IIT DELHI

MASTERS THESIS

Improving security & performance of PoW Blockchains using Anchors

Author:
Shadab Zafar

Supervisor:
Prof. Vinay J. Ribeiro

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Technology

in the

Department of Computer Science and Engineering

June, 2019

CERTIFICATE

This is to certify that the thesis titled **Improving security & performance of PoW Blockchains using Anchors**, being submitted by **Shadab Zafar**, is a record of bona fide work carried out by him under my guidance and supervision. The work presented in this thesis has not been submitted elsewhere either in part or full, for the award of any other degree or diploma.

Prof. Vinay J. Ribeiro
Department of Computer Science and Engineering
IIT Delhi

June, 2019

Abstract

Improving security & performance of PoW Blockchains using Anchors

by Shadab Zafar

After their first public use in Bitcoin, Blockchains have taken the world by storm. Even though research has been done on alternative forms of consensus schemes and new algorithms have been proposed, Proof-of-Work (PoW) is still the most widely used approach. This popularity has also shed light on some of the drawbacks that the current PoW Blockchains suffer from. Since PoW consensus generates blocks at random time instants, the weight of the chain increases in an unsteady manner which leads to forks, increased chance of double-spend and selfish mining attacks. It also results in systems with low performance as the confirmation times increase and throughput decreases.

In this thesis, we present a new signalling mechanism (called Anchors) which aid in improving the security and performance of PoW blockchains, while adding minimal overhead to the system. We implement Anchors in the reference implementation of Bitcoin (the most widely used PoW Blockchain) as a proof of concept. We also create a testbed that allows running Bitcoin experiments and use it to evaluate our implementation of Anchors on a network of 114 Bitcoin nodes. Comparing the results with unmodified Bitcoin we find that anchors propagate much faster than blocks, aiding in quicker fork resolution times (and even preventing some forks from occurring), without adversely impacting block propagation times.

Acknowledgements

First, I'd like to thank Prof. Vinay J. Ribeiro, without whose guidance and supervision this work would not have been possible. Next, I'd like to thank my co-authors and lab-mates Ovia Seshadri and Aditya Kumar for discussing the finer details of the projects and helping me whenever I was stuck.

This work would've moved considerably slowly had it not been for the helpful folks in the Bitcoin community (Slack, IRC, StackOverflow, Reddit etc.) I've received help from many people, but I'd specially like to thank Gregory Maxwell and Luke Dashjr for taking time out of their lives and helping me figure out the peculiarities of the Bitcoin Core client.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

In this chapter we introduce the preliminaries: Section 1.1 begins with the definition of a blockchain and explains key terms adjacent to it. Section 1.2 describes Bitcoin - the first public PoW blockchain system and its reference implementation - Bitcoin Core. Section 1.3 lists some of the shortcomings of current PoW blockchains.

1.1 Blockchain Technology

Blockchains are usually explained by describing Bitcoin, a cryptocurrency which was its first and most popular public application. However, blockchain technology now encompasses a wider scope and are much more general purpose than Bitcoin would suggest. Other systems currently being researched which include blockchains in their implementation include voting platforms, supply chain management, smart contracts, etc.

There exists no formal definition of a blockchain which is widely accepted; with multiple definitions being used in literature, each with slightly different phrasing.

The simplest definition comes from Narayanan et. al. [1]:

A blockchain is defined as a linked list data structure, that uses hash sums over its elements as pointers to the respective elements.

In this view, a blockchain is a data structure where blocks (containers for storing records) are linked into a chain with the use of cryptographic hashes, and each new block stores a reference to its parent.

Another definition comes from a technical committee formed by the International Standards Organization (ISO) to standardize blockchain technology [2] which define blockchain as: *a shared, immutable ledger that can record transactions across different industries, [...] It is a digital platform that records and verifies transactions in a transparent and secure way, removing the need for middlemen and increasing trust through its highly transparent nature.*

Some keywords from the above definition require further explanation:

- *shared* - a blockchain is distributed among a set of nodes which participate in its maintainance and are connected in a decentralized manner, so there is no central party above others and all the nodes are equal in their capabilities.
- *immutable* - In the context of blockchains, immutability refers to the inability to change data which is that has been published onto the chain.
- *transactions* - A transaction is generally considered to be the basic unit of data stored on a blockchain. The contents of a transaction can change, depending on the application of the blockchain and the context in which it is being used.
- *transparent* - Transparency here refers to the fact that a blockchain and consequently its transactions can be reviewed and verified in their entirety by any participating node. In Bitcoin's case, since the blockchain is public, any one can trace the history of any unit of bitcoin.
- *middlemen & trust* - Due to blockchains' decentralised nature, there is no need for middlemen (typically referred to as "trusted third party") to maintain and verify their correctness. The removal of middlemen, coupled with transparency also ensures that there is no need for a node to trust any other node in order to verify the blockchain.

A blockchain can also be thought of as a distributed system, where the goal of participating nodes is to arrive at a consensus on the ordering of transactions in the chain. In this view, blockchains can be categorized into two broad types:

- **Permissionless:** The key property of this type of blockchain is that the set of nodes that take part in the consensus process (over the state of chain) is not known beforehand so anyone can join the network and participate in maintainance of the chain. These blockchains typically use a consensus algorithms such as Proof-of-Work, Proof-of-Stake etc.
- **Permissioned:** Contrary to the above, in this type of blockchain only a previously-known set of nodes is allowed to take part in the network. Furthermore, all the nodes do not have equal capabilities so for eg. only a restricted set of nodes might have the right to create or validate transactions etc. These blockchains typically use the PBFT (Pracrtical Byzantine Fault Tolerance) algorithm for consensus.

Note: In this Thesis, we restrict ourselves to Blockchains that use Proof-of-Work as their consensus algorithm. Bitcoin, explained in the next section, is the largest publically deployed blockchain and uses PoW.

1.2 Bitcoin

Bitcoin is a digital currency (also called cryptocurrency) that was first proposed by the pseudonymous [1] "Satoshi Nakamoto". Satoshi self-published the Bitcoin whitepaper in 2008 [3] and soon after, on January 3rd, 2009, the genesis block of the main Bitcoin blockchain [2] was created.

Although the technical primitives (cryptographic hash functions, asymmetric cryptography) have existed for a while, Bitcoin was the first concept to combine these building blocks with an incentive system (mining rewards), thereby creating the first distributed cryptocurrency and it is by far the most successful one in terms of market capitalization reaching a peak value of 320 billion USD in Nov 2017 [5].

1.2.1 Proof-of-Work Consensus

At the heart of Bitcoin lies it's ingenious proof-of-work based consensus scheme (also called "Nakamoto Consensus"). A proof-of-work (PoW) in general enables a prover to provide evidence to a verifier that the prover has invested computational resources (CPU, memory etc.) into a task. The idea of performing and providing a PoW for security reasons was developed and refined by Hal Finney, Adam Back and others [6, 7]

In Bitcoin, a new block is only accepted to the blockchain if it contains a valid PoW. Each new block points to an older one and can be seen as adding weight to that chain. Honest nodes agree that at any point in time only the longest blockchain is considered valid. In practice, this is implemented as the "heaviest chain rule" - the chain having the most weight as it has been the hardest to compute.

PoW as used in Bitcoin is based on a partial pre-image attack on the cryptographic hash function SHA-256. This process can be seen as trying to solve a computational puzzle to find a block header (and an extra nonce value) that satisfy the relation:

[1]It is unknown whether Satoshi is a person or a group of persons. Multiple people have tried to present "proof" that they're indeed Satoshi but there is still doubt.

[2]Satoshi seemed to prefer "block chain" over "blockchain" as shown by their comment in the original source code. [4]

$$SHA\text{-}256(SHA\text{-}256(block\_header)) \leq T$$

where T is the target space of valid solutions.

In practice this may mean something like ′all blocks whose hashes begin with 69 zeroes are considered valid′.

Proof-of-Work is based on the idea of 1-CPU-1-Vote and acts as a defense mechanism against "Sybil attacks" - where an attacker might create fake identities and gain a large influence on the network. [8]

1.2.2 Mining

Mining is the process of looking for a valid Block by performing PoW and reaching consensus on the current state of the blockchain. The nodes that are actively involved in searching for these solutions called *miners*. They are rewarded with units of the "mined" cryptocurrency (bitcoins) as a compensation for investing computational power into the overall security of the cryptocurrency. Bitcoin is a prime example of *permissionless* system, where anybody is allowed to connect to Bitcoin's peer-to-peer network and participate in the maintainance of the chain. Miners can join or leave the network at any time, increasing or lowering the mining power. As the mining power changes, the hardness of the puzzle needs to be adjusted to ensure that new blocks are generated at regular intervals (which is 10 mintes for Bitcoin).

While miners compete with each other for the mining reward, the Bitcoin network also has other nodes like a *full node* - which might not be mining, but still contributes by forwarding the messages, storing a full copy of the blockchain [3] and participating in the consensus protocol.

[3] The current chain over 200 GBs in size [9].

1.2.3 Bitcoin Core

The reference implementation of Bitcoin, known as Bitcoin Core [10], was first written in C++ by Staoshi Nakamoto and released to the public on 9 Januray, 2009 [11]. The development was later picked up by a community of developers, lead by Wladimir J. van der Laan (as of 2019), who continue to maintain the code on GitHub (a site that hosts open-source code repositories).

Most of Bitcoin Core is written in C++ and is available for all major operating systems like Windows, macOS and Linux [9].

Bitcoin Core includes four clients, three of which are command line-based:

- *bitcoind* - also called "bitcoin daemon" - is the main application that runs in background as a Bitcoin full node. It also contains a wallet implementation which can fully verify payments.
- *bitcoin-cli* - once the daemon is running in background, this application acts as an interface for controlling the daemon's behaviour. It works by sending appropriate RPC commands to the daemon.
- *bitcoin-tx* - this application provides a low-level interface for creating transactions (using the "raw transactions API" discussed in section 3.3.3)
- *bitcoin-qt* - this GUI based application acts as a user-friendly frontend for the bitcoin daemon. The application has been written using the Qt UI Framework [12].

The source repository also includes functional and unit-tests for the Core source code.

We now give some details on the design of Bitcoin Core:

Modes of Operation

The Bitcoin daemon can be run in three modes which differ in the blockchain they maintain and the consensus parameters they use:

- **Mainnet** - This is the default mode of operation, where the daemon connects to other nodes in the network and downloads the main Bitcoin blockchain acting as a full node by participating in the consensus protocol - verifying blocks and transactions.
- **Testnet** - The testnet is an alternative Bitcoin block chain, which is used for testing. Testnet coins are separate and distinct from actual bitcoins,

and are never supposed to have any value. This allows application developers run experiments without having to use real bitcoins.

- **Regtest** - The regression test mode runs a local-only daemon which does not connect to any other node by default. Furthermore, the consensus parameters for this mode are set such that blocks can be generated instantly (skipping any proof-of-work). This is meant for Bitcoin developers to test their code locally.

Testnet and Regtest modes are run by passing *-testnet* or *-regtest* parameters to the bitcoin daemon. In our experimentation testbed, we used the regtest mode of Bitcoin since we wanted to avoid performing proof-of-work computation and run a private network of nodes.

Protocol Interfaces

A running Bitcoin daemon has support for three protocols, each of which serves a distinct purpose and are exposed on a different TCP port as depicted in Figure 1.1.

- **Bitcoin Protocol** - This is the protocol that a Bitcoin node uses to send or receive data from other nodes in the Bitcoin network. It is a custom networking protocol built upon TCP, having its own message structures and types. [13] Some messages of the protocol are: *block*, *tx*, *addr* - each of which contain objects of a specific type.

- **REST API** - For clients that want to query the chain information, Bitcoin provides an unauthenticated REST API based on HTTP. [14] The API allows using binary, hexadecimal and JSON (which is human readable) formats for accessing the data. This API is read-only, in that it can be used to read data like blocks and their contents, but can not be used to modify the blockchain state.

- **RPC API** - The Remote Procedure Call (RPC) interface of a Bitcoin node can be used by other clients to send commands to it that control the behaviour of the node. [15] This interface is more powerful than the REST API since apart from allowing users to read data from the chain, it also allows them to mutate the chain's state. For eg. by making the *generateblock* call which creates a new block. Both requests and responses from this interface are in the JSON format. The interface can be password protected so only authorised users can send the commands.

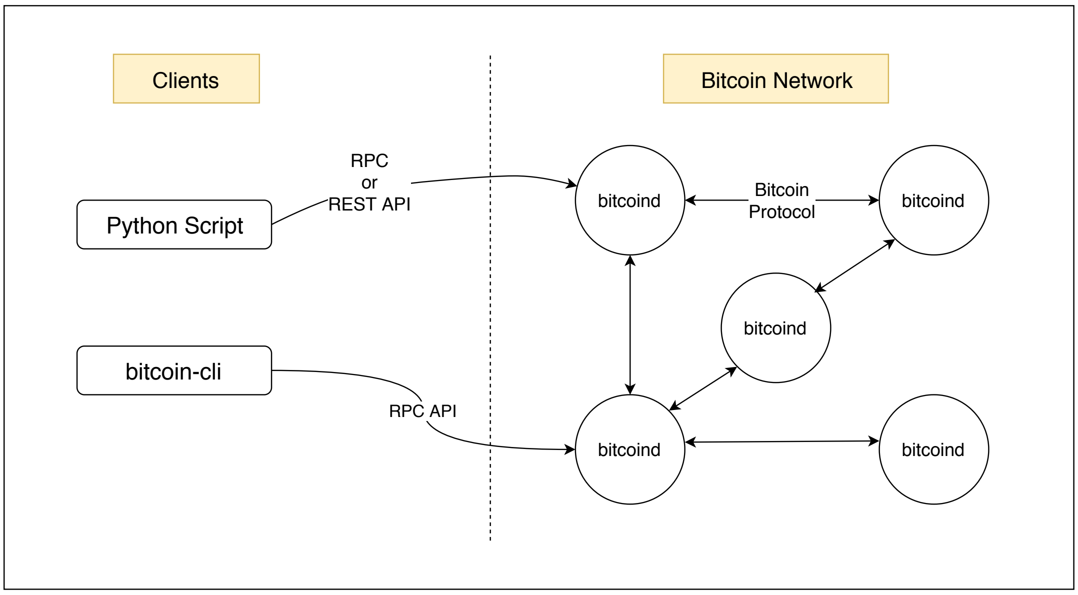

FIGURE 1.1: Protocol interfaces exposed by Bitcoin Core.
bitcoin-cli can only use the RPC API to send commands, while a Python script can use either of RPC or REST.

1.3 Problems with PoW Blockchains

As explained before, in a Proof-of-Work blockchain, every miner tries to solve a computational puzzle and the first one to do so gets to broadcast their block to the network, which the other miners accept and the block is added to the chain. Each block adds some weight to the chain depending on the difficulty of the puzzle that they solved. This strengthens the chain against future attacks. In cases where more than one miner solves the puzzle at the same time, a fork is created. When other miners observe these competing chains they extend the one with the most cumulative weight (also called the Heaviest Chain Rule).

Previous work [3, 16] has modeled the PoW consensus scheme as a Poisson process, which means that blocks get added to the blockchain at random time instants and the weight of the chain increase unsteadily. This causes various security and performance problems.

Forks - A fork is created when more than one miner create blocks pointing to the same parent. This can happen because they both ended up solving the puzzle simulataneously or perhaps one of the node was lagging behind the network and was not aware of the latest blocks, so ended up mining on an older block. These "natural" forks are distinct from forks that might be intentionally created by a malicious attacker. A fork is resolved when one of the branches of the chain gets longer (acquires more weight) than the others. As the majority of the network switches to heavier chain, the blocks in the shorter chain become *stale* and are essentially discarded. Miners who created the stale blocks lose their mining fees, which means their work was "wasted". During a fork there is uncertainty in the ordering of the blocks in the chain as the network hasn't agreed on a single chain. When forks take a long time to get resolved, the blockchain is more prone to attacks such as double-spending and selfish-mining [17, 18].

A successful **double-spending** attack allows an attacker to spend the same bitcoin twice. To execute this, the attacker creates two conflicting transactions that both spend the same bitcoin and broadcasts it onto the network. To guard against these attacks, users are suggested to wait for multiple blocks to arrive on the chain before accepting a payment or releasing the goods. This duration is known as the *confirmation time*.

Selfish-mining, on the other hand, is a block-withholding strategy where an attacker secretly mines a chain of blocks and does not broadcast it to the network. By suppressing this information the attacker tricks honest nodes into “wasting” their computational power by mining on blocks that already have successors (which are known only to the attacker). Once the attacker’s chain gets longer than the honest chain, blocks generated by the attacker become part of the eventually agreed-upon heaviest chain, while blocks generated by honest nodes are discarded. This attack is possible because an attacker with less mining power than the honest miners can occasionally generate a longer private chain.

One way to prevent the occurrence of natural forks is by increasing the block inter-arrival time as that gives the block a longer duration to reach all nodes in the network. In Bitcoin, the average time between generation of two blocks is kept to be 10 minutes to prevent natural forks from occurring; but this also leads to **reduced throughput** and **increased confirmation times**.

Throughput of a Blockchain may be measured as the number of transactions that get on to the chain per unit time (transactions per second). This depends on two main factors, the block size and the block inter-arrival time. Bitcoin has an average block size of around 1 MB while the inter-arrival time is 10 minutes which translates to around 7 to 10 transactions per second. This is orders of magnitudes less compared to traditional payment networks. For example, PayPal is capable of handling a few hundred transactions per second whereas VISA can process up to several thousand transactions per second. [19, 20]

To prevent double-spending attacks, Bitcoin users are suggested to wait till their transaction is atleast 6 blocks deep in the chain before releasing any goods in lieu of the transaction. This rule of thumb ensures that the chance of an attacker being able to overtake the main chain is less than a security 0.001%. This analysis assumes that the attacker has less than 10% of total hashing power [3, 16]. The confirmation time using this rule comes out to one hour, which is impractical for typical scenarios that require instant payment like buying a cup of coffee.

Chapter 2

Anchors

In this chapter, we describe the the key contribution of this thesis - Anchors. We begin with their structure and then discuss their lifecycle - how they're generated, propagated and processed (sections 2.1 and 2.2). We also describe some properties that Anchors have, which make them well suited to solve the problems listed in the last section (1.3). In section 2.4 we list some of the benefits that Anchors provide which are later backed by the experimental results in chapter 5.

2.1 Structure

An **Anchor** is a small bit string consisting of

i Hash of solution block in the main chain, called its *parent block*, and

ii a solution of a Proof-of-Work puzzle that is different (and easier) than that of a block.

Optionally, it may contain information to reward the miner for performing the work to generate an anchor. This can be done the same way as Bitcoin - by including a Coinbase transaction that creates fresh Bitcoins (in that it has no inputs) to reward the miner.

Note: Do note that Anchors contain no other information, specifically, no other transactions. So they have a considerably smaller body when compared to Bitcoin blocks.

In our implementation (see Chapter 3) an Anchor is identical to an empty Bitcoin block - one that contains no transactions apart from the Coinbase. Since anchors contain PoW, they (like blocks) also contribute weight to the chain they belong to.

2.2 Lifecycle

2.2.1 Generation

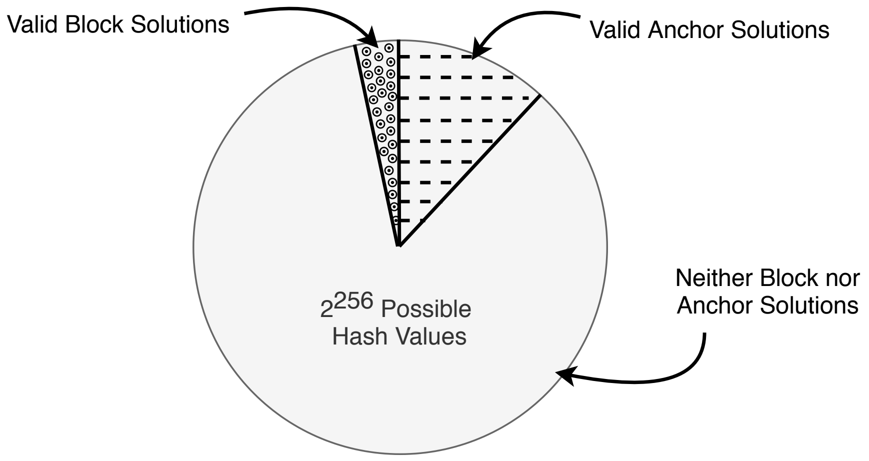
Valid Block Solutions
Valid Anchor Solutions
Neither Block nor Anchor Solutions
2^{256} Possible Hash Values

FIGURE 2.1: Block vs Anchor difficulty targets.
2^{256} values are possible when using the SHA-256 hash function.
A part of which contains valid Block hashes, while a larger & non-overlapping part contains valid Anchor hashes.

An Anchor is generated akin to a block but on a larger & non-overlapping as shown in Figure 2.1. A miner, when mining for a block checks for each nonce value in the header, if he has found a valid block. In case this isn't, the same solution can then be checked to see if it is valid anchor, in which case the miner publishes the anchor (along with the coinbase and a merkle path of the coinbase to the root of the tree.) *This method ensures that no additional effort is needed for mining an anchor*. The miner can simply continue the mining process in search of the next block (or anchor) on the chain.

The Merkle path that is additionally sent, is needed to prove to a receiver that the coinbase contained in the Anchor actually belongs to the Merkle tree of transactions whose root is stored in the header [3]. The value in the 'hash of the previous block' field of the published anchor is the hash-pointer to the parent block of the anchor. It is important to note that, the other values in the header like difficulty targets and the coinbase were originally intended for the next block and since a trial solution became an anchor these fields could not be updated.

2.2.2 Propagation

Once an Anchor has been found, it needs to be propagated throughout the network. Published Anchors propagate through the P2P network similar to transactions and blocks. The small size of Anchors (as compared to blocks) allows them to propagate faster throughout the network. The effects of including Anchors with respect to propagation times are discussed in section 4.2.1 while the results from the experiments are in Sections 5.1, 5.2.

2.2.3 Processing

A node upon receiving a valid Anchor updates the weight of the chain(s) [1] it belongs to. The node then forwards the Anchor to all its neighboring peers. If the revised weight introduced by the new Anchor causes a switch in case of a fork, as calculated by the Heaviest Chain Rule, the node shifts to the new chain tip and continues mining on that chain.

2.3 Properties

We now list some properties of anchors that make them suitable to solve the problems listed in section 1.3.

- **Higher Frequency:** Anchors are generated using a larger PoW target size than blocks. They are hence generated at a faster rate adding weight more smoothly to the chain than blocks.
- **Faster Propagation:** Anchors are fixed size, small structures containing minimal data (only the header and optionally coinbase with no other transactions). Hence, Anchors are light weight and propagate faster in the network than blocks.
- **Faster Verifcation:** Since Anchors do not contain transactions, they can be easily verified in O(1) time. A node only needs to check whether a received Anchor satisfies the PoW difficulty target. While, for a block, all transactions contained within it must also go through a verifcation process - checked for double-spends etc.
- **Non-Forking:** Anchors cannot create new forks in the chain, unlike blocks, as no block or anchor can point to another Anchor. Consequently, every successfully mined and published Anchor may contribute to the weight of the branch(es) its parent block is in.

[1]More than one chain is possible when the received Anchor points to an older Block and since then the chain has been forked

- **No extra hashing power:** Since Blocks and Anchors are simultaneously mined on non-overlapping target spaces as shown in Figure 2.1, Anchors do not require extra hashing power to generate.

2.4 Benefits

The above mentioned properties make Anchors a useful tool in solving various problems and lead to the following benefits:

- **Faster Fork Resolution:** Anchors provide a fast signaling mechanism for miners to know which chain has a majority of the hashing power of the network. In case of a fork, the system has to wait only for the first Anchor to resolve it, rather than the next block like Bitcoin. This matters because Anchors are generated at a higher rate than Blocks so arrive more frequently.
- **Increased Stability and Reduced confirmation time:** Anchors increase stability with a steady contribution to the weight of the chain which makes selfish mining attacks harder. Consequently they reduce the confirmation time of transactions and make double-spends more difficult.
- **Eliminates need to join Mining Pools:** Over time, Bitcoin mining has become centralized in that a few Mining Pools control a large portion of the mining power. Miners are incentivised to join a pool since Block solutions are scarce and being able to find one is hard for a lone miner. As Anchors are generated at a faster rate than blocks, these small miners can reap the anchor rewards more smoothly over time, thus reducing the need for them to join the pools.

Chapter 3

Implementation

This chapter begins with the exact steps required to building Bitcoin Core and then describes two central contributions of this thesis: (i) integration of Anchors in Bitcoin Core (section 3.2) and (ii) implementation of a testbed that can be used to run experiments (section 3.3).

3.1 Building Bitcoin Core

As mentioned before, the reference implementation of Bitcoin (known as Bitcoin Core) is written in cross-platform C++ and is available on all major operating systems like Windows, macOS and various flavours of Linux [9]. Since we had to make changes to the Core code we used the freely available source code of version 0.16 [21]. The code uses GNU Build System [22] to automate the build process. We now describe the process of compiling Bitcoin Core.

Installing Dependencies

The first step is to install all required dependencies, which varies from platform to platform. Listings 1, 2 show how to install them on Ubuntu 16.04 and Manjaro 18.0 respectively.

Compiling the source

Once all required dependencies have been installed, we download the source code from the official repository on GitHub and proceed to compile it. Listing 3 lists the steps.

```bash
#!/bin/bash

$ sudo add-apt-repository ppa:bitcoin/bitcoin
$ sudo apt-get update

$ sudo apt-get install build-essential libtool autotools-dev \
    automake pkg-config libssl-dev libevent-dev bsdmainutils \
    python3 libboost-system-dev libboost-filesystem-dev \
    libboost-chrono-dev libboost-program-options-dev \
    libboost-test-dev libboost-thread-dev libdb4.8-dev \
    libdb4.8++-dev protobuf-compiler libprotobuf-dev

$ sudo apt-get install libqt5gui5 libqt5core5a libqt5dbus5 \
    qttools5-dev qttools5-dev-tools libprotobuf-dev \
    protobuf-compiler libqrencode-dev
```

LISTING 1: Installing dependencies required for Bitcoin Core on Ubuntu 16.04
Commands taken from the original documentation [23].

```bash
#!/bin/bash

$ sudo pacman -S boost libevent qt5-base qt5-tools qrencode \
    miniupnpc protobuf zeromq db4.8
```

LISTING 2: Installing dependencies required for Bitcoin Core on Manjaro 18.0
Commands adapted from Bitcoin PKGBUILD file for ArchLinux [24].

```bash
#!/bin/bash

# Get the source from official repository on GitHub
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin

# We used v0.16 of the source
$ git checkout v0.16

# Disable parts of the code that we don't require to speed up compilation
$ ./autogen.sh
$ ./configure --disable-tests --disable-bench \
    --without-gui --with-incompatible-bdb

# Compile using 4 CPU cores
$ make -j4
```

LISTING 3: Compiling Bitcoin Core
Commands taken from the original documentation [23].

To speed up the compilation process we pass the following flags to *GNU Configure*:

- **–disable-tests** - Skip compiling tests written Bitcoin Core.
- **–disable-bench** - Skip compiling benchmarks.
- **–without-gui** - Skip compiling the Qt based GUI.
- **–with-incompatible-bdb** - Allow using a bdb version other than 4.8

3.2 Anchors with Bitcoin

We decided to implement Anchors in Bitcoin as it is the most widely used Proof-of-Work Blockchain. We used the stable version (v0.16 released on 26-02-2018) of the reference Bitcoin implementation (also called "Bitcoin Core") that was available to us before we started working on this project. For reference, the latest stable release of Bitcoin is v0.18 and was released on 18-05-2019.

In our implementation, an anchor is identical to a block excluding all transactions from its body except the coinbase i.e. anchors contain the block header and a coinbase transaction to identify and reward its creator.

As described in section 1.2.3 *Bitcoin Core* exposes a Remote Procedure Call (RPC) based API that other clients can use to control a Bitcoin node's behaviour. We implement Anchor support in Bitcoin by adding two new RPCs: **submitanchor** & **generateanchor**. We now explain the working of these RPCs.

3.2.1 submitanchor

The *submitanchor* RPC is modelled around the already existing *submitblock* RPC. It takes in as an argument a serialized representation of an anchor (which is generated outside of Bitcoin Core), ensures that it is valid, updates its local Blockchain, and then broadcasts the anchor to its peers where the process continues.

Validating an Anchor

Since we planned to test our implementation in a controlled environment; where all clients only send valid anchors, we have not implemented any validation checks for anchors. We only check whether an Anchor has already been seen before, in which case, we ignore such duplicate Anchors. In a real-world implementation, further checks like ensuring that the Block pointed to by an anchor has already been received etc. would have to be performed before an anchor is considered valid & broadcasted.

Updating the local Blockchain

When an anchor has been deemed valid, its effects need to be updated onto the local copy of the Blockchain.

Bitcoin Core represents the blockchain in memory as a 'parent pointer tree' - which is an N-ary tree where each node contains a pointer to its parent node, but no pointers are stored to child nodes. This structure has worked for Bitcoin's blocks as each Block either extends an existing chain or creates a fresh one (by forking it) so references to child nodes are never required. However, an anchor arriving on a block modifies the weights of all descendants of the block which requires us to also store children information in each block to be able to efficiently walk the "Block-tree".

Figure 3.1 depicts Bitcoin's original design of the Blockchain, whereas Figure 3.2 shows the updated design of the chain where blocks also store a list of children.

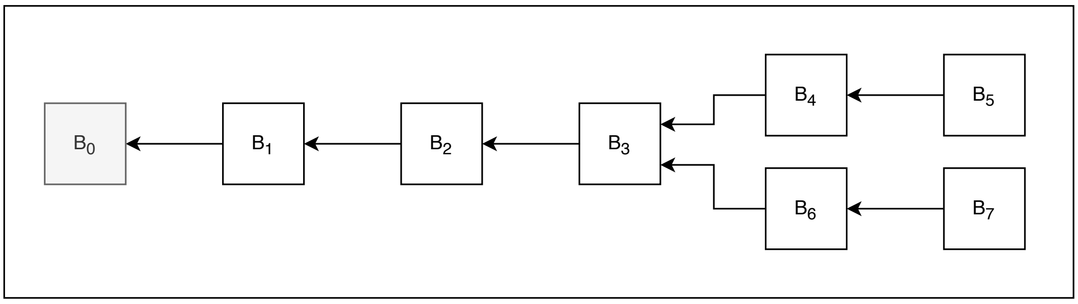

FIGURE 3.1: Bitcoin's original blockchain representation
Single pointed arrows depict that a block only stores a pointer to its parent.

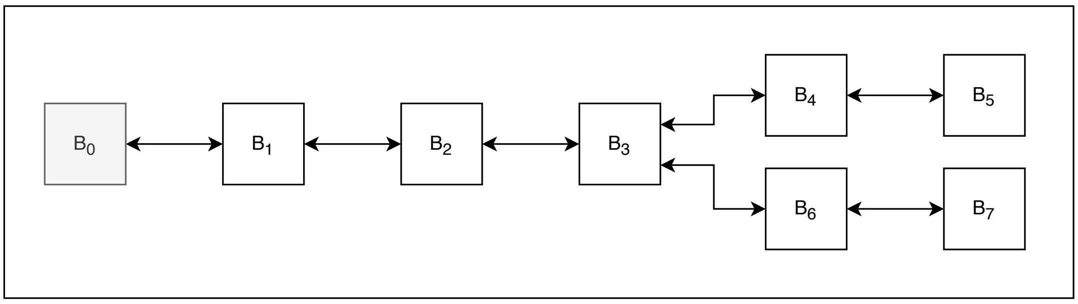

FIGURE 3.2: Modified blockchain representation
Double pointed arrows depict that a block stores pointers to its children as well.

Listing 4 shows the change made to the core data structure that stores the blocks in memory. Specifically, a new list of child pointers is added and is maintained along with the parent pointers.

Note: Do note that these changes have been made to the in-memory representation of a Block; they're only stored locally, and are not propagated across the network. Each Bitcoin node maintains this information locally.

```
1   class CBlockIndex // {
2       public:
3
4       // pointer to the hash of this block
5       const uint256* phashBlock;
6
7       // pointer to the index of the parent of this block
8       CBlockIndex* pprev;
9
10      // pointers to the children of this block
11      std::vector<CBlockIndex*> children;
12
13      // height of this block in chain
14      // (the genesis block has height 0)
15      int nHeight;
16
17      // total amount of work (expected number of hashes)
18      // in the chain up to and including this block
19      arith_uint256 nChainWork;
20
21      ...
22
23  }
```

LISTING 4: Updated CBlockIndex which also stores a list of child pointers. Line 11 has been added.
Snippet of code taken from Bitcoin Core source file *src/chain.h*.

We have not implemented storage / retrieval of anchors to disk because our tests created a fresh chain every time they ran. In a real-world implementation this would also have to be added so that changes can persist after events like client shutdown, machine restart etc.

Updating weight of the chain: An anchor updates the weight of the block that it points to and of all its descendants. This is done by performing a Breadth-first-search originating from the parent block and continuing down the tree.

Figure 3.3 depicts the scenario of an anchor arriving on a Block and the blocks whose weights would have to be updated.

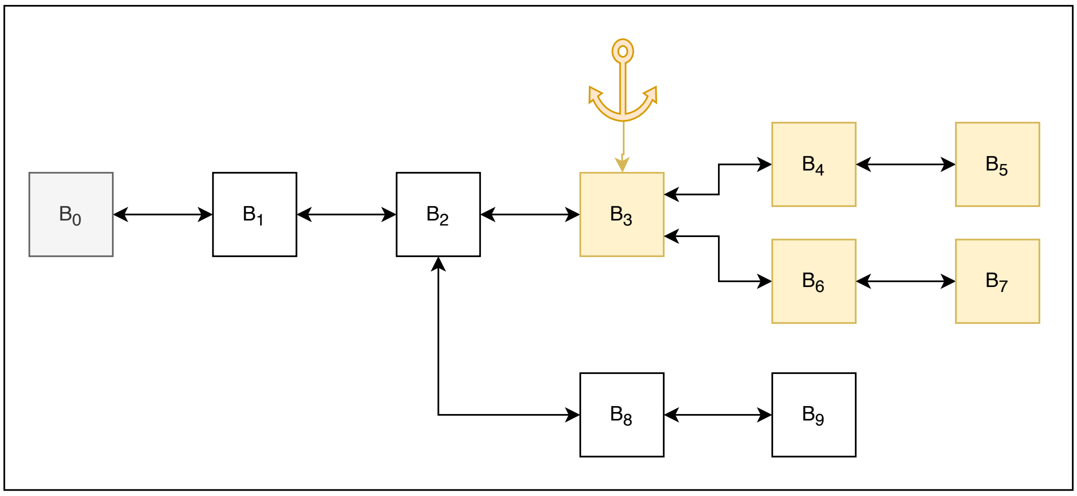

FIGURE 3.3: Anchor arriving on a Block updates its weights and of all its descendants (shown highlighted.)

The double pointed arrows depict that a block also stores pointers to its children.

Anchors-per-Block: We added a new parameter called *anchorsperblock* that controls the amount of weight that an anchor adds to the chain. The parameter is set once when the Bitcoin process starts. As an example, An *anchorsperblock* value of 10 specifies that anchor generation is ten times easier than block generation, so ten anchors are generated for every block and in such a case, an anchor adds 1/10th of the weight that a Block adds to the chain.

Brodcasting an Anchor to the network

The RPC call to submit an anchor happens on the TCP interface of Bitcoin Core which uses JSON as its data representation format. As explained in Section 1.2.3 the Bitcoin Protocol uses a separate interface to exchange messages between the network. We add a new message type that uses the same Bitcoin Protocol to send Anchor messages. See the source file *src/protocol.cpp* & *src/protocol.h* for the relevant code.

Anchors are propagated in the network by flooding, wherein each node broadcasts Anchor message to all its immediate neighbors (which process the anchor and further brodcasts it.)

Dealing with multiple threads: The two interfaces that Bitcoin Core provides (JSON-RPC & BitP) operate on separate network sockets and are handled by separate threads of the process. Data is passed among these threads via a signal & slot architecture, where each thread has handlers that listen on pre-defined slots, waiting for an event (a signal) to occur. When the signal fires, the slot handling function is executed. We follow the same design and create a new signal & the corresponding slot for Anchors called **AnchorConnected** which performs the actual broadcast of Anchor messages. See the source file *src/net_processing.cpp* & *src/validationinterface.cpp* for the relevant code.

3.2.2 generateanchor

Sending a serialized Block as an argument to the *submitanchor* RPC requires more bandwidth than is really necessary. To avoid wasting bandwidth, we created the the *generateanchor* RPC, which is modelled around the already existing *generate* RPC (which is used to generate a block).

While the *submitanchor* RPC receives an anchor's representation as an argument, the *generateanchor* RPC takes no arguments. When it is called, a new anchor is generated by the Bitcoin node itself, and is processed in the same way as explained in section 3.2.1.

3.3 Testbed

In this section we describe the implementation of a testbed that can be used to perform experiments on the Bitcoin blockchain. Some salient features of our testbed are:

- Spawn a network of Bitcoin nodes running on a cluster of cloud-based virtual machines.
- Replace Proof-of-Work with a centralized control program that triggers Blockchain events like generation of blocks & anchors.
- Emulate conditions that a real-world blockchain faces by genenrating a high throughput of transactions, introducing network latencies etc.
- Fine grained control over various parameters controlling the experiment.
- Process the logs generated by the Bitcoin nodes.

The Bitcoin Core code ships with a library (called *TestFramework*) that can spawn a network of Bitcoin nodes; but all those nodes can only run on a single machine. We saw this as a limitation as it prevented us from scaling up our experiments to hundreds of nodes. To remedy this, we extended the library by adding the ability to run Bitcoin nodes on a remote virtual machine (using the SSH protocol.)

Though Bitcoin Core is written in C++, the *TestFramework* library is written in Python. To be able to use the same codebase, we implemented our tested in Python (v3.7) as well. This allowed us to utilize the vast ecosystem of third-party libraries that Python has, some of which are:

- **paramiko** for connecting to remote machines using SSH.
- **socket, http.client** for estabilishing RPC connections to Bitcoin nodes.
- **json** for serializing the data being sent over the RPC connection.
- **random** for generating random events (like block or anchor generation etc.) that resembled an exponential probability distribution.
- **time** for "sleeping" between these events, instead of performing proof-of-work computation.

3.3.1 Centralized Controller

The core part of the testbed is a centralized program (called "Master") that controls the entire experiment. We now explain the various tasks that the Master program performs.

Network Setup

1. The Master program begins by launching the Bitcoin processes on remote virtual machines (VMs) using SSH. This can be configured by specifying the IP addresses of the VMs and the number of Bitcoin nodes to launch on each VM. Each Bitcoin process locks up some system resources like the port numbers it listens on for RPC / Bitcoin protocol connections, and the "data-directory" it uses to store its local blockchain & log files. To launch multiple Bitcoin processes on a single machine, we allocate a different set of such resources for each process by passing in arguments while launching the process.

2. Once all the processes have been launched, a TCP/RPC connection is acquired to each one of them. This connection is used to send RPC messages to the process to control its behaviour.

3. By default, a Bitcoin process has no idea of the other ones running in the network, so the Master program then connects these processes into a network according to a predefined topology by sending *addnode* RPC command. Further discussion of the network topology is deferred to Section 4.1.2.

Chain Setup & Transaction Generation

4. Once the nodes become aware of each other, the Master program generates empty blocks at each node so they have bitcoins to spend.

5. These bitcoins are later used to generate transactions that are used to fill up the Blocks that are generated. Secction 3.3.3 goes over this process in more detail.

Block & Anchor Generation

6. While generating a Block or an Anchor the Master program avoids doing actual proof-of-work computation and instead simulates the mining process by randomly choosing a node as the creator of a Block (or

Anchor) at random time instants. Since Block inter-arrival times in Bitcoin can be approximated by an exponential distribution [3], the random time instants are sampled from such a distribution with a predefined mean. The inter-arrival times between Anchors are sampled from another exponential distribution with a different mean.

7. This process continues for a a predefined amount of time (called the "lifetime" of the experiment) after which Block & Anchor generation is stopped.

Listings 5 and 6 depict the pseudo-code for generating Blocks & Anchors.

```
while True:

    # Wait for some time - sampled from an exponential distribution
    #                       to simulate the Poisson arrival of blocks.
    time.sleep(random.expovariate(1 / block_inter_arrival_time))

    # Choose a random winner for this round
    winner = random.choice(NODES)

    # Send the RPC command to the node that won
    # This only works because we run Bitcoin nodes in regtest mode
    block = winner.generateblock()
```

LISTING 5: Generating blocks using the *generateblock* RPC.
This simulates Bitcoin's random leader election, while skipping proof-of-work computation.

```
while True:

    # Wait for some time - sampled from a different exponential distribution
    time.sleep(random.expovariate(1 / anchor_inter_arrival_time))

    # Choose a random winner for this round
    winner = random.choice(NODES)

    # Send the RPC command to the node that won
    block = winner.generateanchor()
```

LISTING 6: Generating anchors using the *generateanchor* RPC.
Since Anchors are mined on a different (and easier) target than Blocks, they have a lower inter-arrival time but the process is akin to the leader election done for Blocks. Here too, we skip performing any proof-of-work.

Log Processing

Each Bitcoin process logs all the events that happen on the node into a log file stored in the "data-directory". For all the new code that we added to Bitcoin we also added suitable log lines, such as to log when an Anchor is generated, when it is received at a node, when a node switches from one branch of the chain to another because of Anchors etc. Once the experiment finishes, the Master program fetches logs from all the nodes, extracts relevant information that is of use to us and generates a result file for the experiment which contains the various statistics that we measure.

Cleanup

After the logs have been processed, the Master program stops all the running Bitcoin processes, deletes the data-directories and removes all manually introduced delays from the nodes so that the next experiment can start from a clean slate.

3.3.2 Architecture

We went over different architectures for our testbed, starting with the most straightforward - where the Master program handled the bulk of the load during the experiments, and later moving on to a setup where the load was distributed amongst multiple processes running on different machines. We now go over details of various iterations of our architecture, explain what problems we faced and what we did to resolve them.

Iteration 1

We began with a setup where the Master was run on one virtual machine (VM) and it connected to Bitcoin processes running on other VMs via a TCP-RPC connection.

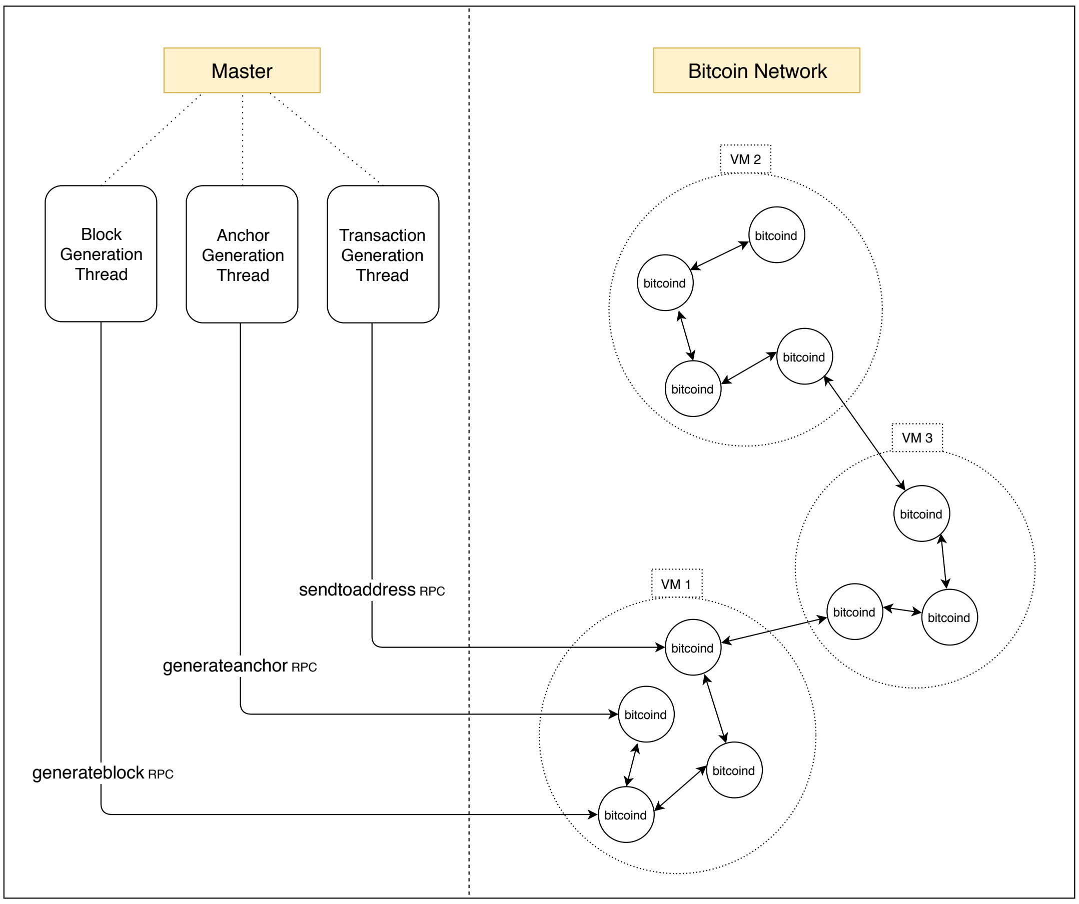

FIGURE 3.4: First iteration of testbed architecture
The "Master" program controls generation of Blocks, Anchors and Transactions.

Figure 3.4 depicts the architecture: On the left is the Master program (running on a single VM) and on the right is the rest of the Bitcoin network running on a cluster of VMs, each of which runs multiple Bitcoin nodes which have been connected to each other according to a predefined topology.

The Master program consists of three separate threads of execution that deal with generation of Blocks, Anchors & Transactions. These threads work by randomly picking a Bitcoin node from the network and sending an appropriate RPC command to it, and then waiting for some time before sending the next command.

The main problem we faced with this architecture was trying to generate a high throughput of transactions (which was required to have fully filled Blocks.) We observed that as we tried to increase the throughput, a majority of the RPC commands would fail due to socket timeouts, leading us to believe that the Master was creating a bottleneck in the system.

Iteration 2

We decided to extract out the transaction generation thread from the Master program, convert it into a separate script and move it to each of the individual VMs.

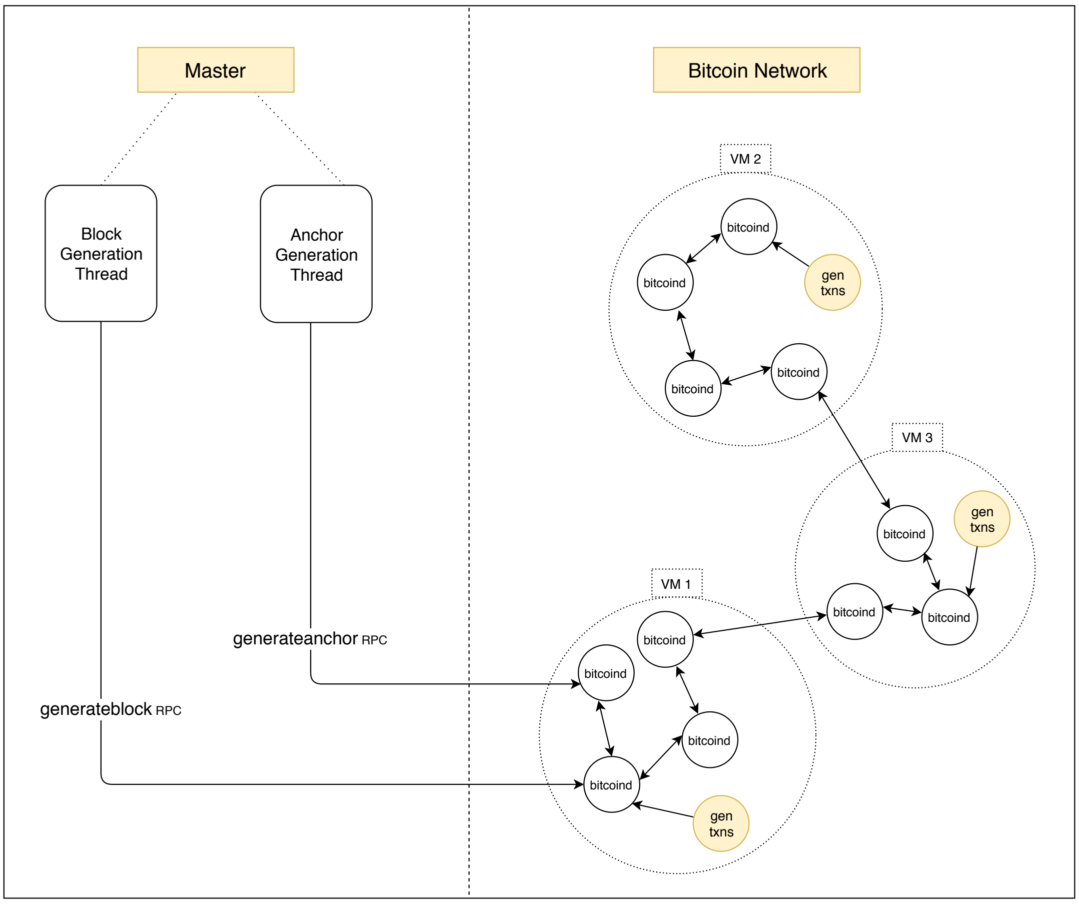

FIGURE 3.5: Second iteration of testbed architecture

Transaction generation thread has been moved to each VM to increase throughput.

Figure 3.5 shows the updated architecture, where a "Generate Transactions" script is run on each VM. The script only connects to Bitcoin processes running on its own machine so that all socket connections act as local (or loopback) connections. This reduces the network overhead of sending RPC commands from the Master program.

This architecture fixed the problem with transaction throughput, and we continued to use it until we needed to scale our setup to a large number of nodes (100+) with the experiments having shorter inter-arrival times between the various events (Blocks, Anchors, Transactions). When we tried to generate an anchor every second, we observed that some of the RPC commands would fail due to socket errors.

Iteration 3

Similar to what we did before, for our final iteration of the architecture, we moved the Block and Anchor generation threads from Master to the individual VMs.

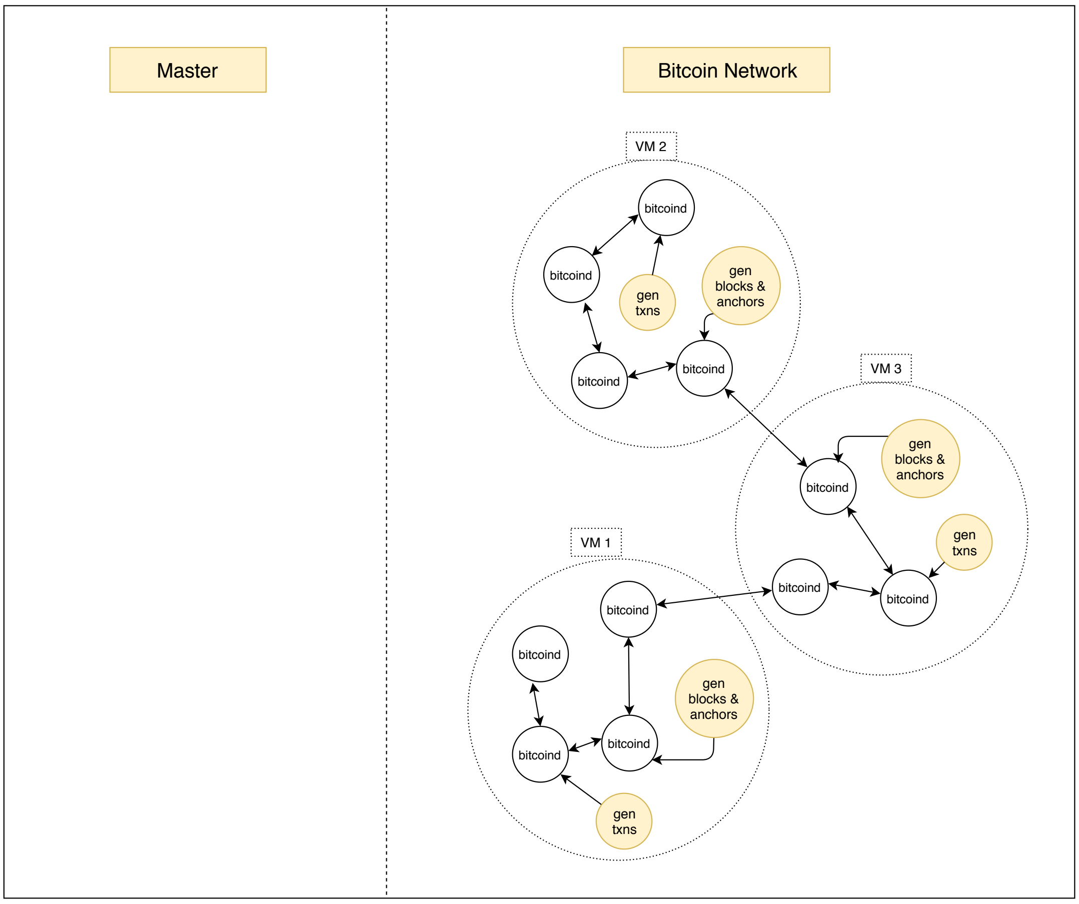

FIGURE 3.6: Third iteration of testbed architecture

Block & Anchor generation threads have also been moved to each VM.

Figure 3.6 shows this architecture. Apart from a transaction generating script, a block and anchor genenrating script has also been added to each VM, that connects to Bitcoin processes running locally and sends *generateblock* or *generateanchor* RPCs at random time instants.

The master program is now only responsible to setup the network, and launch its various components. After which, it sits idle waiting for the experiment to end. We've used this architecture for our final set of experiments.

3.3.3 Transaction Throughput

In the real-world, Bitcoin blocks contain transactions which are generated by millions of people throughout the world when they try to make payments for various goods and services. To emulate this process in our experiments, we generated dummy transactions between various Bitcoin nodes in the network, where one node randomly picks another and sends it a few bitcoins.

To keep our testbed as close to the real-world as possible, we wanted to generate enough transactions per second to fill up all Blocks that are generated. A typical Bitcoin block is of 1 MB in size whereas the smallest transaction, which transfer bitcoins from one node to another (and hence contains one input and one output) is roughly of 300 bytes, so around 3500 transactions are needed for every Block. This is easier when the inter-arrival time between blocks is larger. In Bitcoin, a new block is generated (roughly) every ten minutes, which is enough for transactions to be generated to fill the Blocks. However, we wanted to test scenarios where the block-interval time is short (say in the order of ten seconds) so we had to try various approaches to increase the transaction throughput of the system.

sendtoaddress RPC

As mentioned in Section 3.3.2 we started with an architecture where the Master program genereated all the transactions by sending *sendtoaddress* RPC to one of the nodes at random time intervals. Listing 7 shows the Master's transaction generating thread. We sample two nodes at random, arbitrarily assign one of them to be the payer and another to be the payee. The *sendtoaddress* RPC is sent to the payer, which generates the transaction and broadcasts it throughout the network.

The *sendtoaddress* RPC creates the simplest form of Bitcoin transaction, having exactly one input and exactly one output. As mentioned above, we

```python
while True:

    # Select two random nodes
    payer, payee = random.sample(NODES, 2)

    payees_address = address_of[payee]
    amount_to_pay = 0.01

    # Send the RPC command to the node that will pay
    payer.sendtoaddress(payees_address, amount_to_pay)
```

LISTING 7: Generating transactions using the *sendtoaddress* RPC. One node sends a token amount of bitcoins to another.

need 3500 of such transactions per Block, which for a Block inter-arrival time of 30 seconds comes out to around 120 transactions per second.

In practice we were only able to generate around 10 transactions per second, which is only one-tenth of what was required. Even after we moved to a setup where the transaction generation process was remove from Master and distributed to the remote virtual machines, we did not observe any improvement in transaction throughput.

Apache JMeter

At first, we attributed this low throughput of transactions to our implementation of the Master program, in particular the transaction generating thread, which was in Python and used Python's in-built socket & http libraries. We felt that using a compiled language might give us better results, but since porting the code to another language would take up a lot of time, we instead decided to use an already existing application that would be well suited for this task - Apache JMeter [25] - which is a Java based open-source application that is used to load-test and measure performance of other applications. It supports a wide variety of web based protocols like HTTP, FTP, Generic TCP, LDAP etc.

We used JMeter in a distributed setup [26] since it was appropriate for our setup. We ran JMeter server on one of the VMs (which would act as master) and JMeter clients on other VMs (acting as slaves.) We created a JMeter test plan which uses the *HTTPSamplerProxy* class to send JSON-RPC commands to the Bitcoin nodes running on the VM. JMeter allows the load on machines to be configured via multiple methods. We used a *ThreadGroup* with 100 threads running on each VM with a "ramp up" time of 0 seconds, which essentially means that we expect 100 transactions to be generated per second, but in practice we observed similar results as before - only 10 transactions per second were actually being generated, which meant that the low throughput issue is not with our implementation of Master but instead lies within Bitcoin itself.

JMeter logs the time each RPC command took to finish execution and return its result. Analyzing those logs gave us insight into the real problem. We observed that the inital RPC commands executed faster taking only 0.5 seconds to generate a transaction, while as time went on the response time increased, with some of the calls taking as much as 100 seconds to generate a transaction.

To understand the root cause of the problem, we read the source code involved in generating a single transaction. We found that the *sendtoaddress* RPC is fundamentally limited in the way it chooses the bitcoins to spend to generate a transaction and as the mempool becomes larger new transactions are generated slowly. This has not been a problem for the main Bitcoin network because the transaction throughput of the system is low by design.

While going through the source code of Bitcoin's RPC server implementation (located in *src/httpserver.cpp*) we also discovered some parameters (passed onto the process when it is launched) that control behaviour of the RPC server:

- **rpcthreads** - the number of threads to service RPC commands. By default 4 threads are run, but increased them to 8.
- **rpcworkqueue** - the depth of the work queue used to service RPC commands. The default value is 16, but we increased it to 64 because we found that further increasing it didn't improve performance.
- **rpcservertimeout** - timeout during HTTP requests. We left this to its default value of 30 seconds since we want each request to succeed during that time.

We want to generated transactions to be able to simulate real-world load that the Bitcoin network faces. We also wanted the blocks that were generated by us to be full (1 MB in size). Since *sendtoaddress* couldn't work for our use case, we looked for other ways of creating transactions.

Raw Transaction API

The "raw transaction API" was introduced in Bitcoin version 0.7. It gives users low-level access to transaction creation and broadcast. The API has a set of RPCs that can be used to create a transaction: *createrawtransaction*, *fundrawtransaction*, *signrawtransaction*, *sendrawtransaction*.

Listing 8 shows the right order of executing these calls.

```
while True:

    # Select two random nodes
    payer, payee = random.sample(NODES, 2)

    payees_address = address_of[payee]
    amount_to_pay = 0.01

    # Send all RPC commands to the node that will pay
    r = payer.createrawtransaction([], { payee_address: amount_to_pay })
    f = payer.fundrawtransaction(r)
    s = payer.signrawtransaction(f["hex"])
    t = payer.sendrawtransaction(t["hex"])
```

LISTING 8: Generating transactions using the "raw transaction" API. Four RPC commands are required to create one transaction.

sendmany RPC

While the *sendtoaddress* RPC can only send bitcoins to a single address, the *sendmany* RPC is used to send bitcoins to multiple addresses in a single transaction. A transaction paid to a single address is of 300 Bytes, but one having multiple addresses can be made arbitrarily large by increasing the number of output addresses. Listing 9 shows our usage of the *sendmany* call. We select a random set of nodes that are paid a token amount of bitcoins.

```python
while True:

    # Select a random node as payer
    payer = random.choice(NODES)

    # Select a random set of payees having a size in the range [5, 50)
    payees = random.sample(NODES, random.randint(5, 50))

    amount_to_pay = 0.01

    # Send the RPC command to the node that will pay
    payer.sendmany("", { address_of[p]: amount_to_pay for p in payees })
```

LISTING 9: Generating transactions using the *sendmany* RPC.
Can be used to create transactions of arbitrary size, since they have multiple output addresses.

We finally used the *sendmany* RPC instead of the *raw-transaction* API as it only involves sending one RPC command instead of four to generate a transaction. A typical transaction generated by our implementation was of around 5 KBs which means we only require 200 transactions per Block or 7 transactions per second (assuming Block inter-arrival time of 30 seconds) which is well within reach of Bitcoin's RPC server.

Chapter 4

Experimental Evaluation

In this chapter, we describe the evaluation of our implementation of a Bitcoin system which includes Anchors. We begin by outlining the setup we used (section 4.1) - configuration of the machines and the topology used to connect them into a single network and then move on to describing the experiments we performed (section 4.2).

4.1 Setup

4.1.1 Baadal Machines

We setup our testbed on a cluster of 20 cloud based virtual machines. We used IIT Delhi's private cloud infrastructure (managed by an orchestration software called "Baadal") for this purpose. The configuration of these machines are as follows:

- **Operating System :** Ubuntu 16.04
- **RAM :** 8 GB
- **Hard Disk :** 80 GB
- **CPU :** 8 virtual cores of Intel Xeon CPU E5-2680 v3 @ 2.50GHz

We reserved one machine for running only the Master program while the other nineteen machines ran six Bitcoin nodes each, for a total of 114 (19 x 6) nodes. The other machines also ran a transaction generation script and a block & anchor generation script.

4.1.2 Network Topology

As explained in section 1.2.3 we ran the Bitcoin nodes in *regtest* mode. In this mode, the nodes do not automatically search for their peers, and have to be manually connected into a topology. We use the *addnode* RPC to create a bi-directional connection between two Bitcoin nodes. Since the links are symmetrical, the RPC can be sent to either of the two nodes.

The structure of Bitcoin's topology is complicated, and is intentionally hidden to prevent denial of service (DoS) attacks and maintain the privacy of nodes. There are two main RPCs that provide network related information *getnetworkinfo* and *getpeerinfo*, but they do not reveal the immediate neighbors, but do provide a superset of nodes that they have discovered. The latency among the nodes is also unknown.

There has been some previous research that shed light on some of the details of Bitcoin's topology [27], like the fact that degrees of nodes can be approximated by a power-law distribution. The approach used by this work was patched in version 0.10.1 of Bitcoin (released in April 2015) [28]. Though there have been other approaches [29] that tried to reveal more information about the topology, we are not aware of any detailed work that offers significant insight into the Bitcoin main network.

Similar to other past work [30] that emulated Bitcoin in experimental settings, we construct a network where each node is connected to 5 other nodes chosen at random.

Previous work [31] has found that the Bitcoin's main network is connected, so each node can reach all others. We too ensure that that our generated topology is connected by checking that the network graph has a single connected component. We also computed the diameter of our network of 114 nodes to see the maximum number of hops it took for messages from one node to reach another node. We found the diameter varied between 3 and 5. We used a Python module called *networkx* [32] for performing both the connected components check and computing the diameter.

4.1.3 Network Latencies

Since we used virtual machines (VMs) located in a single computing facility, the inter-machine latencies were negligible, so any messages sent across this network would reach instantaneously. In order to mimic real-world conditions within the network, we introduced manual delays between the nodes.

We marked each VM to represent a major city in the world, and each link between two VMs has corresponding inter city latency taken from Global Ping Statistics [33]. To get a wide geographical distribution of nodes, we chose cities from different time-zones. The exact values are given in Figure 4.1.

Linux NetEm

The Linux kernel provides a network emulator package called *NetEm* [34] which is an enhancement of the Linux traffic control facilities and provides functionality for testing applications by emulating the properties of wide area networks such as variable delay, loss, re-ordering etc. [35]

NetEm is controlled by the command line tool *tc* which is part of the *iproute2* package of tools. Setting a simple delay between two nodes requires executing multiple *tc* commands with different arguments.

So instead of *tc*, we used the Python based *tcconfig* package [36] which is a wrapper over the *tc* command and provides a cleaner API to control the delays. The package provides multiple commands like *tcset* to set delays, *tcshow* - to see the currently set delays and *tcdel* to remove them.

Listing 10 shows the pseudo-code for how delays were set across the network. We use the delay values from table given in Figure 4.1.

```python
# Iterate over all VMs and their neighbors
for source_vm in all_VMs:

    for dest_vm in neighbors_of[source_vm]:

        # Lookup the delay between the two machines from
        delay = delay_between[source_vm][dest_vm]

        # tcset command requires the IP address of the destination machine
        dest_vm_IP = IP_of[dest_vm]

        # Command to set a fixed delay between all packets
        # going from "source VM" to "destination VM"
        # ens3 is the name of the LAN interface that the VMs use
        command = f"tcset ens3 --add --delay {delay}ms --dst-network {dest_vm_IP}"

        # Use SSH to execute the command on the source VM
        ssh.execute(command, on=source_vm)
```

LISTING 10: Setting delays throughout the network using *tcset* command from the *tcconfig* package.
SSH is used to execute the *tcset* commands on all VMs.

City	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Adelaide (1)	0	322.478	324.752	217.291	158.629	221.814	301.07	240.907	359.966	443.586	241.13	318.534	179.417	339.143	396.119	228.227	327.753	19.457	230.093	328.809
Amsterdam (2)	322.408	0	229.02	98.18	215.336	190.816	13.271	81.857	71.874	161.2	76.24	12.127	141.875	239.652	298.991	164.633	25.048	286.033	87.729	15.269
Bangkok (3)	318.713	216.251	0	251.168	67.833	202.647	247.368	252.748	243.547	111.188	286.555	204.941	193.722	153.847	230.445	65.303	306.793	217.937	273.725	200.744
Chicago (4)	216.845	97.568	274.366	0	213.579	98.319	87.777	23.371	149.923	221.461	23.475	96.012	51.536	198.451	287.763	223.594	112.26	201.674	16.294	116.801
Hong Kong (5)	158.519	215.473	68.942	219.217	0	129.538	242.317	196.769	279.746	250.772	223.08	281.747	156.216	120.274	210.75	33.895	220.697	158.827	244.335	270.929
Honolulu (6)	221.763	189.393	235.195	98.295	129.377	0	180.113	117.127	222.317	253.609	117.076	191.236	59.781	198.957	273.476	189.597	200.265	202.78	116.792	194.694
London (7)	301.743	13.097	259.359	86.029	242.261	180.063	0	80.87	51.311	138.12	75.404	5.186	161.183	292.729	247.463	172.112	25.601	281.339	90.545	25.198
Montreal (8)	240.935	81.883	274.188	24.029	196.744	117.096	80.795	0	127.48	236.042	9.459	81.731	72.789	273.287	238.298	234.766	106.4	258.365	8.287	96.553
Moscow (9)	359.862	71.974	259.511	142.76	279.799	222.334	51.424	127.191	0	182.3	131.194	48.871	195.967	347.679	262.81	189.566	19.086	346.504	127.585	51.951
New Delhi (10)	443.114	161.511	130.483	222.377	250.787	253.687	138.195	241.518	182.245	0	207.956	145.391	264.517	401.264	421.59	70.416	173.42	289.307	233.755	158.673
New York (11)	241.1	76.191	297.35	22.569	218.884	116.982	75.443	9.442	131.177	207.915	0	74.249	70.354	247.133	234.047	247.402	95.822	213.626	11.978	103.882
Paris (12)	313.009	17.616	203.92	91.485	244.683	194.32	4.137	82.348	49.216	173.393	84.873	0	144.178	264.478	259.406	245.792	30.398	279.41	87.528	13.669
San Francisco (13)	179.411	142.018	234.571	51.785	156.318	59.72	161.197	72.772	196.008	264.446	70.331	144.255	0	167.629	164.349	168.186	164.309	152.079	63.375	166.86
Shanghai (14)	368.599	257.577	241.753	335.279	132.536	206.723	316.9	72.772	250.068	400.41	242.636	294.528	167.619	0	30.528	168.186	264.322	332.21	259.428	166.86
Shenzhen (15)	368.599	330.459	241.753	335.279	132.536	206.723	316.9	72.772	250.068	400.41	242.636	294.528	167.619	0.044	0	168.186	264.322	309.734	259.428	166.86
Singapore (16)	228.315	164.687	51.543	220.171	34.208	189.522	172.151	235.201	189.6	65.057	247.449	242.467	168.158	265.77	390.675	0	188.129	93.93	241.711	170.708
Stockholm (17)	327.663	25.06	314.189	112.901	222.678	200.122	25.647	106.458	19.071	173.567	95.956	28.783	164.304	266.964	241.32	188.123	0	308.59	109.8	33.014
Sydney (18)	19.364	286.342	203.229	202.69	158.421	202.659	281.414	258.239	346.489	289.99	213.652	279.966	152.022	323.34	307.296	92.903	308.34	0	213.521	309.075
Toronto (19)	230.057	87.712	287.936	15.838	242.712	116.867	90.439	8.295	127.657	233.971	11.985	87.653	63.415	248.021	298.331	241.611	109.85	213.545	0	129.925
Zurich (20)	328.837	15.328	219.488	115.708	270.982	194.598	25.329	96.507	52.061	158.741	103.925	13.534	166.92	292.636	218.718	170.628	33.069	308.965	129.792	0

FIGURE 4.1: Ping latencies between world cities in milliseconds.
Data taken from Global Ping Statistics [33].

4.1.4 Network Time Synchronization

Network Time Protocol (NTP) is client server based networking protocol used to synchronize clocks between computer systems.

Since most of our experiments deal with time calculations, it was critical that we don't let clock drift affect the results. To ensure this, we let the default NTP client provided by Ubuntu run on all nodes which were setup to synchronize the time using IIT Delhi's internal NTP server [37].

4.2 Experiments

In this section we describe the experiments that we performed on our testbed to compare unmodified Bitcoin with a Bitcoin system that also includes Anchors.

4.2.1 Propagation Times

Propagation time of a message is the difference between the time it is broadcasted from the source node and time it arrives at the destination. Since an Anchor is considerably smaller than a Block, we expect it to propagate faster throughout the network thereby acting as a signalling mechanism for the true mining power in the network.

Each Bitcoin node logs the time when a Block or an Anchor is generated and when it is received. For each Block (or Anchor) we compute the difference between the time it was generated a node and when it was received at all other nodes. The maximum of these values is the total time taken by the Block (or Anchor) to propagate throughout the network. We observed that taking the maximum value results in outliers, as some nodes always lag behind the others. To prevent the actual results from being lost in outliers, we took 95 percentile values for propagation times.

4.2.2 Resolution Time of Forks

As explained in Section 1.3 Forks are a problem in a blockchain system as they reduce security and waste computation. Consider the scenario of a blockchain in the absence of Anchors. After a fork occurs, the weight of the chain remains constant in the interval between the time of creation of the fork and the arrival of the next block on any of the two branches. So the fork lasts for one block interval time, which can be substantial as for Bitcoin (10 minutes). While, in a blockchain system with Anchors, forks can be resolved by the first arriving Anchor.

Figure 4.2 depicts a scenario where a fork is caused by Block B_2 and occurs at time $T_0 + 5$. In a system without Anchors, the fork would be resolved at time $T_0 + 9$ with the arrival of Block B_3, whereas in a system with Anchors, the fork would be resolved earlier - at time $T_0 + 6$ with the arrival of Anchor A_1.

Resolution time of forks (RTF) is calculated by every miner as the difference between the time a fork was realized on its chain and the time at which

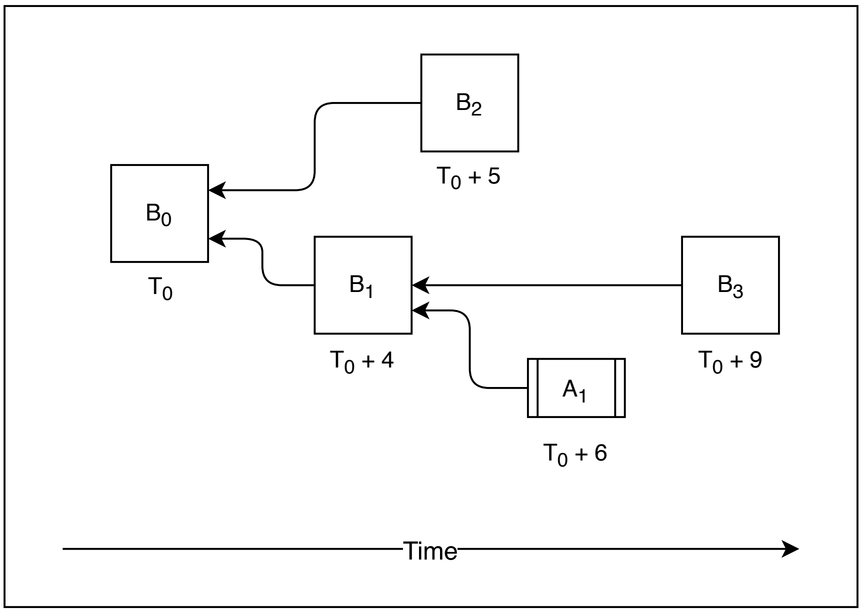

FIGURE 4.2: Fork Resolution in a Blockchain

the next block or anchor increased the chain weight on one of the branches of the fork, effectively resolving the fork. We then compute the mean RTF value of all forks that occur during the experiment.

In Bitcoin, the block interval time is set particularly high to make forks rare events. Confirmation times and resolution time of forks increase due to this. We are interested in finding the average resolution time for each fork across all nodes, and the branch of the fork they choose locally to continue mining on. Therefore, in order to get good statistical estimates in short experimental runs, we scale down the block interval of Bitcoin from 600 seconds to 30 seconds, so that a small number of natural forks occur based only on network delays.

Apart from Fork resolution times, we also keep track of the chain that a fork eventually gets resolved to. So for eg. in Figure 4.2, as one miner resolves the fork in favour of the branch containing Block B_1, we track what percent of nodes of the network also resolve in favour of this branch. This is useful as ideally we would want the entire network to switch to the same branch with the presence of Anchors.

4.2.3 Forks Prevented

With the introduction of Anchors, there can be scenarios where a fork is prevented from happening. This happens when an anchor arrives on a block earlier than a sibling block that would otherwise create a fork in the chain and the anchor prevents the fork before it could even occur.

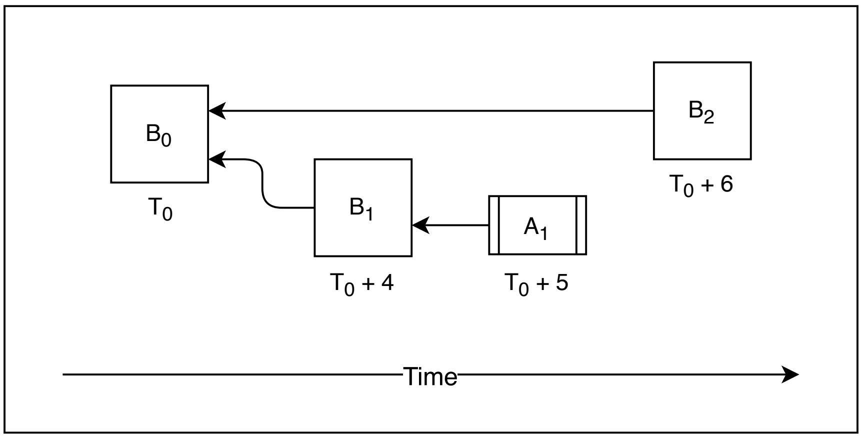

FIGURE 4.3: Fork Prevention in a Blockchain with Anchors

Figure 4.3 depicts such a situtation, where A fork would've been caused by Block B_2 but since Anchor A_1 arrives before B_2 the fork never occurs as the chain having A_1 is still heavier than the other one.

To compute results, we count for each unique fork that occurs, the number of nodes that it gets prevented on, and then average it over the multiple runs of experiments we perform.

Chapter 5

Results

We conducted all experiments in 4 separate configurations, controlled by the *anchorsperblock* parameter (or *a*):

A Bitcoin system with anchor frequencies per block a = 2, 5 & 10 and to compare the results with unmodified Bitcoin which has no anchors present ($a = 0$).

Each configuration was run 15 times for 60 minutes, with block inter-arrival time fixed to be 30 seconds. The expected number of blocks in each run therefore were 120 (60 minutes / 30 seconds).

5.1 Propagation times of Anchors vs propagation time of Blocks (with Anchors)

Due to the absence of any transactions (apart from a Coinbase) in the anchor structure, all anchors have a constant size of 264 Bytes. The mean block size, on the other hand, across all experimental runs, was observed to be 992KB. Figure 5.1 plots the observed propagation times. When we compare mean propagation times of Blocks and Anchors, we notice that Anchors propagate around 5 times faster than Blocks. This aligns our hypothesis that Anchors are much smaller than Blocks and should have a much smaller propagation time. This shows the effectiveness of Anchors as a signalling mechanism. Quick propagation times by Anchors ensure that all nodes have more recent information about the hashing power division in the network when compared to Blocks alone.

5.2 Propagation times of Blocks with and without Anchors

Figure 5.1 also plots the propagation time of blocks with and without the presence of anchors in the system. We observe that the inclusion of anchors does not negatively affect the propagation times of blocks, with propagation times including anchors in the system increasing only 0, 11 and 21 percent

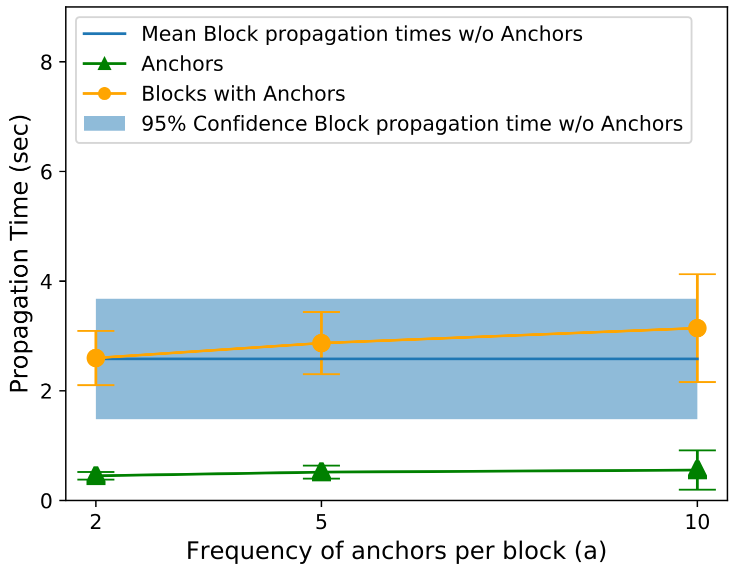

FIGURE 5.1: Mean propagation times of Anchors and Blocks with and without the presence of Anchors.

for a = 2, 5 and 10 respectively. Hence we can safely conclude that anchors do not create significant bandwidth or latency penalties in the system.

5.3 Frequency of Anchors per Block vs Resolution time of forks (RTF)

Figure 5.2 plots the average resolution time of forks (RTF) as observed in the network while varying anchor frequency (a) against a Bitcoin network without anchors. On comparing fork resolution times with and without the presence of anchors in the system, we observe that the Anchors reduce the mean fork resolution times by 57, 81 and 88 percent, with a = 2, 5 and 10 respectively. This shows how effective Anchors can be in resolving forks quicker.

5.4 Unition of mining power in forks in the presence of Anchors

Due to the decentralised nature of Bitcoin, each miner maintains their own copy of the chain locally, so forks occur and are resolved at each node. Analyzing the logs at each node, we observed that on an average (mean) - 94.7% of the network resolved a fork to the same branch when Anchors were present. Thus, we can see how Anchors are successful in reuniting mining power (divided across the branches of a fork) to the same chain.

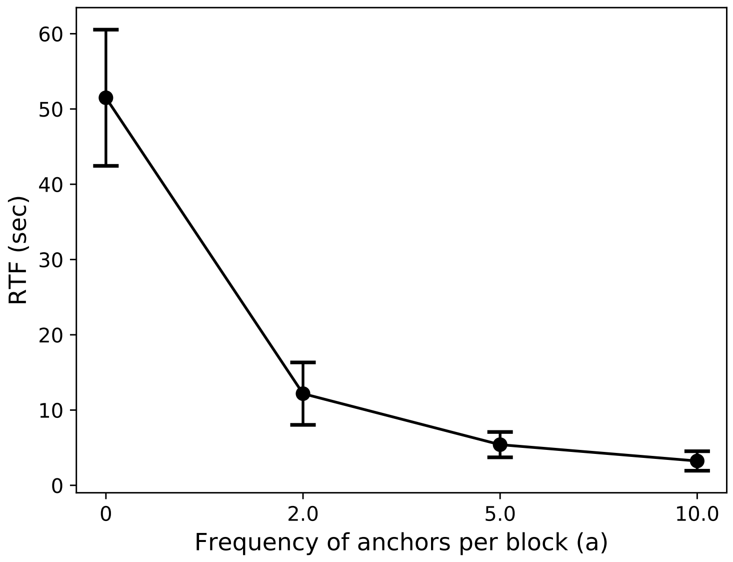

FIGURE 5.2: Mean Resolution Time of Forks (RTF) for different frequencies of anchors.
95% confidence intervals are drawn around the mean values.

5.5 Frequency of Anchors per Block vs Number of forks prevented

In this experiment, we calculate the percentage of nodes in the network where a fork was prevented from occurring. Figure 5.3 plots these results. No forks were prevented when Anchors are not present, and though the absolute values are not high, we still see that as a increases the percentage of network where a fork is prevented also increases.

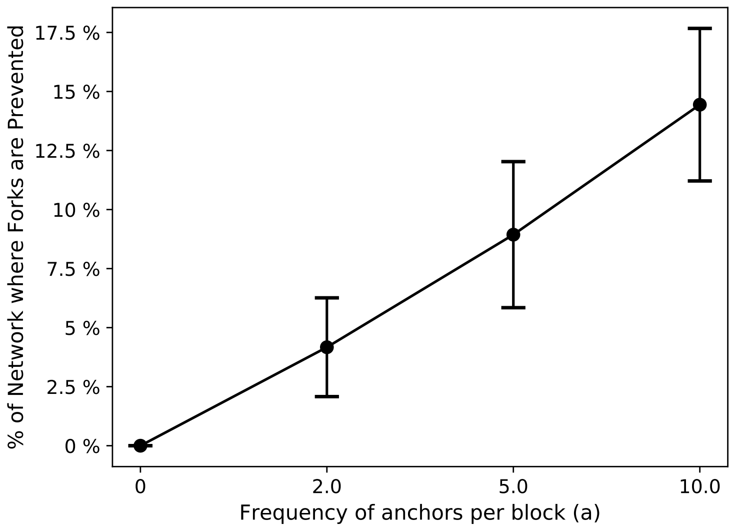

FIGURE 5.3: Percentage of network where forks are prevented.
95% confidence intervals are drawn around the mean values.

5.6 Effect of network topology

As mentioned in Section 4.1.2, we used a simple topology for the testbed network that we conducted all our experiments on, where each node was connected to five other randomly chosen ones. As suggested by prior work [27], the main Bitcoin network has been observed to have a topology where the degree of nodes follow a power-law distribution with majority of the nodes having a degree between 8 and 12. We tried to use similar degree values for our network as well (varying them between 4-8), but we found effectively no change in our results when we switched to a more Bitcoin like topology. This could perhaps be due to the fact that we have a considerably smaller network (of only 114 nodes) when compared to the main Bitcoin network, which has more than 6000 full nodes running at a time and when switching to a random node degree the diameter of our network didn't change.

Chapter 6

Conclusion & Future Work

We presented a new signaling scheme (called Anchors) that gives early insight to miners about the true hashing power division in the network. We created a proof of concept implementation of Anchors in the reference Bitcoin client (Bitcoin Core). After benchmarking unmodified Bitcoin against a variety of configurations of Bitcoin with Anchor support, we can conclude that Anchors are successful in making Bitcoin more robust, reducing fork resolution times and preventing forks altogether, without any substantial downsides to the system at large.

However, work on Anchors is far from complete. Currently, our implementation of Anchors does not consider rewards, which is a critical component if this proof-of-concept needs to be deployed in the real-world. Adding rewards might mean that we deviate from the current Bitcoin block structure, in which case, we might want to look at methods that could work as a "soft-fork" into Bitcoin. As new information about the exact topology of Bitcoin's overlay network becomes available, it could be integrated into the testbed to bring the emulation closer to the real world system.

We implemented Anchors in Bitcoin because it is the most widely deployed PoW blockchain, but in theory, Anchors could also be incorporated in blockchains using the GHOST protocol [38], or Ethereum [39] and give similar benefits. We could also experiment with implementing selfish-mining [17] in Bitcoin Core and then testing whether Anchors have the potential to mitigate such attacks.

Bibliography

[1] Arvind Narayanan; Joseph Bonneau; Edward Felten; Andrew Miller; Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies - Princeton University Press*. 2015. URL: https://d28rh4a8wq0iu5.cloudfront.net/bitcointech/readings/princeton_bitcoin_book.pdf.

[2] *ISO/TC 307 - Blockchain and distributed ledger technologies*. 2019. URL: https://www.iso.org/committee/6266604.html.

[3] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[4] Satoshi Nakamoto. *First use of the term "block chain"*. URL: https://github.com/trottier/original-bitcoin/blob/master/src/main.h#L795-L803.

[5] *Market capitalization of Bitcoin*. 2019. URL: https://coinmarketcap.com/currencies/bitcoin/.

[6] Adam Back. "Hashcash - a denial of service counter-measure". In: (2002). URL: http://www.hashcash.org/papers/hashcash.pdf.

[7] Hal Finney. "Reusable Proofs of Work". In: (2004). URL: http://web.archive.org/web/20071222072154/http://rpow.net/.

[8] John R. Douceur. "The Sybil Attack". In: *Peer-to-Peer Systems*. 2002, pp. 251–260. URL: https://link.springer.com/chapter/10.1007%2F3-540-45748-8_24.

[9] *Download - Bitcoin Core*. 2019. URL: https://bitcoin.org/en/download.

[10] The Bitcoin Core Developers Satoshi Nakamoto. *Source Code - Bitcoin Core v0.16*. 2018. URL: https://github.com/bitcoin/bitcoin/.

[11] *Bitcoin Core - Wikipedia*. 2019. URL: https://en.wikipedia.org/wiki/Bitcoin_Core.

[12] Haavard Nord; Eirik Chambe-Eng; Trolltech; Nokia; Qt Project; Digia; The Qt Company. *Qt Framework*. 2019. URL: https://www.qt.io.

[13] *Protocol documentation - Bitcoin Wiki*. 2019. URL: https://en.bitcoin.it/wiki/Protocol_documentation.

[14] *REST Interface - Bitcoin*. 2019. URL: https://github.com/bitcoin/bitcoin/blob/master/doc/REST-interface.md.

[15] *RPC Interface - Bitcoin*. 2019. URL: https://github.com/bitcoin/bitcoin/blob/master/doc/JSON-RPC-interface.md.

[16] Meni Rosenfeld. "Analysis of hashrate-based double spending". In: *arXiv:1402.2009* (2014). URL: https://arxiv.org/pdf/1402.2009.

[17] Ittay Eyal and Emin Gün Sirer. "Majority is not enough: Bitcoin mining is vulnerable". In: *International conference on financial cryptography and data security*. Springer. 2014, pp. 436–454. URL: https://arxiv.org/pdf/1311.0243.pdf.

[18] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. "Optimal selfish mining strategies in bitcoin". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 515–532. URL: https://arxiv.org/pdf/1507.06183.

[19] Kyle Croman et al. "On scaling decentralized blockchains". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 106–125.

[20] *VisaNet - The technology behind Visa*. URL: https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/visa-net-booklet.pdf.

[21] The Bitcoin Core Developers Satoshi Nakamoto. *Source Code - Bitcoin Core v0.16*. 2018. URL: https://github.com/bitcoin/bitcoin/releases/tag/v0.16.0.

[22] David Mackenzie. *GNU Build System*. 2019. URL: https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html.

[23] The Bitcoin Core Developers. *Building Bitcoin Core on Unix*. 2018. URL: https://github.com/bitcoin/bitcoin/blob/master/doc/build-unix.md.

[24] *Building Bitcoin Core on ArchLinux*. Christian Rebischke, Timothy Redaelli, shahid. URL: https://git.archlinux.org/svntogit/community.git/tree/bitcoin/trunk/PKGBUILD.

[25] The Apache Software Foundation. *Apache JMeter*. 2019. URL: https://jmeter.apache.org/.

[26] The Apache Software Foundation. *Distributed Testing - Apache JMeter*. 2019. URL: https://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html.

[27] Andrew Miller et al. "Discovering bitcoin's public topology and influential nodes". In: (2015). URL: https://allquantor.at/blockchainbib/pdf/miller2015topology.pdf.

[28] *Reduce fingerprinting through timestamps in addr messages - Bitcoin Core*. 2015. URL: https://bitcoin.org/en/release/v0.10.1.

[29] Matthias Grundmann; Till Neudecker; Hannes Hartenstein. "Exploiting Transaction Accumulation and Double Spends for Topology Inference in Bitcoin". In: *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2019, pp. 113–126. URL: https://link.springer.com/chapter/10.1007/978-3-662-58820-8_9.

[30] Ittay Eyal et al. "Bitcoin-NG: A Scalable Blockchain Protocol." In: *NSDI*. 2016, pp. 45–59. URL: https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-eyal.pdf.

[31] Varun Deshpande, Hakim Badis, and Laurent George. "BTCmap: Mapping Bitcoin Peer-to-Peer Network Topology". In: *IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*. IEEE. 2018, pp. 1–6. URL: https://www.researchgate.net/publication/329299820_BTCmap_Mapping_Bitcoin_Peer-to-Peer_Network_Topology.

[32] Aric A. Hagberg; Daniel A. Schult; Pieter J. Swart. "Exploring network structure, dynamics, and function using NetworkX". In: *Python in Science Conference (SciPy2008)*. 2008, pp. 11–15. URL: https://networkx.github.io/documentation/stable/index.html.

[33] *Global Ping Statistics - WonderNetwork*. 2019. URL: https://wondernetwork.com/pings.

[34] Stephen Hemminger et al. "Network emulation with NetEm". In: *Linux conf au* (2005), pp. 18–23. URL: https://www.rationali.st/blog/files/20151126-jittertrap/netem-shemminger.pdf.

[35] *netem - The Linux Foundation Wiki*. URL: https://wiki.linuxfoundation.org/networking/netem.

[36] Tsuyoshi Hombashi. *tcconfig - a tc command wrapper*. 2019. URL: https://github.com/thombashi/tcconfig.

[37] *IIT Delhi NTP Servers*. 2017. URL: http://www.cc.iitd.ac.in/CSC/index.php?option=com_content&view=article&id=106&Itemid=133.

[38] Yonatan Sompolinsky and Aviv Zohar. "Secure high-rate transaction processing in bitcoin". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2015, pp. 507–527. URL: https://www.cs.huji.ac.il/~yoni_sompo/pubs/15/btc_scalability.pdf.

[39] Vitalik Buterin and Others. "Ethereum white paper, 2014". In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.