at such low bandwidths, the network exhibits a delay whose trend is linear. Most importantly, though, is the observation that there seems to occur a bottleneck effect after producing ~40 blocks.

As such it seems that at these speeds, this technology cannot be possibly used in production, even in a case such as the one on the experiment, where there is only a very limited number of mining authorities. However, this is to be expected, because bitcoin, like almost any other advancement in technology, is strongly dependent on the existing technologies and possibilities of its time.
5.2 Multi-node network with Multiple Hosts

In this use-case, the most common use of the software that was created was demonstrated. More specifically, **btc-network** was used to deploy a network of twenty nodes, one per machine, in a real network of peers. During the necessary preparations for deploying this network, that is, when deploying each succeeding node, it was observed, through some initial tests, similar to the ones that are discussed below, that the results were very comparable. As such, this final number of twenty nodes was considered appropriate enough for conducting the experiments and extracting meaningful results.

As such, this experiment is more closely related to the real network, —as it is itself a real-world network—. This time, the default node image was used, because there is not a need to simulate bandwidth and latency limitations, as they are inherent properties of the network.

5.2.1 Experiment Setup

The experiment setup closely resembles the one of the previous experiments, with some small modifications. To begin with, as before, first a network was deployed to the twenty peers, by running the appropriate commands, much like the ones...
described above. The only difference being that the inputting JSON file now consists of nodes with different IP addresses. Then, addresses were created for each node and some blocks were initially mined, in order to be able to create a large amount of transactions later on. Like previously, extra care was taken on circulating funds, in order to be able to run the experiment endlessly, if needed.

The infrastructure that was used in order to implement this network was provided by GRNET [51]. The National Infrastructures for Research and Technology is a country-wide cluster of machines that is used extensively for research. It provided a huge amount of resources for running different kinds of applications in the research field and is shared across almost all Greek Universities. Although it provides users with full virtual-machines, and not bare-metal hardware, for the purposes of this work, this is not significant, and it can safely be assumed that the network running in this infrastructure is a network of truly different machines. Specifically, each machine was equipped with a 4-core CPU clocked at 2.1GHz and 4GB of RAM.

Given the feasibility limitations, but also the observations from above, the scripts that were run, instructed the nodes to mine a block every 4 seconds, instead of ~10 minutes. This didn’t seem to hinder the results at all. The same process as before was followed, where 200 blocks were mined.

5.2.2 Results Acquired

After flooding the network with new blocks and transactions, the log files from each node were gathered and merged, and the resulting times were acquired. The results that occurred are the ones that are presented below.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Time</td>
<td>1 ms</td>
</tr>
<tr>
<td>Maximum Time</td>
<td>240 ms</td>
</tr>
<tr>
<td>Mean Time</td>
<td>34.4 ms</td>
</tr>
<tr>
<td>Median Time</td>
<td>26 ms</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>32 ms</td>
</tr>
</tbody>
</table>
As it can be observed, the results are outstanding! Quintupling the number of peers, on top of conducting the experiment on a real setting not only did not seem to affect the results at all, but it made them more consistent, as it can be observed by the statistics above.

![Figure 29: Multi-node, Multi-Host - Arrival Time per block](image)

Per the author’s opinion, this goes to show the potential of the technology.

### 5.2.3 Relevant Source Code

Given that a use case such as the one that was just described appears to be a common one; Given a network of nodes, one wants to execute some operations and then gather the relevant log files, the scripts that were used are also provided in the source code of the software of this thesis\(^\text{11}\).

For completeness of this essay, some examples of them are also provided in section A.3 below. Note that, these are made to be run on a *NIX (or compatible) Operating System.

\(^{11}\) https://github.com/iamnapo/btc-network/tree/master/scripts
Chapter 6
Conclusions and Future Work

Following the discussion and the results of the experimentation scenarios that were presented previously, this Chapter acts as a summarization of the work that was done in the context of this Thesis. Moreover, it further discusses its findings, as well as the possibilities for future work and extension of both the methodology and the development of the software tool.

6.1 Conclusions

In this work, a software tool aiding in rapid deployment of bitcoin blockchains was proposed, dubbed *btc-network*. This tool provides its users not only with a very robust and performant, as explained throughout this essay, underlying infrastructure that uses containerization, but also a, very friendly for the user, Command Line Interface that hides any complexity and automates every process.

Firstly, all the required theoretical and technological background that is need-to-know from the reader is explained in detail, as well as every piece of previous research that was either proven crucial for the inception of the context of this work, either provided a useful insight for the currently presented implementation, or, lastly, provided the author with valuable resources for standing upon and expanding.

Continuing, the formalization of the requirements of the software is catalogued. A step of utter importance when aiming to create a piece of software. As is explained, cornerstones of the implementation were the use of Open Source Software, the ease of use and lastly, but most importantly, the ability to easily extend what was built, by adding new features and functionalities. Moreover, the complete structure of the system is analysed, in order to make it understandable and reproducible. A structure that was built by having scalability in mind.

Lastly, a series of experiments were performed and explained in detail. These aim to first and foremost provide a tangible example of ways to use the created tool. Referencing the previous Chapter, firstly a way to deploy a network to a single host was demonstrated and then, a real word blockchain was deployed. Although the main
purpose of these cases was to demonstrate the use of **btc-network**, it also provided useful insights into the bitcoin technology itself.

Two noteworthy insights stood up. First of all, it seems rather safe to assume that testing hypotheses on a small network deployed in a single host can provide useful information about the behavior of a real-world network. Secondly, the technology of bitcoin is such one, that, it does not seem to be affected by the addition of new nodes, at least in the case of small-scale networks, making it scalable and easy to work with. Moreover, this may be a sign that these conclusions could carry on to large scale networks as well, making bitcoin an extremely scalable technology. However, examining the right-most part of Figure 28, and especially Figure 29, it seems to be affected by the ever-ending increase of the blockchain; as the chain gets longer, the arrival time seems to get slightly bigger. But, as seen from the results of the previous chapter, internet speeds seem to be very ahead in the curve, making it a non-issue, at least for the foreseeable future.

To conclude, in the context of this research, two main contributions for the community materialized. On the one hand, it proposed a new way of deploying Bitcoin nodes through system-level virtualization. Using this, it is possible to create a full node in almost any hosting operating system, with very little to no needed configuration at all, while also maintaining a strong safety net that separates the blockchain’s software from other resources. On the other hand, a software infrastructure in the form of a Command Line Interface was created, whose purpose is to bundle the aforementioned proposal into an easy to use tool.

It is the author’s opinion that these contributions will benefit the research community and ordinary users alike, where the need for automation and abstraction of complexity when deploying a peer-to-peer network appears to be common ground.

### 6.2 Suggestions for Future Work

As it can be easily understood, the extensions and capabilities that can be added to the system are largely based upon the needs of its users. Although it has been made an attempt to include the most common range of capabilities, there are certainly many that are not implemented, which could benefit its users.
Moreover, there are parts of the system’s structure itself that could be extended with new features. Some possible improvements are described below:

6.2.1 Graphical User Interface

Although it was argued previously that a Command Line Interface provides one of the easiest ways to interact with the system, a full-blown Graphical User Interface would certainly prove useful. Not only would it aid in the creation and deployment of a blockchain, but also, it would provide a shelter for different features that would be important for the users.

On one hand, when creating a new blockchain through a graphical interface, it is a lot easier to provide requirements such as the structure of the network as a graph. Like a real network, the graph that it is consisted of, is a very sparse graph, where some nodes are more popular than others. As such, it would make sense to have a graphical interface where users, for example, could connect the nodes themselves and create a network according to their own likening and needs.

On the other hand, having a graphical interface opens a much wider range of extensibility. For example, the addition to the system of analytics. That is, different windows with statistics and graphs about the network’s state. This could help users achieve a second layer of automation; Currently, the deployment of the chain is automated, but the interactions and measurements with it are not. By the use of this interface those, too, could be automated and sped up immensely.

6.2.2 Support for Network-wide commands

In Section 5.2 it became apparent that executing commands network-wide can be proven very useful. As such, it would make sense to create more options either in the Command Line Interface that currently exists or to a new Graphical User Interface, as described above. There are tools readily available that allow for command execution to multiple remotes, such as parallel-ssh\(^{12}\) who could be integrated into the

\(^{12}\text{https://parallel-ssh.org}\)
system, in order to help users with the execution of some of the most common functionalities that appear when experimenting with a blockchain. The collection of log files to a single place, is an example.

6.2.3 Support for Different Blockchains

This is likely the most obvious piece of future work that could be done. Even though bitcoin was the technology that paved the way in the field, other, quite impressive and innovative projects have arisen in the past few years. There has been an extensive amount of research on this field and, as such, adding the functionality of deploying different blockchains, would be of utmost importance.

6.2.4 Experiment Continuation

Even though the experiments of Chapter 5 were more of a use case than a direct point of this research, there are ways to expand and build more experiments on top of them, which may lead to new, interesting results. Most notable is the case of deploying a network using multiple hosts, where each host contains multiple nodes (i.e. a hybrid of Sections 5.1 and 5.2). Also, a custom base image could be created on top of iamnapo/btc-network:trickle, in order to experiment by using different throttling limits in each node in the network.

Lastly, bundling the operations and code that were used to extract measurements from the network could provide a very helpful steppingstone for even more experimental work.
Appendix

A. Local Compilation & Usage

Although there is readily available version of the tool online, it could be useful for some to compile it from source, so as to be used in a particular field with a non-standard development environment. This section describes ways in which this can be achieved. Also included below, are a set of some of the most commonly used commands, across different use cases.

A.1 Manual Installation

Manually installing the system on a computer involves the installation of all of the required technologies and packages, as described throughout the above analysis. In particular, the Node.js execution runtime as well as the npm package manager should be installed first. Then, after obtaining the necessary source code from the thesis repository, the `npm install` command will have to be executed via terminal, to install all the required dependencies.

Finally, the `npm start` command must be executed. An example of this order of execution is shown below.

```bash
# Clone the repository
git clone https://github.com/iamnapo/btc-network.git btc-network

# Navigate to the repository
cd btc-network

# Install dependencies
npm install

# Start the project
npm start
```

*Figure 30: Example instructions for local installation*

A.2 Creation Of A Binary Executable Bundle

Bundling the source code into a binary executable bundle is done with the use of pkg, as mentioned previously. To create an executable for each one of the three
most common operating systems, Windows, MacOS and Linux, the `npm run build` would suffice. Otherwise, a command in the form: `./node_modules/bin/pkg . -t <architecture>` must be executed, where available options for `<architecture>` can be found on pkg’s online documentation.\(^\text{13}\)

### A.3 Commonly Used Commands

Most commonly when experimenting with a blockchain, there is a need to extract, save and manipulate the logs of each individual node. The commands below showcase a way that this could be accomplished.

#### Figure 31: Example command to start a node via ssh

```
ssh <HOST> -l <USER> -i <RSA_KEY> 'ENDSSH'
export NVM_DIR="$HOME/.nvm"; [ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
btc-network -o <BTC-NETWORK_FOLDER> -r <NODE_ID>
ENDSSH
```

#### Figure 32: Example command to start saving logs to a remote node

```
ssh host -l user -i key "docker-compose -f \
<BTC-NETWORK_FOLDER>/btc-node-<NODE_ID>/docker-compose.yml \logs -f -t --no-color >> logs.log &"
```

#### Figure 33: Example command to stop a node via ssh

```
ssh <HOST> -l <USER> -i <RSA_KEY> 'ENDSSH'
export NVM_DIR="$HOME/.nvm"; [ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
btc-network -o <BTC-NETWORK_FOLDER> -s <NODE_ID>
ENDSSH
```

B. Command Line Interface UI

B.1 Network Creation

This section contains various screenshots from btc-network’s Interface flow for creating a new network.

Figure 34: Example command to stop saving logs on a remote node

```bash
ssh <HOST> -l <USER> -i <RSA_KEY> <<'ENDSSH'
kill $(ps aux | grep 'logs' | sed \$d | awk '{print $2}')
ENDSSH
```

Figure 35: Example command to get logs from a remote log locally

```bash
scp -i <RSA_KEY> -r <USER>@<HOST>:/path/to/remote/logs.log path/to/local/logs.log
```

Figure 36: Screen to provide node information in btc-network
Figure 37: Screen to provide base image information in **btc-network**

Figure 38: Screen to provide blockchain parameters information in **btc-network** [1]

Figure 39: Screen to provide blockchain parameters information in **btc-network** [2]
Figure 40: Screen to provide output information in **btc-network**

Figure 41: Creation success message in **btc-network**
References


    [Accessed 2019].

[46] "An Accelerated History of Internet Speed (Infographic)," [Online]. Available:


[50] "Loopback Address," [Online]. Available: