THE CHINESE UNIVERSITY OF HONG KONG

FINAL YEAR PROJECT REPORT (TERM 2)

A Time Synchronization Protocol based on Distributed Network

Author
Tsz Hin Leung
(1155079351)

Supervisor
Prof. LYU Rung Tsong Michael

LYU1804

Department of Computer Science and Engineering

Faculty of Engineering

Abstract

Timechain is a proposed blockchain-based time keeping solution introduced in this project. It utilize the nature of distributed computing without a central authority to eliminate the single point of failures existing in the NTP technology. Credibility for a time is being developed when the chain grows and more nodes participates in the network. Statistics shows that by analyzing the discrete timestamps stored in the blockchain, a fairly accurate estimate on the real time can be achieved. With further investigation on how such timestamps is chosen, this Timechain solution is ready to be deployed in real life situations.

Table of Contents

1. Introduction 6

2. Background Research 7

2.1 Consensus Algorithm 7

2.1.1 Proof of Work 8

2.1.2 Proof of Stake 10

2.1.3 Byzantine Fault Tolerance 11

2.1.4 Analysis 13

2.2 Timing mechanism 17

2.2.1 Lamport Clock 17

2.2.2 Vector Clock 20

2.2.3 Network Time Protocol 23

2.4.4 Analysis 25

3. Design and Implementation 28

3.1 The block 28

3.2 Consensus algorithm 29

3.3 System Flow 30

3.4 Implementation 32

3.5 Usage 34

4. Testing and Evaluation 36

4.1 Case 1: All legitimate nodes 36

4.2 Case 2: Nodes time shifted 40

5. Improvements 47

6. Conclusion 48

7. Bibliography 49

1. Introduction

With the emergence of distributed computing, keeping a synchronized “time” across different nodes became important so as to determine the ordering happening between events. By “time”, we do not only mean the hours and minutes and seconds, but anything then can be used to distinguish events happening at different period can be called time. There exist various mechanisms for distributed systems to keep their time, but none seems perfect.

Following-up discussion of the last semester, I have examined the network transmission vulnerability in Network Time Protocol, and proposed a blockchain solution to act as a creditable time source. In this semester, further investigations have been made on consensus algorithms and timing mechanisms, and the blockchain implementation is being employed to collect operating data for analysis.

This project investigates further possibility of using blockchain in time keeping. It eliminates the central authority by a distributed network and successfully maintain consensus among all participating nodes. The collected data reflects that this usage is indeed possible.

In this project, I will investigate different consensus algorithms and timing mechanisms in chapter 2, together with an analysis on how such ideas can be implemented on my project. Chapter 3 describes the design details for the proposed Timechain solution, as well as the establishment of the prototype. Chapter 4 put the proposed solution into real operation and compares the collected data with time synchronization in NTP. And followed by the data obtained, the possible improvements needed are described in chapter 5.

2. Background Research

2.1 Consensus Algorithm

Consensus problem is an important concept in distributed systems to achieve a consistent data value across all the nodes in the network. Some of the nodes in the network can be faulty or unreachable at any point of time, and thus a fault tolerant consensus algorithm is in place to solve the conflicts between different nodes to achieve a single consensus value. It allows different nodes to communicate in a trustless manner. Byzantine fault represents nodes that can inconsistently appear as functioning and failed to other nodes monitoring the network. Typical Byzantine faults including the following:

1) Node not responsive within a timeout period
2) Node responding an incorrect result
3) Node returning deliberately misleading result
4) Node provide different result to different sections of the network

In the scenario of a blockchain, as the operation is decentralized, rules must be setup such that every node adheres to to maintain the system. consensus is needed to achieve the following:

1) Ensuring the proposed block in the chain generated by a node is legitimate; and
2) Preventing malicious users from successfully derailing the system.

Following-up with the discussions in Term 1, Proof of Work is used in the proposed blockchain system to hold the time. Proof of Work requires a large computational overhead, and thus other consensus protocols are investigated in the section to find if there are any suitable alternatives.

2.1.1 Proof of Work

Proof of Work ("PoW") for use in a blockchain was first introduced by Natoshi Nakamoto in 2008 and implemented in Bitcoin. [1] The participants of the blockchain are called miners. To prepare a new block, a miner first bundle all the information to be included in the new block. It then solve a complicated mathematical puzzle, essentially a hashing function, to verify the block. To compute the hash, the algorithm takes in the bundle of information and generate nonce values. A combination of the block content with the varying nonce values result in different hash values. The computational puzzle is considered as solved when the hash value matches a certain pattern. The block is then ready to be broadcasted to other nodes in the network.

	Nonce	Hash
Block content	0001	888B19A43B151683C87895F6211D9F8640F97BDC8EF……
	0002	4FAC6DBE26E823ED6EDF999C63FAB3507119CF3CB……
	0003	446E21F212AB200933C4C9A0802E1FF0C410BBD75F……
	……	
	1234	03AC674216F3E15C761EE1A5E255F067953623C8B38……

Figure 2.1 This figure demonstrates the use of Proof of Work. The constant block content combined with varying nonce values generate different hashes. A block is accepted if the hash value matches a certain pattern.

While the computational process is basically a brute-force approach and requires a significant amount of computing power[1], it is fairly easy for a node to verify if the hash

[1] Computational power required to generate a block varies by the pattern required. A tighter required pattern results in the computational puzzle more difficult to be solved.

solution matches the content and the nonce of the block, which takes only constant time. The nature of hashing ensures that the generation of hash value is not predictable. Pattern of the generated hash is equally distributed. In other words, any single node in the network of the same computational power has the same probability of generating the new block. The allows nodes in the network to participate equally without any single node easily dominating the whole network.

There may exist a tie in a PoW system such that two valid blocks are being generated by different nodes at almost the same time. Due to network transmission delay, different participating nodes may receive different blocks and append that block to their respective chains. The resulted in a fork such that different nodes on the network are working on different, yet valid chains. This situation is resolved by taking the faster developing chain as the only solution. From time to time, each participating node broadcast its own blockchain to the other nodes. If the receiver finds out that the received blockchain is longer than the one it is currently working on, it will replace the local blockchain with the received one and continue to work on the longer blockchain. The incentive to replace the blockchain is that all nodes are supposed to be configured to work on the longer chain, and hence the longer chain will surely develop faster than the others. There is no reason to continue to work on an obsoleted chain.

Generally, we do not want the mathematical puzzled to be too easily solved. It is to prevent a malicious attacker of the system to easily alter the blocks existing in the chain. For a malicious attacker to change a particular block, its time required can be estimated by the following:

Assuming a block, on average, is being generated for every *t*-seconds. Also assume that the malicious user possesses *y%* of the total computational power of the whole network, to alter the block at position *n*-*k*, it takes at least $\frac{(k+1)t}{y}$ seconds for the forked chain to reach position *n*. However, for the remaining nodes working on the legitimate chain, the legitimate chain will be reaching the $n+\frac{(k+1)t}{y} \div \frac{t}{(1-y)} = n+\frac{(k+1)(1-y)}{y}$. A block in the malicious chain is generated at a rate of $\frac{y}{t}$, while blocks in the legitimate chain is generated at a rate of $\frac{1-y}{t}$. Keeping t constant, as long as $y < 0.5$, the malicious chain is guaranteed to be growing slower than the legitimate chain. Therefore, if the pool of machines working on the blockchain is large enough, it is difficult for a single malicious user to obtain a subset of the computation power which exists 50% of the whole network, thus this method is tolerant to faulty nodes.

2.1.2 Proof of Stake

Proof of Stake ("PoS") is a consensus algorithm jointly proposed by Sunny King and Scott Nadal in 2012 and implemented in the same year in a cryptocurrency named Peercoin. [2] It employs a similar mathematical puzzle idea with PoW. However, for a node to participate in the network, it is required for the node to stake an amount of their tokens as a deposit to be chosen as a forger of a block. The deposit is locked by the system, and used as a collateral to participate in the block generation. The system is maintained by the economic incentives associated with it.

For a node to be chosen as a forger, there are two steps. In the first step, a node having a larger stake has a higher possibility of being chosen as a forger. The reason behind is that the deposit is paid by tokens that are value-for-money in real world. If the cryptocurrency system cannot work properly, the currency will be significantly deprecated. For a forger staking more tokens than the others, malicious actions set them back by a greater amount than other nodes staking less. Therefore, a node heaving maliciously comes with a great risk of losing a significant amount of real money. It creates an economic incentive for a potential forger to work legitimately.

However, as the forger earns transaction fee as a reward (not to be discussed in this project as a blockchain for time synchronization does not involve virtual currencies), and more stake a forger owns, a better change for it to be chosen as the next forger. It creates a vicious cycle for the wealthy node to dominate the system, shifting the system from decentralized to centralized. Two extra mechanisms come into place to prevent this from happening. In the first mechanism, blocks being generated are selected randomly such that the other participating nodes, although having less stakes, can also participate in the system. Another mechanism is by use of the coin age. The PoS system in Peercoin requires stakes to be at least 30 days old (called coin age), and the probability of forging the next block increases with the coin age until reaching a maximum at 90 days coin age. For a stake used successfully to generate a block, the coin age is reset to zero, and require at least 30 days without spending them to be eligible to participate in another round of forging a block. This ensures the stability of the system while no single node having a very old or very large collection of tokens successfully dominating the network.

2.1.3 Byzantine Fault Tolerance

Byzantine Fault Tolerance is developed from the class Byzantine General Problem: Byzantine Army composed of several divisions are planning on an attack to an enemy city. The divisions, each commanded by a general, can only communicate with each other through messengers (assuming that the messengers are trustworthy). In order to initiate a successful attack, a strong majority of the generals have to attack at the same time. Among the generals, some of them are traitors. The loyal generals will need to agree on a reasonable plan of action based on the algorithm, essentially coming up with the same decision, while the traitors will do anything they want. The problem requires an algorithm that guarantees the following:

1) All loyal generals decide upon the same plan of action
2) Traitors cannot cause the loyal generals to adopt a bad plan [3]

This Byzantine General Problem is being widely studied and several solutions have been proposed. One of the solutions most widely-employed in modern distributed computing system is the practical Byzantine Fault Tolerance ("pBFT") scheme. [4] Its objective is to mitigate the influence of faulty nodes on the collective decision making process so as to defend against system failures. The algorithm is based on the assumption that all the maliciously nodes are behaving independently.

In each round of the consensus, a leader node is elected among all the participating nodes in a round robin format. The consensus is then broken down into 4 phases:

1. The client sends a request to the leader node
2. The leader node forwards the request to all other nodes in the network
3. All the nodes process the request and send the reply directly to the client
4. The request is considered successful if m+1 nodes replies with the same result, where m is the maximum number of nodes they are allowed to be faulty in the system. The final result is the agreement that is reached by the honest nodes.

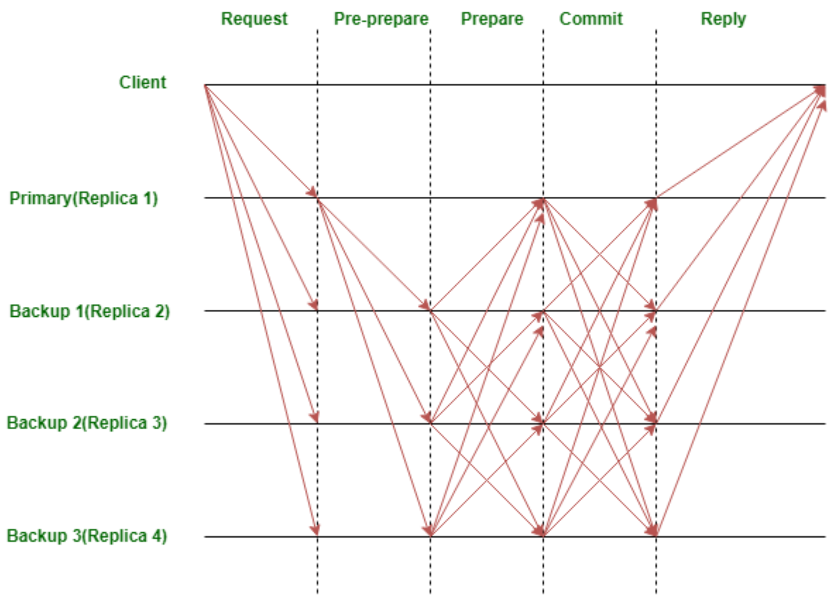


Figure 2.2 Illustration on how pBFT works [5]

There also exist a mechanism for supermajority of honest nodes to decide whether the leader is faulty and replace it with the next leader as specified in the round robin format. The pBFT system can function as long as the maximum number of malicious nodes allowed on the network is not greater than or equal to one-third of all the nodes in the system.

2.1.4 Analysis

1) Computational overhead

   PoW clearly consumes the most computational power as solving the mathematical puzzle requires a brute-force approach. The more the computational power a user have control of, the more likely for him/her to mine the next block. Although it is possible to make a block easier to be generated (i.e. require a looser hash pattern requirement), it makes it easier for a malicious user to deploy enough computing power to alter the existing blocks on the chain. Hence, the computational power is considered to traded-off for the security of the blockchain.

   While PoS is based on a similar hashing requirement compared to PoW, its computational overhead is significantly lower as only the staked nodes are allowed to forge the blocks. Chances for successfully forging a block is determined by the amount of stake and possibly the coin age, and hence more computational power cannot help in gaining an advantage in the system.

The computational overhead for pBFT is the lowest as its consensus is reached by network communications.

2) Network overhead

In contrast, the network overhead for pBFT is the highest. It requires all nodes to ne reachable with each other, forming a completely connected network. Every request is being sent to all the nodes and all the nodes are required to reply with their result. In a large network, there may exist numerous of requests and responds going in and out of every single node.

PoW and PoS shares similar network overhead. They do not require every node to be completely connected. As they allow blocks being generated at the same time and resolution of the conflict is deferred, each participating node only requires broadcasting their block / blockchain to its neighboring nodes. The neighbor nodes will then propagate the valid block / blockchain to their neighbors. After several levels of propagation, every nodes will agree on the same blockchain and thus consensus being reached.

3) Fault tolerance

Both PoW and PoS are vulnerable to 51% attacks. For a PoW system, a malicious possesses of 51% (to be exact, >50%) of the networks overall computational power can deliberately agree on any chain they want if the hash is valid, whether the content is valid or not. As the fork of the chain having more computational power working on it is going to develop faster than the others, due to the nature of taking the longer chain, this malicious chain will be accepted by other nodes.

PoS is also susceptible to 51% attack. Instead of possesses of 51% of computational power, PoS attackers possesses 51% of the tokens in the network can create a dominant position in forging new blocks. However, as successful attacks on the system will severely harm the credibility of it, the attacker staking 51% of tokens will result in the most significant loss compared to nodes staking less. Therefore, the economic incentives make such attack not likely to happen.

pBFT is the least fault tolerant as it requires a supermajority to reach a consensus. It guarantees integrity only when the malicious nodes in the network cannot simultaneously equal or exceed 1/3 of the total nodes in the system.

4) Transaction finality

pBFT is able to provide transaction finality as a request initiated by a client is agreed upon all nodes in the network. When there is enough nodes agreeing with the same value, the result is finalized and not changing. This is achieved by the fact that all honest nodes agree on the same result through network communications with each other.

On the other hand, as PoW and PoS allows forks, the chain a node is working on may not be the final node. Although forks are usually resolved within a few blocks in the blockchain, there exist occasions that the blockchain a node working will be obsoleted upon receiving a long chain. Hence, blocks are not finalized until it is reaching further levels deeper.

5) Scaling

PoW and PoS enable completely scalable consensus network. As each node is only connected to its neighbors, it allows the system to expand exponentially. Also, the nodes do not require knowing all other nodes in the network for the algorithm to function properly. For example, by connecting only to the neighboring 2 nodes, a message can reach over 10,000 nodes within only 14 hops. In the system, it allows different blockchains being developed at the same time and deferring the conflicts to be resolved. Therefore, it does not require a generated block being dispatched to all other nodes in real time, and hence much easier for the system to scale up.

In contrast, pBFT requires all nodes to be completely connected, and the total number of nodes have to be known to determine the number of acceptable faulty nodes. This can hardly be deployed on a wide area network where there may exist a numerous number of nodes attempting to participate. Therefore, pBFT is usually deployed in permissioned networks.

In contrast, pBFT requires all nodes to be completely connected, and the total number of nodes have to be known to determine the number of acceptable faulty nodes. This can hardly be deployed on a wide area network where there may exist a numerous number of nodes attempting to participate. Therefore, pBFT is usually deployed in permissioned networks.

2.2 Timing mechanism

In a single machine where all the components share the same bus, there isn't usually a problem to organize event as they are referring to the same CPU time. However, in distributed system, in order to determine the ordering of the events, it is much harder as different clocks on different machines have different values. Also, the clocks may skew or have errors, and network delays may change the ordering of the events. Therefore, we need to find out mechanisms that can help us determine the orderings.

In different timing mechanisms, there except two concepts of time:

1) Physical time – The time that we use in real world in the format of years, months, days, hours, minutes and seconds. In computational systems, we present it using an epoch: the number of seconds elapsed since a predefined date and time. The Unix epoch is 00:00:00 of 1st January, 1970.
2) Logical time – Essentially a monotonically increasing counter that is influenced by the events happening in the system, hence only logically applicable to the participants in the network.

In this section, we will examine 3 important timing mechanisms.

2.2.1 Lamport Clock

Lamport algorithm is a simple algorithm in order to determine the logical ordering of events in a distributed system. It is developed by Leslie Lamport in the year 1978. [6] With this algorithm, we can determine happened-before relationships with minimal overheads.

A counter is kept in each node of a system. The algorithm works based on the following rules:

1. Increment the counter for each internal event
2. When a node sends a message, it increases its counter by 1 and include the new timestamp in the message
3. When a node receives a message, it adopts either the local counter or the timestamp in the message as the new local counter, depends on whichever is larger. The local counter is then incremented by 1 to represent the event of receiving the message.

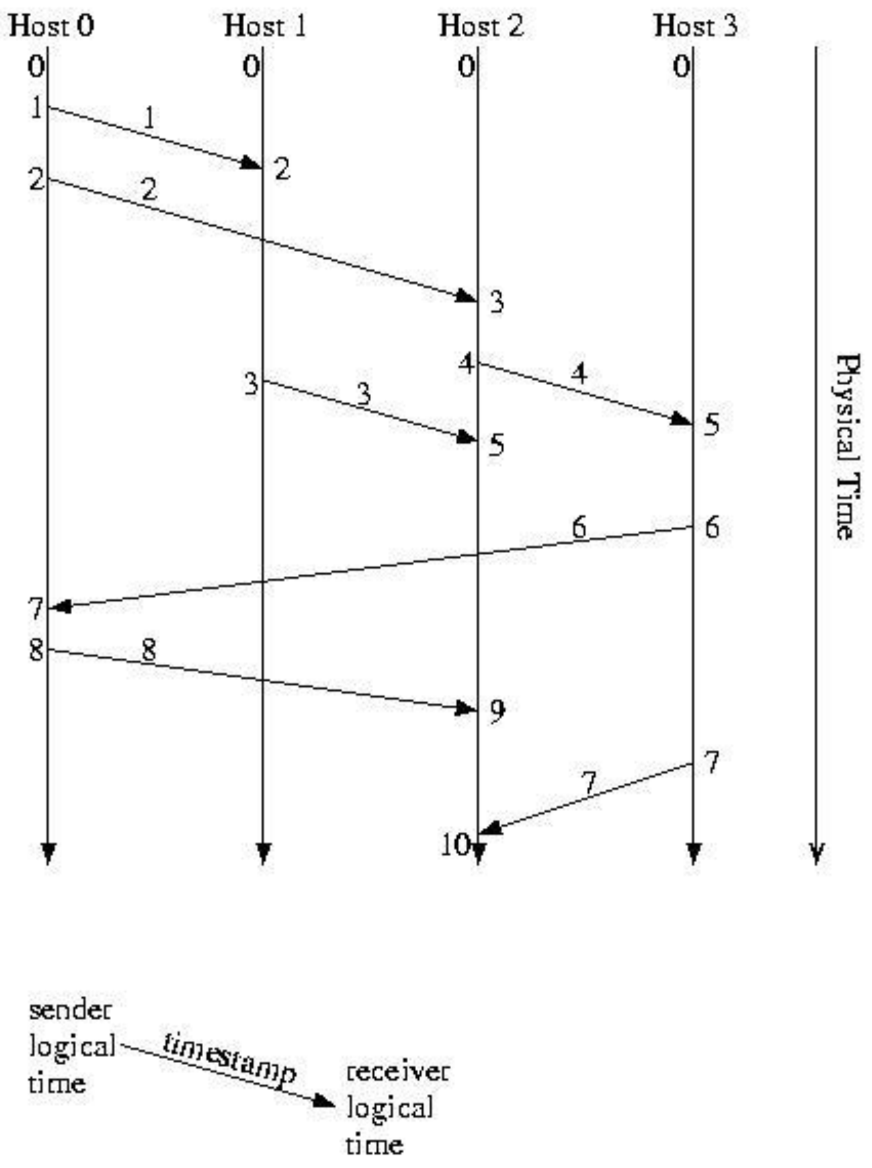


Figure 2.3 Illustration on how Lamport clocks work. [7]

Pseudocode for sending a message:

```
local_time++;

send(message, local_time);
```

Pseudocode for receiving a message:

```
receive(message, timestamp);

local_time = max(timestamp, local_time) + 1;
```

By comparing the timestamp of the events, we can define happened-before (→) relationships as the following:

1) On the same node, a → b if and only if C(a) < C(b)
2) A node sends a message → another node receives the message
3) Relationship is transitive: if a → b and b → c, then a → c

If two nodes have never communicated with each other, not even through any third parties, the ordering of the events between them cannot be determined. They are said to be concurrent events. Yet, if they do not exchange message, then they probably do not need a shared common clock. Therefore, only partial ordering can be reached with Lamport Clock. That is, if a → b, then C(a) < C(b), but the other way (saying if C(a) < C(b) then a → b) does not always hold. We can only determine that if C(a) ≮ C(b), then a cannot be happened before b. Mathematically:

$$a \rightarrow b \Rightarrow C(a) < C(b) \qquad C(a) \nless C(b) \Rightarrow a \nrightarrow b$$

Just by having the Lamport timestamps we cannot determine if there exist a casual happened-before relationship. A strong clock consistency condition cannot be achieved with Lamport Clock.

2.2.2 Vector Clock

Vector clock employed the concept of Lamport clock, but instead of keeping only one counter, each node keeps a vector of counters, each representing the last-known counter value of a single node in the network. [8] The algorithm works according to the following rules:

1. Increase the counter representing the node itself for each internal event
2. When a node sends a message, increase the counter representing the node itself and sends the whole vector
3. When a node receives a message, increase the counter representing the node itself and updates every element in the vector by taking the maximum of the received vector and the local vector

Pseudocode for sending a message:

```
local_vector[self]++;

send(message, local_vector);
```

Pseudocode for receiving a message:

```
receive(message, timestamp_vector);

timestamp_vector[self]++;
```

```
for i in range num_of_nodes:

      if (timestamp_vector[i] > local_vector[i]):

            local_vector[i] = timestamp_vector[i]
```

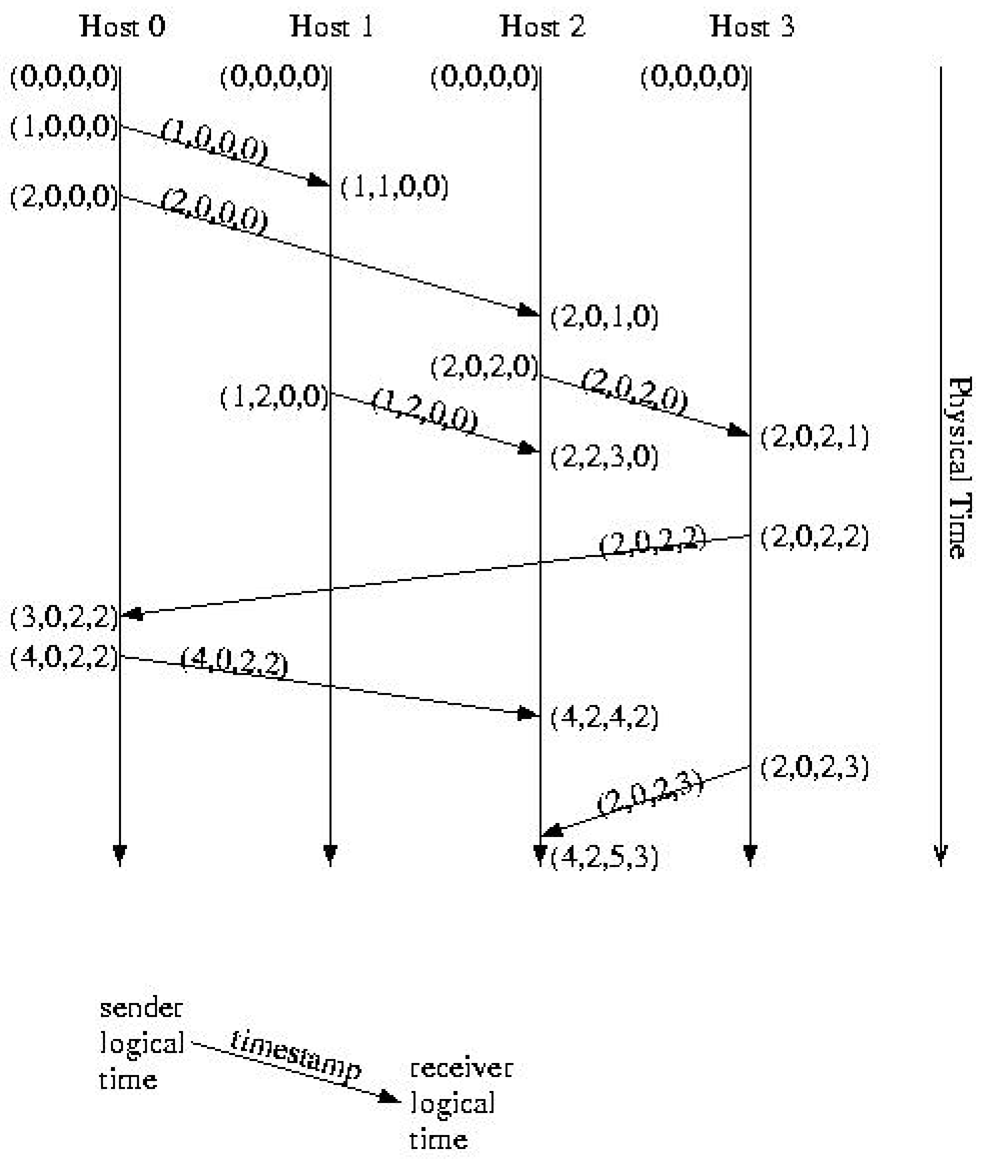


Figure 2.4 Illustration on how vector clocks work, using the same scenario in figure 2.3. [7]

It can be seen that in the vector clock, only the entry representing the node itself is the most updated. A recipient may have a more updated version of the vector clock, and hence it is needed for every node to determine the most updated value to be included in the vector.

By comparing vector timestamps, the following properties can be observed:

1) If VC(a) = VC(b), then a and b are the same event
2) If a $\rightarrow$ b, then each element in a's timestamp is less than or equal to the corresponding element in b's timestamp, and at least one element is less than the corresponding element in b's timestamp. Mathematically:

   $$VC(x) < VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z] \wedge \exists z'[VC(x)_{z'} < VC(y)_{z'}]$$

   where $VC(x)_{z'}$ represents the value of the vector clock for node z. [9]
3) If a and b are concurrent, their vector timestamps are mixed. Mathematically:

   $$\exists j'[VC(x)_{j'} > VC(y)_{j'}] \wedge \exists k'[VC(x)_{k'} < VC(y)_{k'}]$$

4) Both of the following can be ensured:

   $$a \rightarrow b \Rightarrow VC(a) < VC(b) \qquad VC(a) < VC(b) \Rightarrow a \rightarrow b$$

5) Relationship is transitive: if a $\rightarrow$ b and b $\rightarrow$ c, then a $\rightarrow$ c
6) Relationship is antisymmetric: $VC(a) < VC(b) \Rightarrow \neg(VC(b) < VC(a))$

Causality violation can be detected with vector timestamps by comparing the timestamp of the received message with the local time. For the local timestamp to be ahead of the timestamp of the received message, some other prior message must have been received for the advancement. The sender of the prior message must have obtained the latest message before the prior message is being received. Therefore, if the local clock is ahead of the timestamp of the message being received, there is a causality violation. If the events are considered concurrent, this is not a problem because the relationship between the events does not matter.

2.2.3 Network Time Protocol

Network Time Protocol (“NTP”) is a widely adopted protocol for physical time synchronization. It was first implemented in 1988[10] and the latest version is NTPv4, released in 2010[place]. It adopts a hierarch structure for time sources, starting from stratum 0, consisting of high precision time-keeping devices such as atomic clocks and satellites, and goes down to child time servers which end users usually synchronize the clock of their machine with. The devices at each stratum synchronize their time with the stratum above them, forming a layered structured. As shown in figure 2.5, apart from relying a single server as the only time source, user can configure a pool of servers and the selection algorithm selects a time source to synchronize itself with, based on estimating the trustworthiness of the configured time sources.

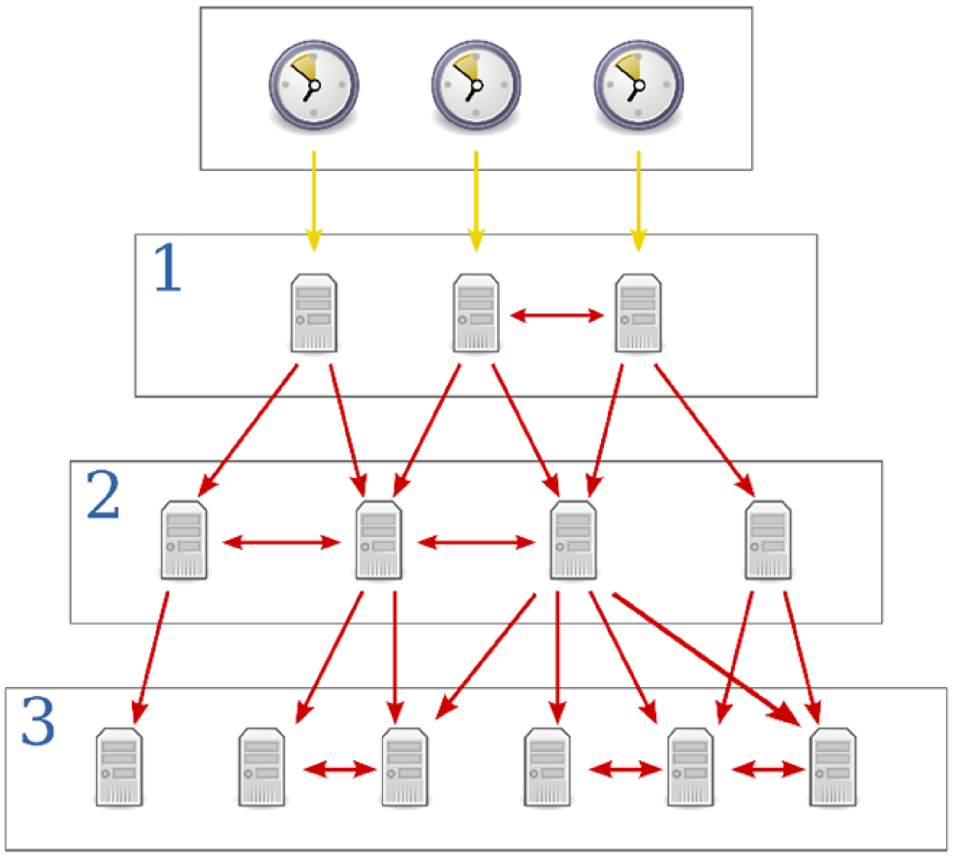


Figure 2.5 Illustration of NTP hierarchy [12]

NTP uses UDP port 123 for communication with the server. Upon receiving an NTP packet, the time server only appends the required information in the designated fields of the packet, and return it to the client as described below. This minimizes the load of the NTP server.

In the transaction of NTP packets, a minimal of 4 timestamps are used to calculate the time difference between the client and the server, taking transmission delay into consideration. The basic operation is as follows:

1. Client sends the NTP packet with its own local timestamp t_1 in "Origin Timestamp". It is the time the packet is being transmitted to the server.
2. Upon receiving the packet, server its local time of receive t_2 in "Receive Timestamp" in the data packet.

3. Server sends the response with the timestamp of the packet leaving the server t3 appended in “Transmit Timestamp”.
4. Client receives the NTP packet and log the receive time t_4.

With the above timestamps, the round-trip delay can be obtained with the following formula.

$$\delta = (t_4 - t_1) - (t_3 - t_2)$$

The offset between the local clock and the server clock can then be calculated, with the assumption that up-link and down-link delay is symmetric.

$$\theta = \frac{1}{2}[(t_2 - t_1) + (t_3 - t_4)]$$

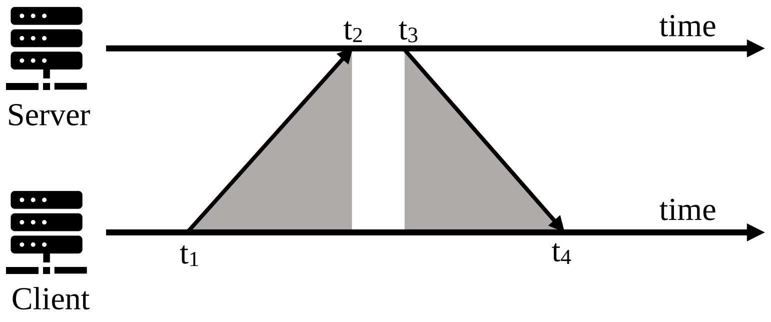


Figure 2.6 Round-trip delay δ

Following the discussions in the last semester, NTP is vulnerable to Man-in-the-middle attack and can suffer denial of service attacks.

2.4.4 Analysis

Among the timing mechanisms introduced above, Lamport clock and vector clock belongs to the family of logical clock, while NTP is designed to synchronize the physical time. As the mechanisms designed for logical ordering among the events are targeted on closed network, their implementation is merely an algorithm, without the design of any security policies or conflict resolution measures in place. Also, the vector clock stores all the counter values for every single node that has had communications with it (both direct and indirect), resulting in an extremely large vector if deployed in a large network. With these limitations, deploying Lamport clock or vector clock in a wide area network is not feasible.

On the other hand, NTP works well in providing a system with an accurate physical time, but it needs improvement in the security aspect. By the nature of the client-server model, its role of a single point of failure cannot be easily mitigated.

The proposed time synchronization method, Timechain, comes in place, where it acts as a hybrid solution between synchronization the real time and achieve events ordering using the blocks as the logical time. It will be further elaborated in the following sections. It is also included in the following figures to demonstrate its position comparing to other time synchronization mechanisms.

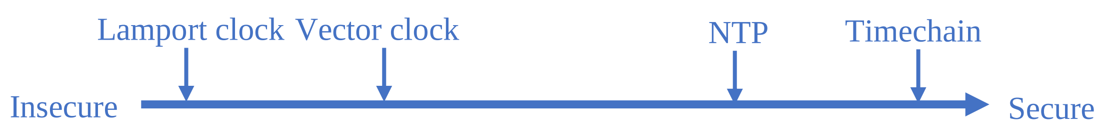


Figure 2.7 Comparing timing mechanisms by security level

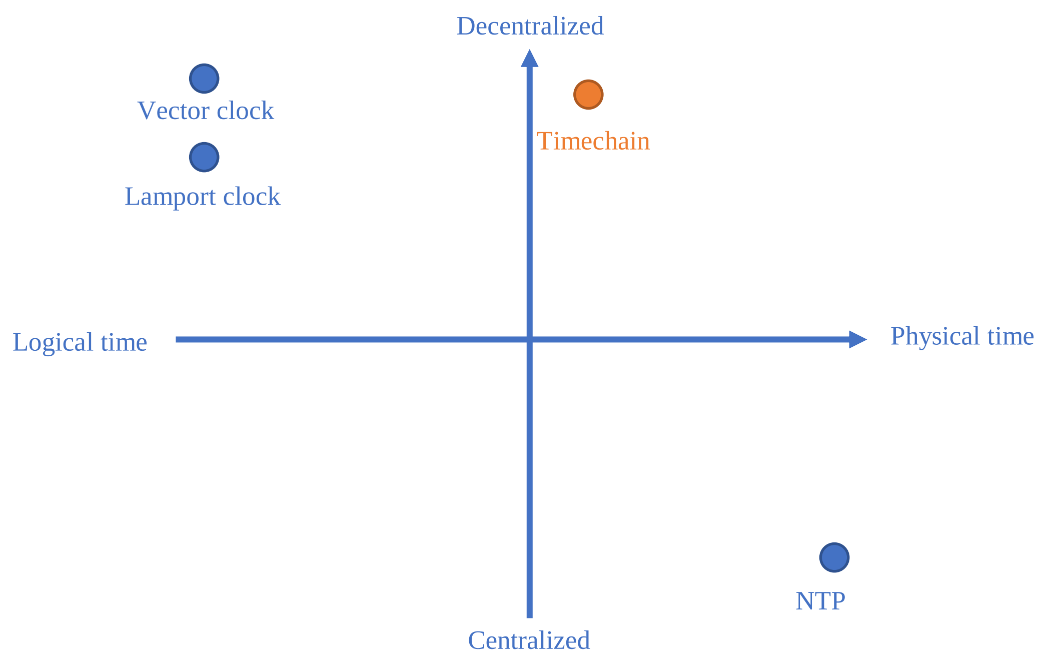


Figure 2.8 Comparing timing mechanisms by the degree of centralization and logical/physical time used

3. Design and Implementation

A decentralized and distributed solution based on blockchain is designed to keep the time. The blockchain is named “Timechain”, essentially a chain of blocks keeping the time. It allows time data to be appended to the chain by all the participating nodes to verify and audit the data. The blocks are being built on top of the previous blocks such that no user can easily alter the data in the blocks. Single point of failure is eliminated due to the distributed nature. The design of Timechain is described in this section.

3.1 The block

The blockchain is composed of a chain of blocks which each blocks consists of various fields for data storage. The whole chain is stored on every single node of the network, and all nodes are allowed to write their time data on to the Timechain (subject to validation by the consensus algorithm). A block in the Timechain consists of the following fields:

- Block index: An incrementing value starting from 0 (genesis block).
- Timestamp: The local timestamp of the block generator (at the time of block generation).
- Hash: The hash value is generated based on all the other fields in the block.
- Previous hash: The hash value of the previous block. It links the current block to the previous one.
- Nonce: The value used to track the Proof of Work (“PoW”) algorithm counter. It is added after the implementation of the last semester as this value acts as an unpredictable salt to the hash value. Using the timestamp as the salt is not a good design, as the timestamp value is predictable.

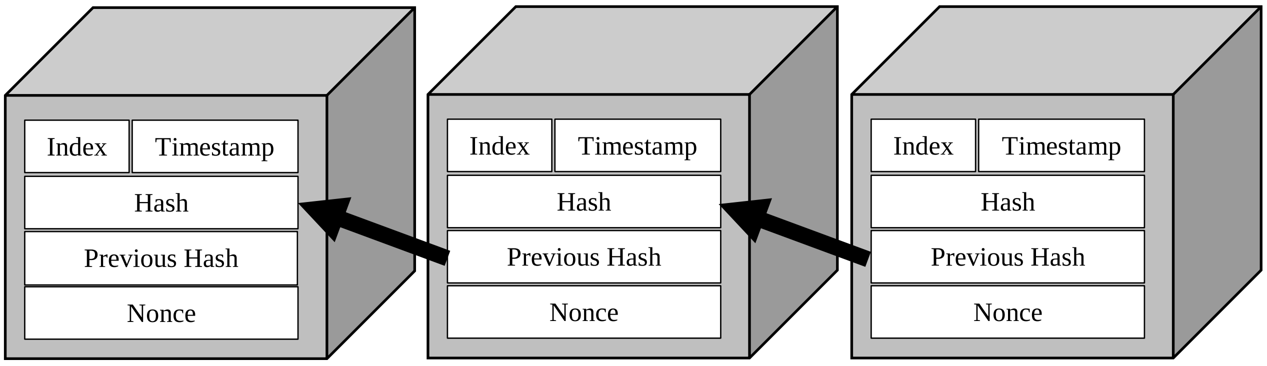


Figure 3.1 Structure of the Timechain

Hashing provides references between the blocks. It ensures no blocks are being tampered with. As the hash value is based on all the other content fields in the block, altering any content will result in a hash value mismatch, while altering the hash value will disconnect a block with the whole chain. As a block is broadcasted for other nodes for validation, an altered block will be rejected, thus fail to propagate and remains in the chain.

3.2 Consensus algorithm

After analyzing various consensus algorithms in Section 2.1, including Proof of Work, Proof of Stake, and practical Byzantine Fault Tolerance, PoW is chosen as it is most suitable in this scenario.

PoS is operated based on economic incentives. As no currency is being introduced in the Timechain, PoS does not have the foundation to be employed. PBFT is operated in a completely connected network. The network overhead is large, and it is not scalable to be adopted in a wide area network with numerous nodes. Using pBFT will limit Timechain to only a limited number of participants.

In contrast, PoW can be most easily employed, and it has the capability to form a large network to achieve a time exchange across many nodes. Although the computation overhead for PoW is significantly larger than other methods, by using PoW in Timechain, it overcomes the limitation for Lamport clock and Vector clock being unable to be deployed in public networks. PoW also empower Timechain to be expandable easily, thus allowing the service more accessible for different machines. Hence, the computational overhead is worth it to achieve the greatest improvement with the current timing mechanisms.

By using PoW, a user needs to try different combinations of the block content and different nonce values. In the Timechain implementation, to prepare any single block, the nonce and timestamp are changing values. Before any calculation of the hash value, the nonce and timestamps are being updated. The timestamp comes from the local time of the machine, and the nonce value can be from a random source. The desired hash pattern is achieved when a combination of the nonce, the timestamp, and the rest of the content of the block together generates the correct hash pattern.

In the Bitcoin scenario, the node first appends the timestamp into the block, and only the nonce value is changed in every brute-force trial. In contrast, the timestamp is changed for every attempt to calculate the hash, ensuring that the included timestamp in the block is the most up-to-date. The two changing values also makes it much harder to predict how a block can be generated, thus a barrier to malicious user attempting to alter the content of the blockchain.

3.3 System Flow

The Timechain runs in the following sequence:

1. Each node repeats the following until the mathematical puzzle is solved or a block is received:
    a. Create a block with the current timestamp and generates the nonce value
    b. Calculate the hash of the block
2. If the puzzled is solved, the block is broadcasted to neighboring nodes
3. Nodes that receives a block log the local time of receive and verify the hash result
4. If the hash is correct, a node determines if the block is valid by looking into the encapsulated content. If the block is valid, it is being accepted.
5. If the block is being accepted, it appends the block to the end of its chain and use its hash to work on the next block. Otherwise, it continues to work on the current block and wait if some other nodes broadcast other solutions

For every certain amount of time, each node will broadcast their local chain of blocks to their peers. By doing so each of the nodes will check if they are having the same chain. If no, the longest, and yet valid chain will replace the existing local chain.

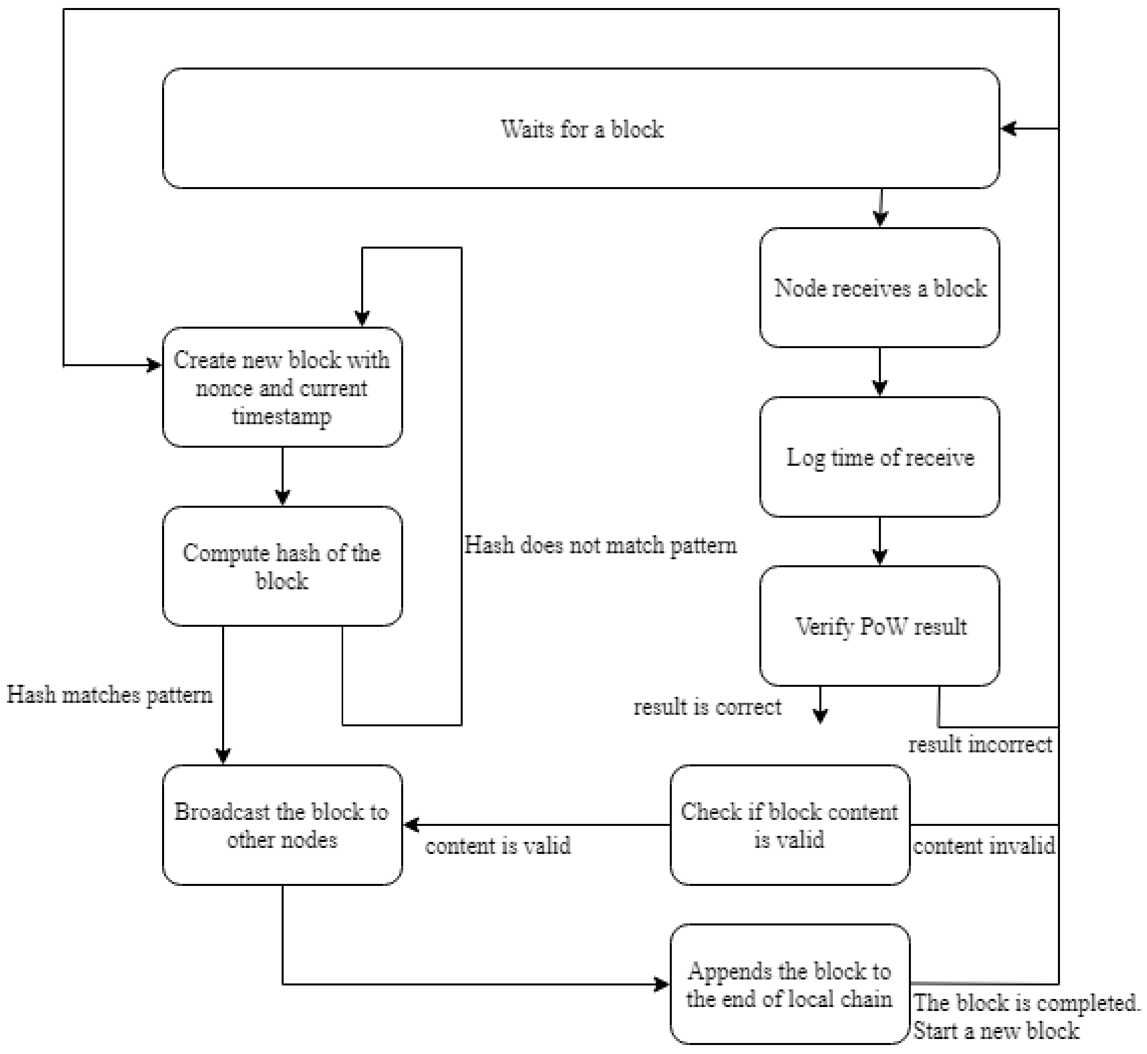

Waits for a block
Node receives a block
Log time of receive
Verify PoW result
result is correct
result incorrect
Create new block with nonce and current timestamp
Compute hash of the block
Hash does not match pattern
Hash matches pattern
Broadcast the block to other nodes
content is valid
Check if block content is valid
content invalid
Appends the block to the end of local chain
The block is completed. Start a new block

Figure 3.2 System flow of Timechain

3.4 Implementation

A prototype of Timechain is being implemented. This supports the following major functions:

1) Block generation – A block is composed with the local machine time and different nonce combinations are generated to test the hash.

2) Send generated block – The mined block is being sent to other neighboring nodes.

3) Receive block – It receives the block mined by another node. The program then checks if the block is valid and append it to the blockchain.
4) Send chain – The whole chain is being sent to neighboring nodes so as to solve any possible conflicts.
5) Receive chain – The local chains of other nodes are being received. The program will check if any received chain is longer than the local one. If there exist such chains, it will further check if the chain is valid. If the chain is valid, the longer chain will replace the local chain.
6) Logging – The chain and the corresponding activities are being logged and write to a .csv file for data analysis.

To simplify the development and to mainly focus on the functionality of the program logic, HTTP post/get methods are used to transfer blocks or chains information in JSON format between nodes. An HTTP request multiplexer library is used in handling the post/get requests and responses. This creates extra overhead in HTTP handshakes, but the delay is negligible. Peer discovery is not included in this prototype. It can be considered as a separate topic and requires extra in-depth study. In this prototype, the peer nodes are explicitly specified.

The following parameters can be configured to simulate different operating environments:

1) Difficulty – This refers to the tightness of the hash pattern requirement. In this implementation, the hash value of the generated block must start with a number of zeros, which the number is defined as the difficulty value. The more number of zeros required, the harder for a matching hash value to be found, and thus a longer time for generating a block.

2) Speed – Usually, a miner in a block fully utilize its available computing resources to compete for generating the block. However, in order to save computing resources, a speed value is configured to extend the intervals between each brute-force attempt. If this value is configured to 0, all the available computing resources will be used to calculate the hash. The speed and the difficulty, together determine the average time required for a block to be produced.
3) Threshold – This is the threshold value for deciding whether a block shall be accepted. If the different of timestamp on the block compared with the local machine time is larger than the threshold, the block is determined to be invalid and is ditched.

3.5 Usage

Timechain can provide two pieces of information for keeping both the physical and the logical time.

The physical time is kept by the timestamp of the block. A selected subset of timestamps in the Timechain can be used to calculate an estimation on what the real time is. Although accuracy is limited without determining the propagation delay of the broadcasts of the block, it can act as a level of assurance on the system time. With the nature of the blockchain, data in the blocks cannot be easily modified. Hence the physical time stored in the block is a more creditable source of time compared to other mechanisms.

On the other hand, the logical time is kept by the hash value of a block. For every event, the hash value can act as a signature indicating the time which the event happens. With the SHA256 hashing algorithm, the hash values are very, very unlikely to collide, thus it can relate the event to the block. With the relationship being generated, ordering of the events can be achieved by the help of the ordering of the blocks. It guarantees that an event cannot be faked to be happening in the future as the hash value for the block representing that time have not yet been generated. The speed of this logical clock depends on the generation rate of the blocks. Although we cannot determine the ordering of the events happening within the same block period, the order of events in different block periods can be well assured.

Therefore, this Timechain solution opens up further distributed computing applications on untrusted nodes. The improvement over existing timing mechanisms is that it can help solving the problem of order on nodes that we do not physically own it. Distributed applications that can only run in private networks can make use of the advantage of Timechain and deploy it in a public network.

4. Testing and Evaluation

A pool of virtual machines is being used in this testing state. As in the blockchain setting, all nodes are equivalent to each other, all the machines share the same piece of code. The testing aims to verify and functionality of the idea, and observe its behavior in different scenarios. Therefore, certain deploy requirements are loosen so as to collect the data for analysis. The blockchain is configured to the following:

1. On average a block is generated every 30 seconds.
2. The time different threshold is 1000 seconds (16.67 minutes). This threshold in lines with the threshold for NTP. Afterall, this loose requirement allows us to collect more comprehensive data to observer the behavior of the blockchain.

In the following setups, the Timechain is compared to the NTP to observe their differences in performance. It is also assumed that the network transmission is not under any kind of attacks. Note that the while the clients in the NTP setup corrects its time based on the NTP algorithm, the blockchain nodes are not applying any correction on the time. The major purpose of the tests is to obtain data for the behavior of the blockchain.

4.1 Case 1: All legitimate nodes

Number of machines: 3

Running time: 6 hours

In this setup, all nodes are legitimate. There exists a minimal time error due to transmission and processing delays, and also the fluctuation of the local clocks.

The setup for the NTP scenario involves 2 NTP servers forming a pool, while a single client connected to this pool of servers. All 3 nodes are initially calibrated to agree on a single time.

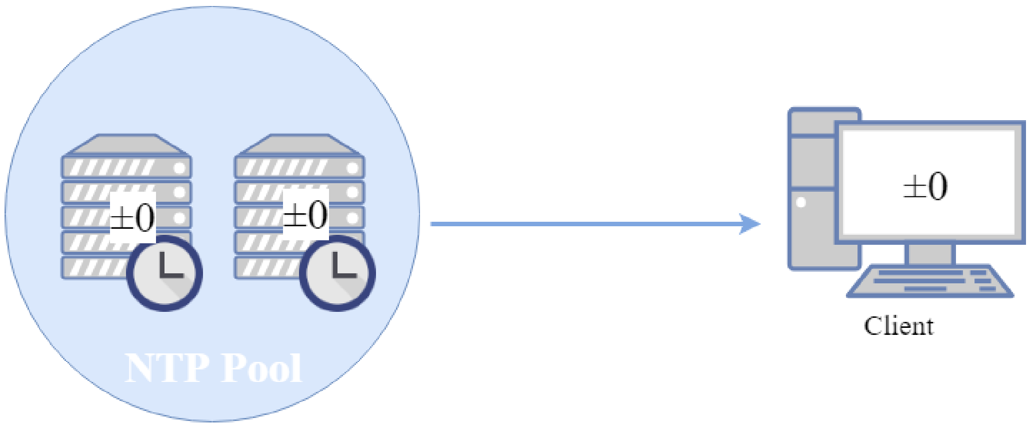


Figure 4.1 Illustration for the NTP setup. Time differences are shown in the figure.

The resulted NTP behavior can be observed as the following:

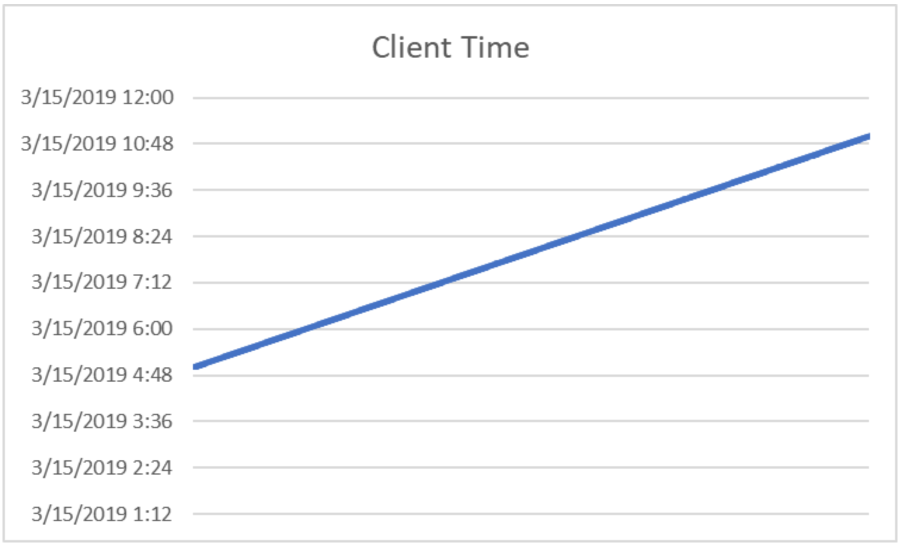


Figure 4.2 Corrected client time

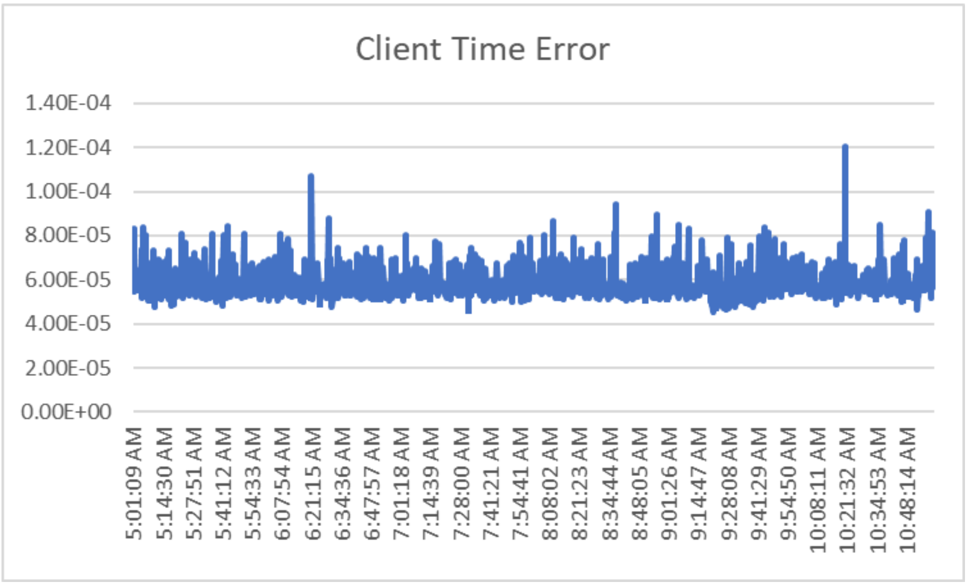


Figure 4.3 Difference between client local time and an external time source

In the setup of a pool consisting of 2 NTP servers, where both are legitimate, the client can obtain a fairly precise time, which is slightly faster than the NTP time. The difference ranges from 5x10^{-5} seconds to 7x10^{-5} seconds, while occasionally reaches a difference of over 1x10^{-4} seconds.

On the other hand, the Timechain is configured to the following:

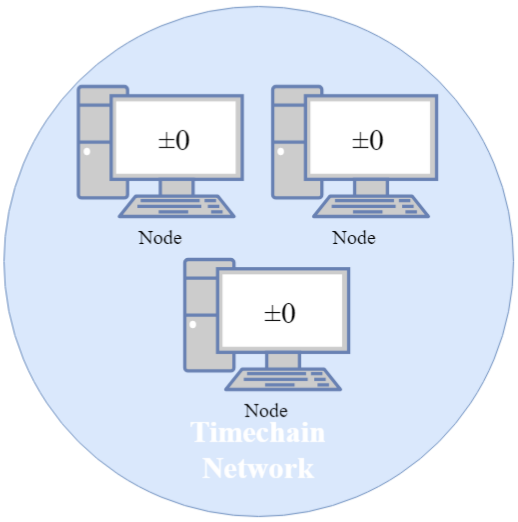


Figure 4.4 Illustration for the Timechain setup. Time differences are shown in the figure.

The blocks being generated as follows:

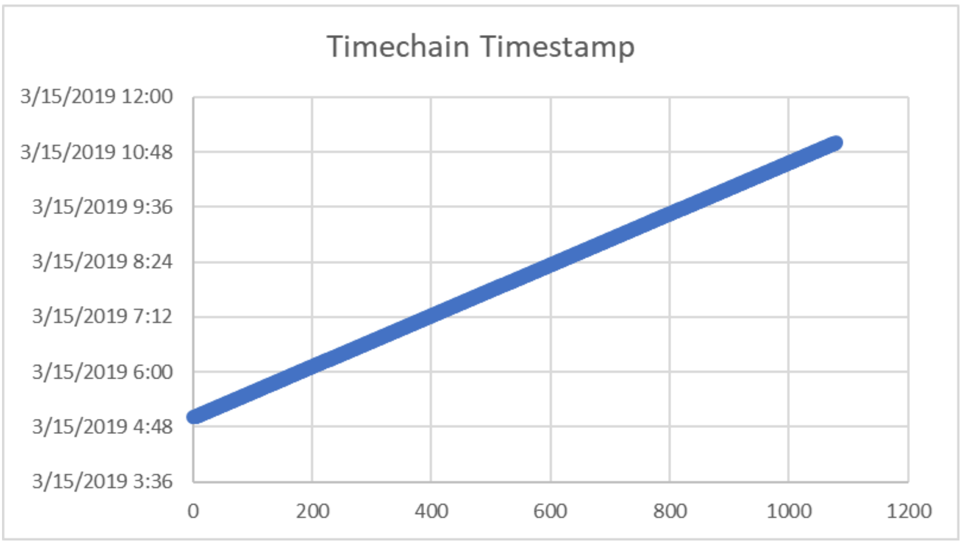


Figure 4.5 The Timestamps on the blocks

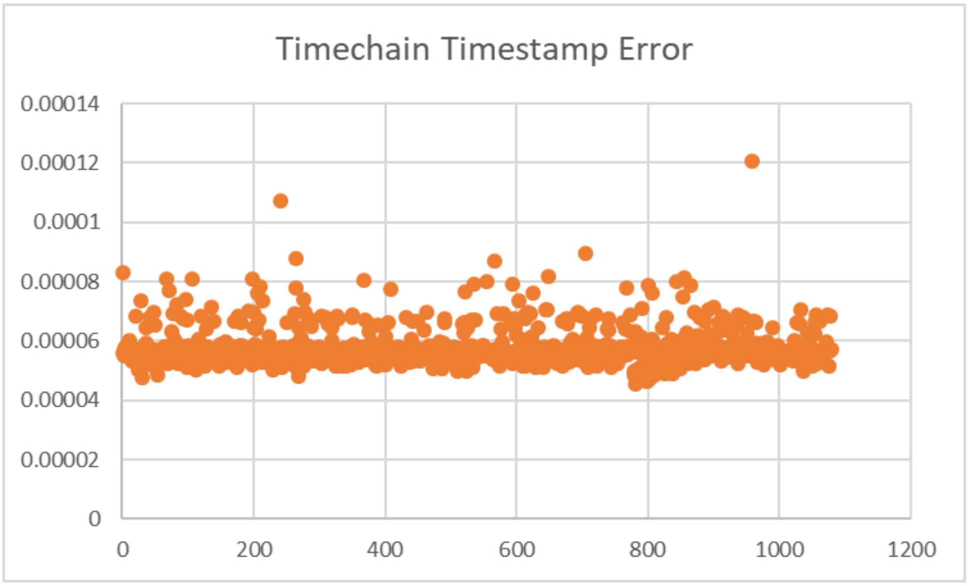


Figure 4.6 Difference between Timechain timestamp and an external time source

With the about observation, it can be seen that majority of the blocks are being generated in the interval of 5x10^{-5} seconds to 6x10^{-5} seconds, similar to the client time achieved by using NTP.

4.2 Case 2: Nodes time shifted

Number of machines: 7

Running time: 30 minutes (NTP) / 1.5 hours (Blockchain)

This setup demonstrates that some of the machines in the network has a constant time error with the true physical time. We would like to observe how these time error will result in time behavior of NTP and the Timechain.

The setup of NTP is as follows. There are 3 machines in the NTP pool, where 2 of them have their local time advanced for 2 minutes, while another one as time delayed by 2 minutes. All clients are initialized to have their time in line with the correct physical time.

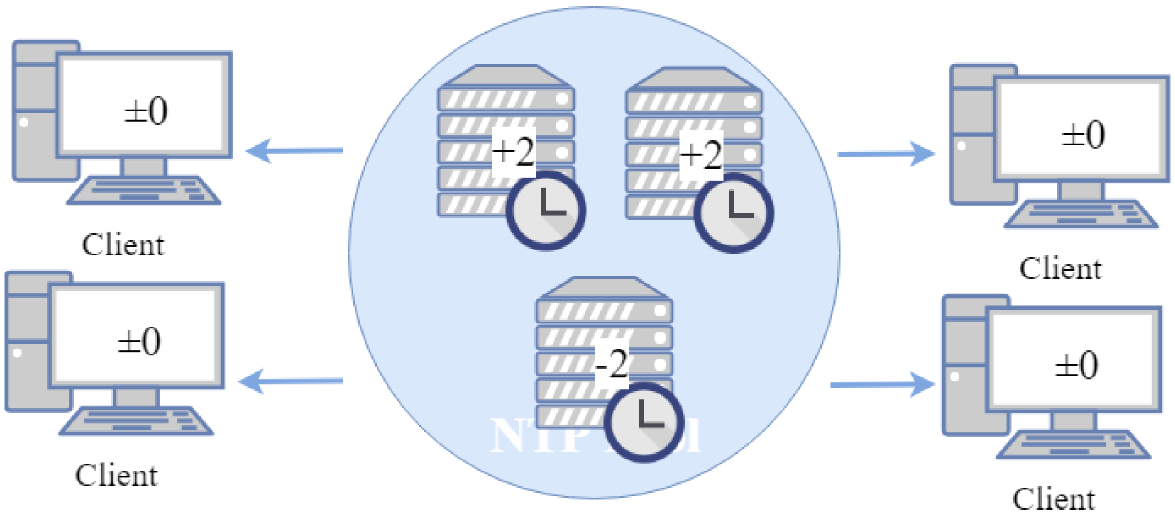
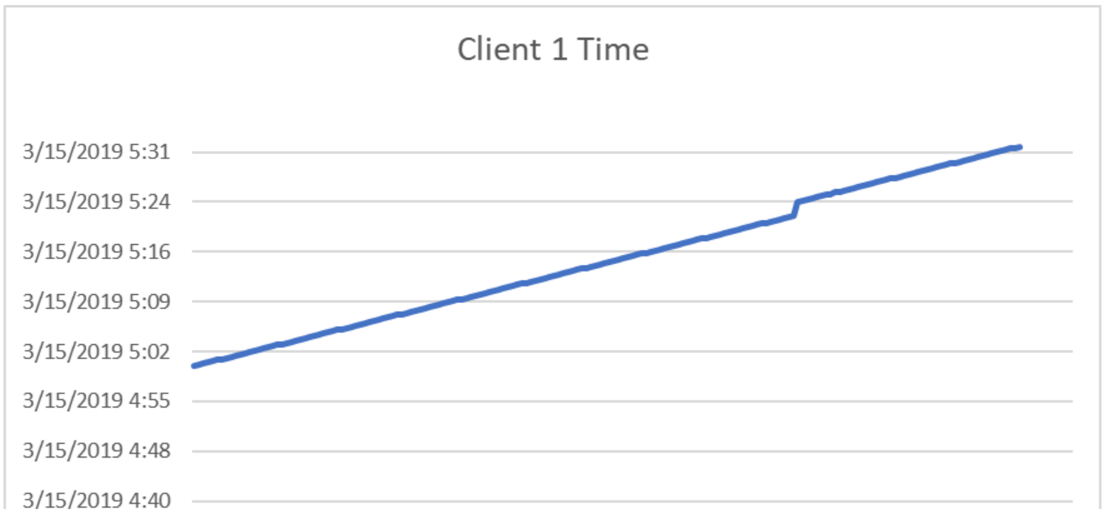


Figure 4.7 Illustration for the NTP setup. Time differences are shown in the figure.

The result NTP behavior on the 4 clients can be seen as follows:



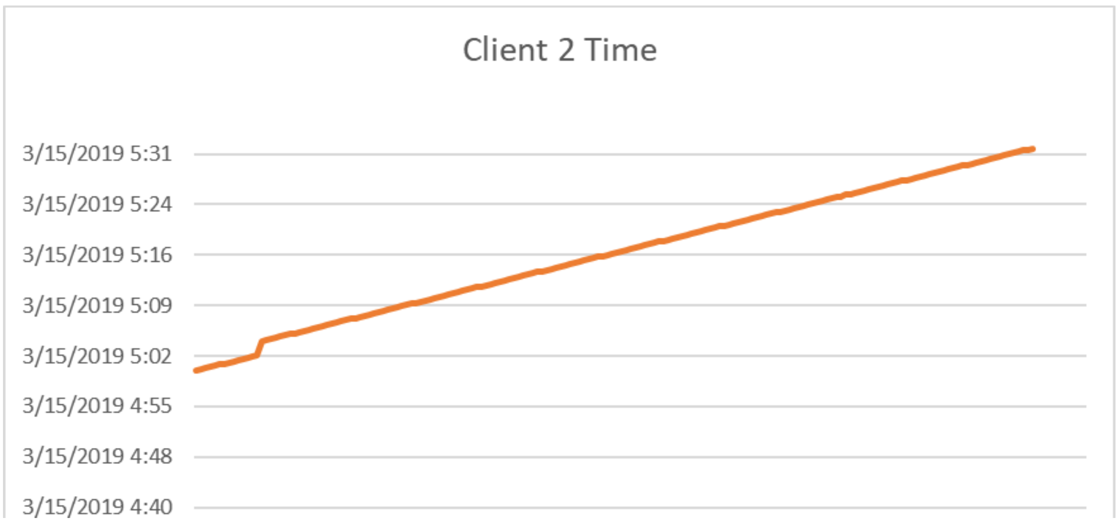
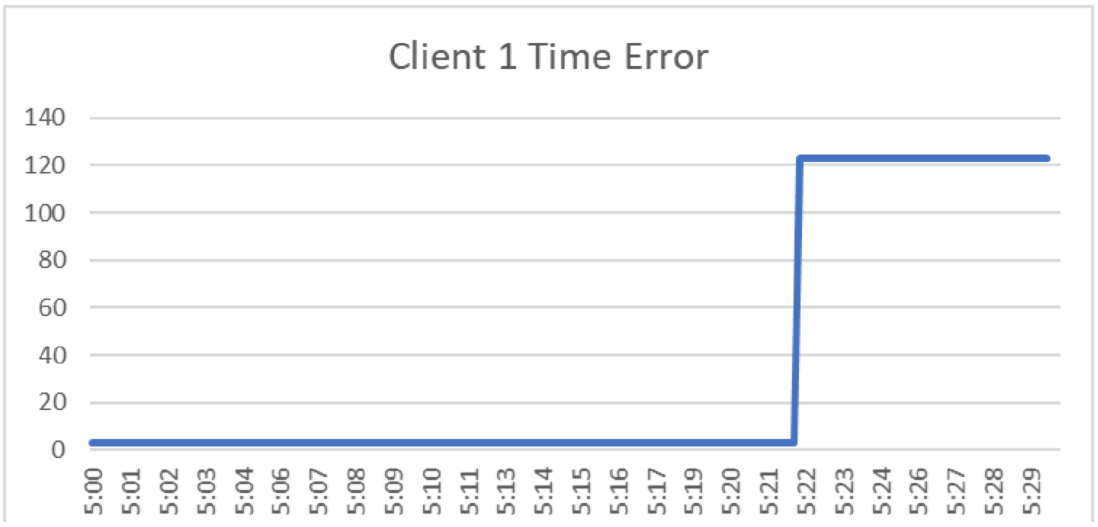


Figure 4.8 Corrected client time for client 1 and 2



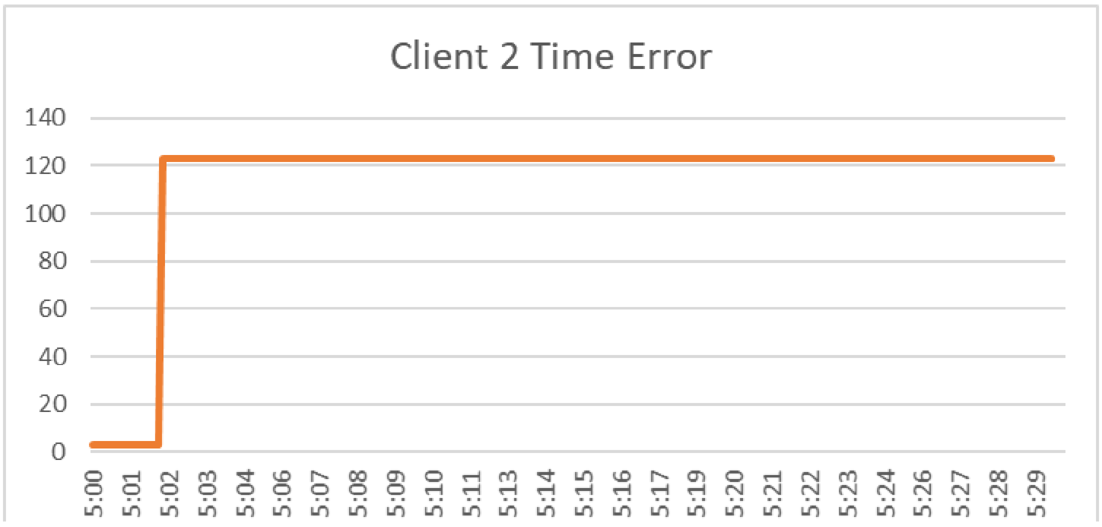


Figure 4.9 Figure 4.3 Difference between client local time and an external time source

In both client 1 and client 2, the time is shifted towards the majority of the time server in the NTP pool. There is a sudden step from the initial local time to a difference of +2 minutes. This can be explained by the time stepping mechanism of the NTP protocol. When the NTP client resolves a time of having an offset smaller than 125ms, the time will be skewed slowly to achieve the corrected time. However, for offsets larger than 125ms and smaller than 1000 seconds, if skewing is used, it takes a long time for the system time to be corrected. Therefore, the time is stepped to correct the time difference.

It can also be observed that different clients step their local time at different position. This is mainly due to the synchronize history of the client. NTP defines a stepout value. A step will only be applied once for every stepout period. This is to prevent a faulty NTP server driving the NTP client to step the time rapidly.

As the local time is stepped and a fairly consistent time error is maintained, it can be overseen that the data will behave similar to the case 1, except the time is advanced by 2 minutes.

On the other hand, the Timchain is configured as the follow:

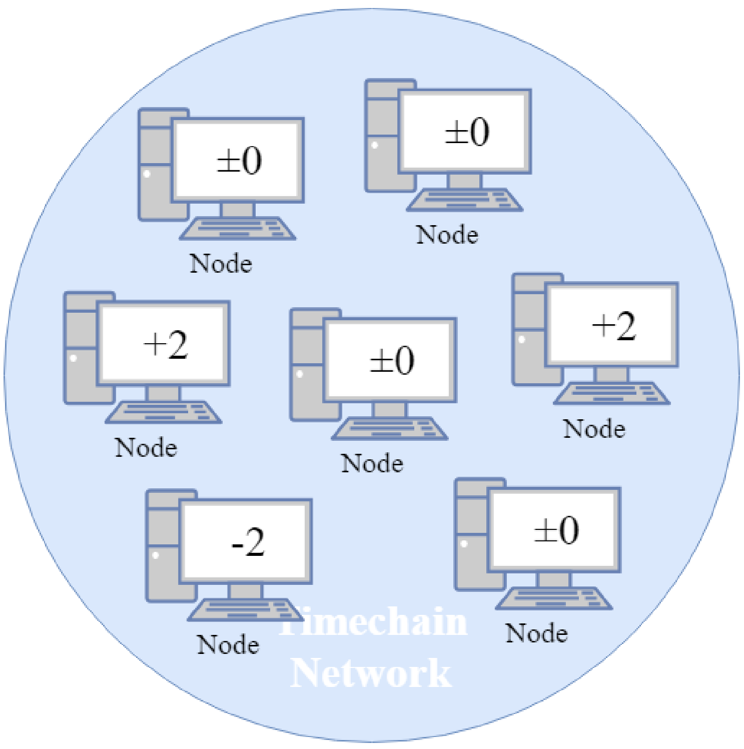


Figure 4.10 Illustration for the Timechain setup. Time differences are shown in the figure.

The blocks being generated as follows:

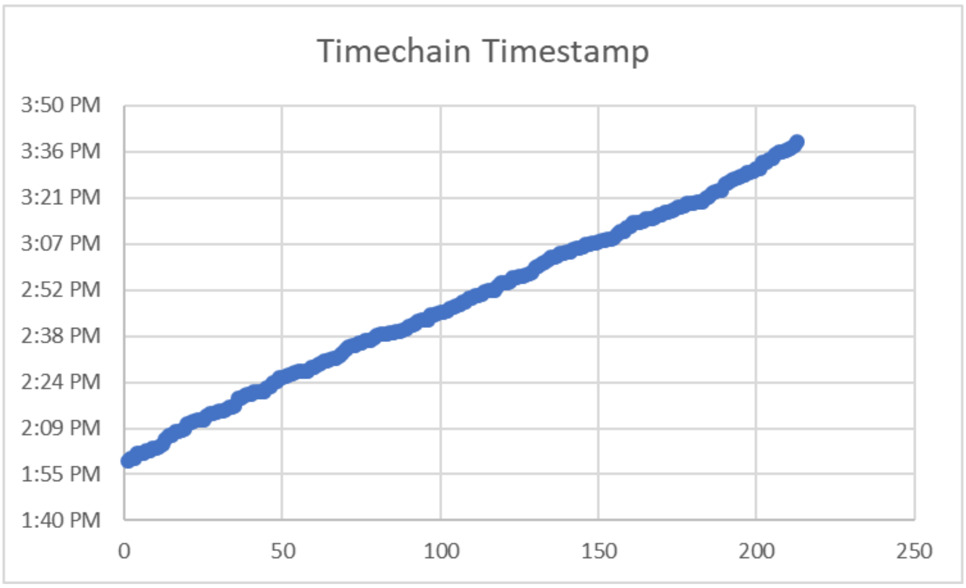


Figure 4.11 The Timestamps on the blocks

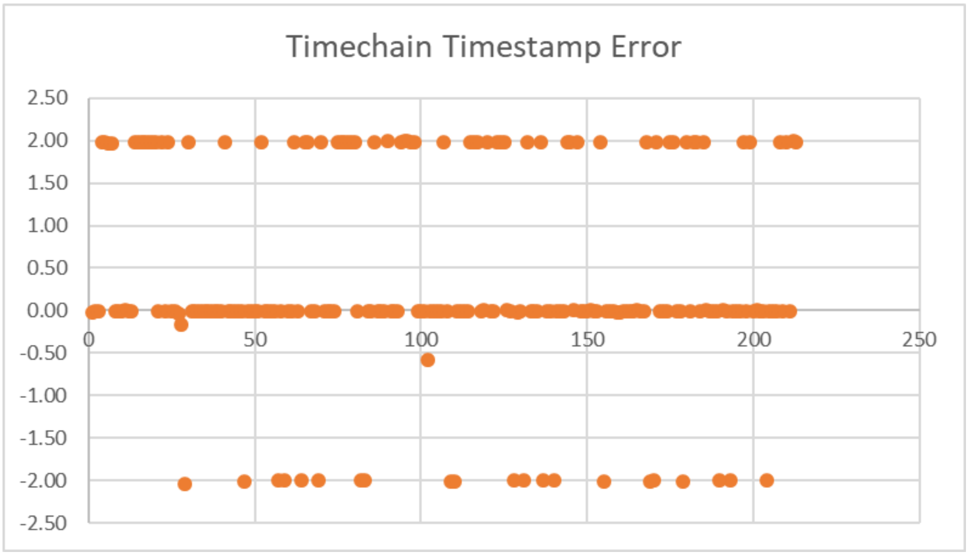
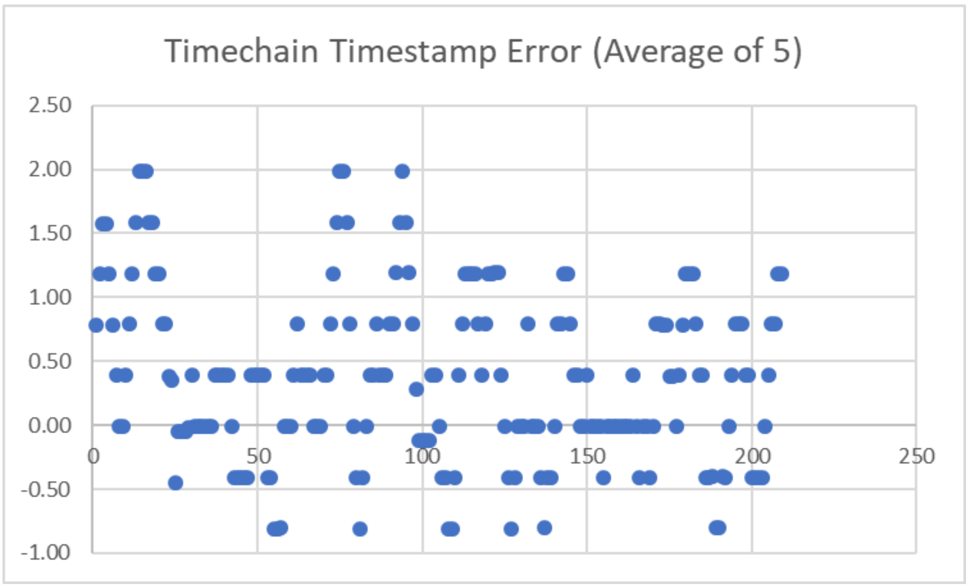


Figure 4.12 Difference between Timechain timestamp and an external time source

With the nodes having a 0 time difference be the majority of the network, it can be seen that the majority of the blocks generated have a timestamp close to the physical time. The data obtained in this test case is then used to further analyze the effect of choosing different subset of clocks to determine a time closer to the error 0.



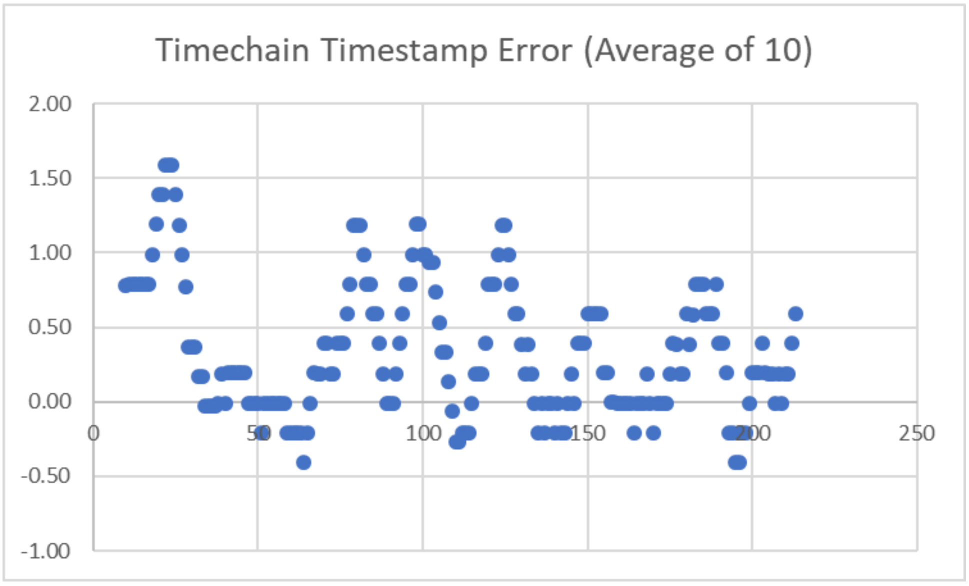


Figure 4.13 Averaged differences between Timechain timestamp and an external time source

By taking the average of the 10 most recent blocks, the effect of the nodes having an error value of 2 seconds are being mitigated. Comparing to taking the average of the 5 most recent blocks, It can be seen the upper bound is reduced from 2s to 1.6s, whereas the lower bound is risen from -0.8s to -0.45s. The operation averages out the extreme timestamps. As long as the trustworthy nodes still account for over half of the blocks being generated, by taking more timestamps into consideration, the average operation is guaranteed to coverage to a single value representing the current real time.

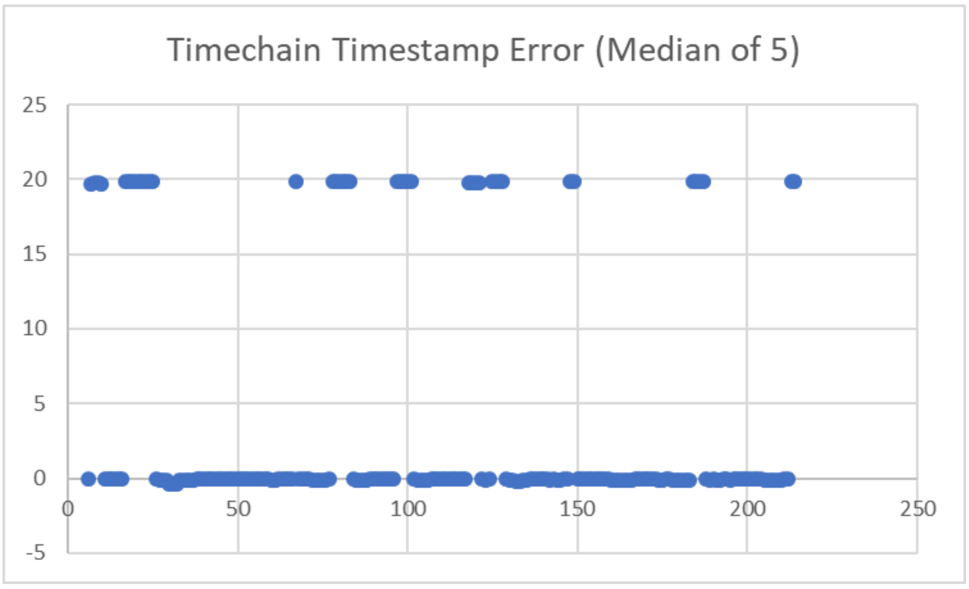


Figure 4.14 Median differences between Timechain timestamp and an external time source

By taking the median of the most 5 recent blocks, the effect of the node having time error of -2 is completely eliminated. However, this node only contributes to 1/7 of the total computational power in the network.

From the above result, we can see that by using different statistical analysis, the nodes possibly having an incorrect time can be filtered out. Usually, the time errors in trustworthy nodes are not really large. With the remaining timestamps, we can achieve a fairly precise estimation on what the real time is.

5. Improvements

The Timechain protocol currently lacks a mathematically proven mechanism for solving an accurate time among all the timestamps provided in the blockchain. In order to achieve a better time estimation, possible ways are doing in-depth analysis such as finding overlapping intervals between timestamps. It is also possible to look into how NTP clocking selection algorithm works and instead of selecting an NTP server, we can apply it to select a Timechain block in this context.

Also, for the Timechain mechanism to be deployed, more work is still needed to implement missing features, such as peer discovery and using network socket programming to reduce the possible delays caused by handshaking procedures.

6. Conclusion

Comparing to the 1st term, the work on this semester is mainly based on investigating the effect of a false NTP server on the client, and how a blockchain solution can be resilient to such vulnerabilities. The focus is more on how an authentic time can be reached through network communications, instead of the transmission vulnerabilities in the NTP protocol. I examined different consensus algorithms and time keeping mechanisms, and decided that a PoW system with a hybrid usage of both logical and physical time can achieve an improvement on the currently available timing tools, especially enabling a secure time keeping and ordering of events in untrusted networks.

The implementation of Timechain demonstrates blockchain's flexibility in adopting different data and the robustness of PoW to reach a consensus among different nodes. The testing shows that the Timechain system can obtain a fairly reliable physical time if the number of malicious nodes does not exceed 50%.

Throughout this project, I had the opportunity to gain hands-on experience on fully distributed networks. I successfully wrote my own, functioning blockchain, which allows me a more in-depth understanding on the emerging technology. Apart from blockchain based on PoW (the Bitcoin model), I also investigated different variants of the blockchain. This project allows me to gain valuable experience so that I can incorporate the concept of blockchain in my future career.

7. Bibliography

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[2] S. King, S.Nadal, “PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake,” 2012. [Online]. Available: https://decred.org/research/king2012.pdf

[3] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, pp. 382–401, 1982.

[4] M. Castrom B. Liskov, “Practical Byzantine Fault Tolerance,” 1999. [Online]. Available: http://pmg.csail.mit.edu/papers/osdi99.pdf

[5] “practical Byzantine Fault Tolerance(pBFT),” GeeksforGeeks, 14-Jan-2019. [Online]. Available: https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/.

[6] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” Communications of the ACM, vol. 21, no. 7, pp. 558–565, 1978.

[7] Lecture 4: Physical and Logical Time, Causality. [Online]. Available: https://cseweb.ucsd.edu/classes/sp16/cse291-e/applications/ln/lecture4.html.

[8] C. Fidge, “Timestamps in Message-Passing Systems That Preserve the Partial Ordering,” 1988. [Online]. Available: http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf

[9] “Vector clock,” Wikipedia, 25-Mar-2019. [Online]. Available: https://en.wikipedia.org/wiki/Vector_clock.

[10] D. Mills, "Network Time Protocol (Version 1) Specification and Implementation, RFC 1059," July 1988. [Online]. Available: https://tools.ietf.org/html/rfc1059.

[11] D. Mills, E. J. Martin, J. Burbank and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification, RFC 5905," June 2010. [Online]. Available: https://tools.ietf.org/html/rfc5905.

[12] B. D. Esham, "Network Time Protocol servers and clients.svg," Wim, September 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:Network_Time_Protocol_servers_and_clients.svg.