

Alfie Ranstead

Building Blockchain Technology

Computer Science H446-03

Candidate Name	Alfie Ranstead
Candidate Number	1317
Centre Name	Marling School
Centre Number	57037

Note to Assessor: This project has been created with interactive elements. In order to view as intended by the students, please follow the secure link below.

<https://marling-school.gitbook.io/alfie-ranstead-1/LGzOha8ERwp6uFCPw9wn/>

1 Analysis

1.1 Problem Identification

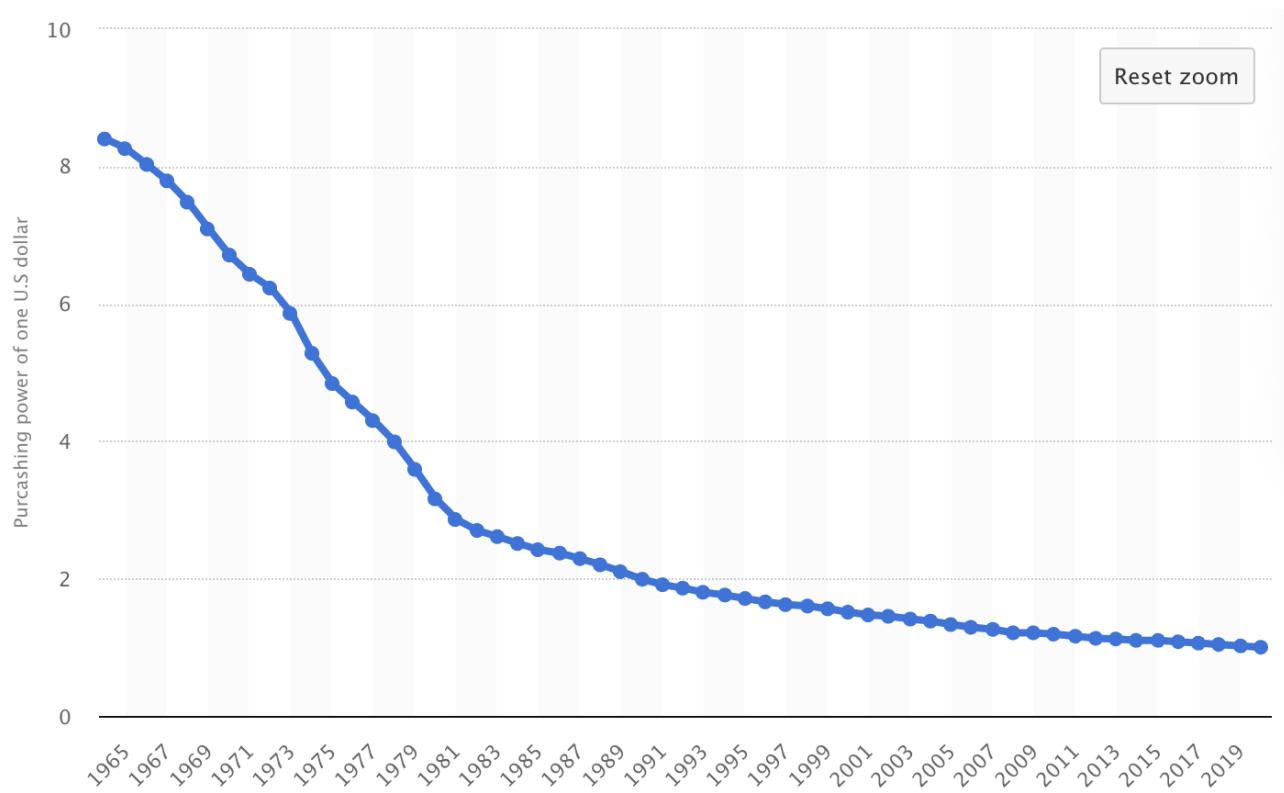
Why Blockchain Technology?

Technology is centralising industries through monopolies and oligopolies ([Hall, 2021](#)) at an alarming rate, destroying healthy competition and removing innovation in exchange for the few companies in charge's profit lines. Yet in recent years crypto currencies and other decentralised technologies have risen up in a fire of publicity and investments, sparking a new wave of technologists sharing rumours of a new generation of the internet, a generation that is decentralised and owned by its users rather than by a small group of powerful leaders that care more about the lining of their pockets rather than their users.

Although technology is definitely not the only culprit of this growing wealth (and therefore power) inequality, as ever since the dollar became a fiat currency in 1971 its value has continued to diminish. Which paired with inflation being higher than that of interest rates means that the average working or middle class citizen who chooses to save their money in a bank is actually losing money over time, rather than gaining it, and the real way that the corporate elite increase their spare funds over time is through the use of stock exchanges and investments.

Fiat money is a government-issued currency that is not backed by a physical commodity, such as gold or silver, but rather by the government that issued it.

([Chen, 2021](#))



The purchasing power of the US dollar relative to its current value between 1965 and 2020 - ([O'Neill, 2020](#))

A monetary system dominated by central banks and use of traditional fiat money has shown to benefit certain people more than others, such as bankers and moneylenders ([Thornton, 2015](#)). The traditional fiat money system is an inflationary system with a tendency to hurt the workforce and savers, resulting in hurting the lower and middle classes. It is designed to enrich those who control and operate the financial industry and the upper class ([Thornton, 2015](#)).

([Abdullah-Othman et al., 2020](#))

That is where blockchain technology comes in, allowing data to be stored on a decentralised series of nodes (user's computers) so as to give people control over their own data rather than the oligopolies and governmental bodies, and because of how this data is signed by the sender and we can track such data through the entire chain, this also allows the construction of cryptographically verified data and proof of ownership.

However, current Blockchains don't tend to have much support for actually useful data and either only have support for the currencies they were built alongside - Such as the bitcoin blockchain which only has support for bitcoin - or very small amounts of metadata. Such as the Ethereum blockchain, which relies upon all nodes that store the blockchain storing all of the data in it, or having connections to other nodes with all of the data, massively limiting what can be stored - right now this is about 0.5-1TB of data. ([Anon, 2022](#)).

This is where this project comes in, as instead of the methods of data storage mentioned above, the "monochain" (The name of the blockchain being developed in this project.) will be based upon using JSON data that can effectively store any form of data needed (by putting raw binary in a string, converting other data objects into JSON format, etc) and does not need every node to store every piece of data, instead nodes just need to be aware of enough other nodes that store all the other data, such that any individual node can always get any piece of data it needs. This will mean that initially, nodes will need to store all the data on the network as there won't be enough of them to only store a portion each, however if the network was to scale to one of a fraction of the size of a system such as bitcoin or ethereum's network then this segmented data storage method would take maximum effect.

What does this actually mean?

This project will come in three parts:

1. A protocol that enforces the rules of which to create a decentralised system of computers (who are referred to as nodes throughout this documentation) that can interact in such a way that they can trust each other enough to agree when someone holds an item/some data with value without requiring a governing central body such as a bank.
2. A node software that will run on any computer that wishes to be a part of this network, this software ensures any blocks generated by the computer running it abide by this protocol and that other nodes that it interacts with are also doing so. The way to think of this is that the protocol is like the blueprints for a city, and this node software is a house that fits into that blueprint, other people can build their own houses, as long as they fit to the blueprint of the city, but for most people a standard house is all they need.
3. A website, which will act as the portal into this blockchain for the majority of users, it will contain a basic wallet viewer, a basic transaction market for users to buy and sell items on the blockchain, a source for downloading the node software and a place to learn about this blockchain. This will be referred to as the webportal for a lot of this project, that's because it is website that acts as a portal into this blockchain, hence webportal.

1.2 Stakeholders

The whole internet: **Web 3.0**



A collection of technologies said to represent each generation of web. Source - <https://lizard.global/blog/what-is-web-3-and-how-will-it-change-the-internet-as-we-know-it>

For the past few years there have been rumours of the beginnings of a new form of web: **Web 3.0**. This form of web promises the beginnings of a fully fleshed virtual world where people own their own data, where there are no oligopolies that control the tech world and instead groups of users will have direct input into how control is distributed and how the internet should evolve. This project is not going to promise any of that, in all likelihood it will not be the start of a digital revolution that results in the destruction of Google and Facebook, instead this project will likely be a part of the slow transition towards some form of future that intends to be as the one I mentioned.

How will this actually effect the everyday person? Well, simply put, it'll change the nature of what is possible on the internet and how they will interact with and use the internet, not in any kind of groundbreaking sudden way but they will notice it more in the wave of new technologies that they can use that come along due to "**Web 3.0**". This will be like how the addition of **backend** servers and user-specific code that changed what it did based upon who was viewing the website allowed the generation of internet that created social media, entertainment streaming systems and many other systems that were not possible with non-personalised, static websites. No one actually knows what the next generation of web3 apps will look like and whether or not they will be introduced to the benefit or detriment of humanity.

Content Creators



An example of a content creator creating content. Source - <https://www.forbes.com/sites/forbesagencycouncil/2021/07/23/making-content-is-not-enough-why-you-need-a-digital-creator/>

One group who will massively affected by this shift will be content creators, as currently the measure of who owns a piece of content is whoever uploaded it to a given platform first, and if it reaches a larger scale, whoever has the most money to win a legal battle.

The shift to blockchain oriented platforms will allow content creators to upload their content to a platform which will then "mint" the content on the blockchain (likely including a smart contract clause to give the minting platform a small cut of the revenue for all future transactions) and will act as a proof of ownership for the content across multiple platforms. Although this will require platforms to agree on a general protocol in order to be able to accept each others ownership certificates, but this should hopefully be able to be implemented by having the protocol being implemented from the launch of the blockchain, which I aim to do as part of this project.

A massive benefit of this kind of content ownership is the use of something called a smart contract. A smart contract allows rules to be made so that anytime a transaction or transfer is used on that ownership certificate whatever is being done to it must abide by those custom rules. Some key rules that I aim to include in the default ownership certificate are:

- Upon mint, an owner can enforce a rule which ensures that anytime anyone sells the ownership certificate they receive a percentage of the transaction price.
- An ownership certificate can be set to expire, this is in the case of an entity wishing to sell an ownership certificate for an item for a temporary period and it would then expire after a set date. This could be used to deal with subscription based pay model for some software that would expire every 30 days and require a new certificate which is given upon the user paying the monthly charge.

Now although it is worth bearing in mind that if someone other than the original content creator mints the ownership certificate, it will likely result in a legal battle and simply transferring the certificate to the actual owner would leave the uploader's smart contract attached, which would allow them to continue to collect a cut of the actual owner's revenue when they shouldn't, so the smart contract must also be directly editable during a transaction (assuming that the sender signs this) but only sections added by the sender should be editable.

The benefits of this model, if it is implemented correctly, is that content creators will be able to not only have a certificate of ownership for their content, but also be able to sell that certificate, allowing them to "sell" their videos or even gift them. An example of this is if an investor wanted to get in on a creator's success, they could purchase pieces of content from them and hopefully gain the revenue from them in the future, alternatively if a content creator wanted to give a charity the revenue from a video they could gift them the certificate of ownership, or edit the certificate to specify to send the revenue directly to the charity if they still wanted ownership of the video.

In the latter scenario the ownership certificate could even be split up into multiple sub-sections, this could work by only allowing the current owner of each section (for example, there could be an "owner" of the content and a "revenue" holder) to change this section in a transaction, so if you wanted to change the revenue holder for a piece of content you would have to be the current revenue holder.

Developers

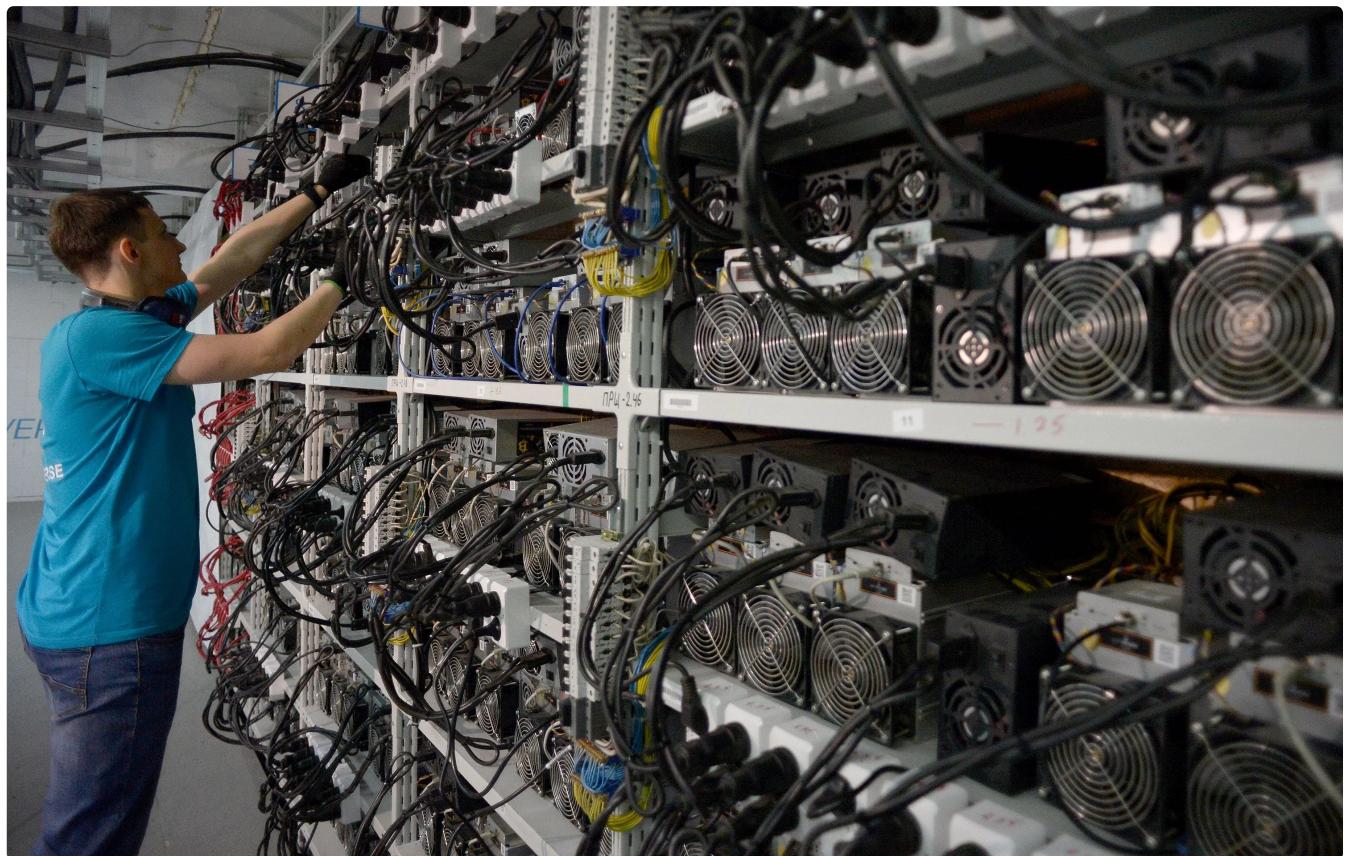


Average Developer

With the decentralised method of storage and data, developers and programmers could choose to build their platforms with the traditional methods of web2.0, or they could choose to decentralise their storage, which would be a lot cheaper initially but would mean that the nodes storing their data would receive some of the revenue they generate every time they add or remove data.

Now although it would make sense for nodes to charge developers for storing data on themselves over time as well as charging for adding and removing data, this wouldn't actually need to happen, as one of the main things that makes a node more valuable to other nodes is how many blocks it stores, because the more blocks a node stores, the more other nodes will want to add it to their pool to ensure they always have a connection to all the blocks on the blockchain. Therefore storing more data increases their usefulness to users and other platforms and in turn they will receive more transaction requests and make more money. This means that by storing more data the nodes should receive more transactions/blocks to process and get paid for, hence they are indirectly being paid more both when anyone wants to change that data and just by receiving more seemingly unconnected transactions that are due to them being more valuable as nodes.

Crypto Miners



An example bitcoin mining rig and accompanying miner. Source - <https://www.scmp.com/tech/policy/article/3005334/china-home-worlds-biggest-cryptocurrency-mining-farms-now-wants-ban>

Crypto miners are individuals who use their computers and servers to "mine" cryptocurrencies, typically being ethereum or bitcoin. This means that they are acting as nodes and giving up their computing power in exchange for transaction fees and block rewards, effectively they are being paid by the blockchain for letting it use their computer. The issue with the current state of crypto currency mining is that it is mainly used for proof of work, which works almost like a lottery, with thousands or even millions of computers trying to solve a problem and only the one that solves it first gets paid for their work, this has led to lots of miners grouping together as clusters to mine together and when any one computer in the cluster mines a block it's split amongst every computer in the cluster. This can create an unreliable and insecure scenario for people who need to be able to pay for energy bills and the running cost of their computer in order to make mining worth it for them, and if it isn't worth it then the amount of nodes on the network decreases and that's bad for the security of the blockchain.

The problem I look to solve regarding crypto miners is to make their income more sustainable through the [proof of worth](#) system. This is the hypothetical system that I designed for this project and it's advantages include that it should reward miners more frequently, in smaller frequencies and they should have to use less energy to be rewarded. This means that the [ROI](#) of their energy costs should theoretically be much higher and more consistent (although it will obviously also rely on the cost of the coin which may not be consistent).

1.3 Research

Bitcoin

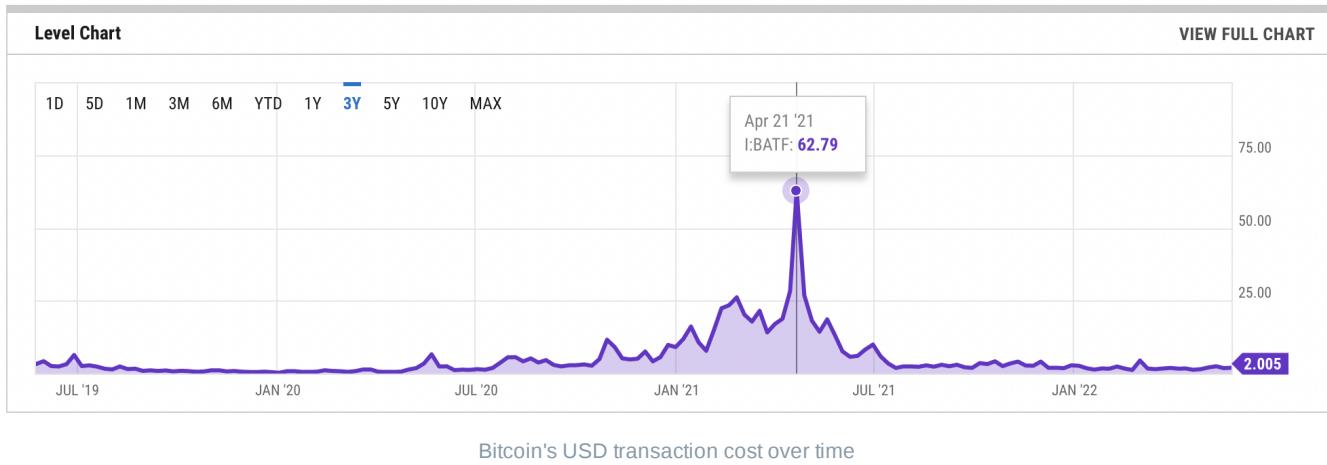


Overview

Bitcoin was created in 2008 and was the first cryptocurrency to be created digitally, it has also gone on to become the most valuable cryptocurrency per coin, being valued at £24,082.50 GBP per coin as of the 18th of May 2022, being over 10 times the value of the next most valuable per coin cryptocurrency, Eth on the Ethereum blockchain (which I have also researched below) of which is valued at £1,638.47 GBP. Although it is worth mentioning that there are approximately 120 million Eth in circulation at the point of writing, compared to Bitcoin's 18 million, which closes the total valuation gap considerably.

Consensus Protocol

Bitcoin uses [proof of power](#) as it's [Consensus Protocol](#), the benefit of this is that in order to deceive the blockchain protocol and successfully defraud the system, an individual would need 51% of the computing power connected to the network, which is feasible initially but as the network grows becomes extremely difficult to do so, especially at it's current state. The downside to using proof of power is that it is extremely inefficient and wastes a lot of energy ([for an explanation why click here](#)). This is not only bad for the environment but also raises transaction fees as nodes have to do a lot more work to complete each transaction and have to be compensated fairly.



The above chart shows this transaction fee over time, and although right now it is pretty reasonable at approx \$2 per transaction (£1.59) it is not always stable can get quite high, such as during July 2020 to July 2021 where transaction fees rose to averaging around \$10 and peaking at \$62 per transaction, an amount that makes any small transaction completely unreasonable.

Features

Features I will Include

Feature	Justification
A base cryptocurrency	This is what the majority of systems in use as their currency and acts a central point of reference. It will also likely be what transaction fees are measured in.
Some kind of transaction fee	The consensus protocol I intend to use will require nodes to require some kind of transaction fee. This is because however because it is up to the node to verify a transaction, it could theoretically complete a transaction before another node could. This means that nodes need to compete with each other to verify transactions and keep their transaction fees competitive.
Cryptographically safe transactions	This is what allows nodes to verify that a transaction actually sent a transaction and it isn't a forged or fraudulent transaction.

Features I won't be including

Feature	Justification
Proof of work based trust	Wastes a lot of computing power and time, which could be used for other, more useful purposes.
The limit of only using the blockchain for a singular currency (bitcoins) and nothing else.	Does not allow the blockchain to store other types of data, such as json object data, which is what the blockchain is based around.
≈10 minute block times	This is a good time period for ensuring the network reaches a consensus, but it limits the amount of transactions per second to approximately 7.

Ethereum



Etérium

Welcome to Ethereum

Ethereum is the community-run technology powering the cryptocurrency ether (ETH) and thousands of decentralized applications.

[Explore Ethereum](#)



The <https://ethereum.org/en/> homepage

Overview

Ethereum is the second largest cryptocurrency as of time of writing (Friday 20th May 2022), costing approximately £1,650 GBP per coin. It was launched on the 30th of July 2015 and has grown to a current market cap of £195,873,606,031.48 (~£200 Bn) in the just under 7 years it has been in use.

Consensus Protocol

Ethereum began as a [proof of power](#) blockchain but due to the environmental issues it has begun switching to a [proof of stake](#) blockchain and intends to merge the current developer network that runs the proof of stake version of ethereum with the main network that uses proof of power sometime in August this year (2022) although that date has already been postponed over a year so it may be completed at a later date. Because Ethereum is currently undergoing a transition to proof of stake and by the time you are reading this is should have been completed I will be referring to Ethereum's consensus protocol as proof of stake for the remainder of this project.

Features

Features I will be including

Feature	Justification
A consensus protocol that rewards long term users	This is very useful to keep users on the blockchain what helps to encourage long term usage and gain for those who do use the blockchain long obviously not guaranteed but it should still be the network.
Digital ownership certificates (Nfts)	This is what allows most of the more useful technology on the Ethereum blockchain and will unlock a lot of potential in this blockchain. This is how ownership is proven on the blockchain and is what differentiates it from web3.
≈10-15 second block times	This is long enough so that nodes should have time to reach an agreement on which blocks are valid, keeps transaction times low and allows lots of traffic through the network.

Features I won't be including

Feature	Justification
A consensus protocol that requires users to have a large amount of money to invest in the network in order to mine.	This greatly limits who is allowed to mine on the blockchain and defeats the purpose of diversifying the pool of miners. Alongside going against the purpose of the project's blockchain it is also very difficult to implement without a large pool of pre-existing users.

Consensus Protocols

A brief description of what a consensus protocol is and any protocols I mentioned in this project that do not have explanations in other documents.

What is a Consensus Protocol?

A consensus protocol is the protocol that a blockchain system uses to validate blocks created by nodes during the block creation stage. This is needed because in a decentralised system, the nodes (aka computers) that run the system cannot necessarily be trusted, as one node may wish to do something that benefits themselves such as paying themselves currency that doesn't actually exist. This type of node is typically referred to as a bad actor

Proof of Work

Proof of work is the most common consensus protocol and was originally introduced by the bitcoin blockchain, it is quite simple and secure which is what made it a good choice for the first cryptocurrency, however it does also have some serious issues.

It works by requiring the hash of a block that is added to the blockchain to be below a target value based upon how long it took the past

explanation for proof of power goes here ([How Bitcoin makes it harder to mine dynamically](#))

Benefits	Drawbacks
If miners are grouped into pacts the entry cost can be relatively low	Miners acting alone are extremely unlikely to be the node who successfully mines a block and can therefore spend lots of money on power/energy to receive no revenue.
Simple and hard to bypass	Costs a ridiculous amount of cumulative energy from all the mining nodes to mine each block - which is very bad for the environment.
Scales with increases in computing power extremely well	If a large entity with lots of computing power wanted to they could theoretically create a node network at least 51% of the size of the whole network, centralising it and having complete control of all the data flowing through the network.
	Nodes with extremely large computing power do not have to group up and can therefore reap all the rewards of their work alone, growing faster and resulting in a big divide between the ultra wealthy and powerful single nodes and the less wealthy, not very powerful grouped nodes that can't accomplish anything on their own.

Lots of the drawbacks were sourced from [\(Wackerow, 2022\)](#).

Proof of Stake

Proof of stake is a consensus protocol that relies upon nodes/miners *staking* capital which usually comes in the form of the native currency for a specified blockchain, such that the staked capital can act as a form of deposit. This stake/deposit can then be destroyed if the node/miner becomes dishonest or lazy, where being lazy tends to mean only validating their own blocks and no-one else's.

Benefits	Drawbacks
better energy efficiency compared to proof of work	usually very high stake requirements, requiring a much larger initial investment compared to proof of work
lower hardware entry requirements	more complex than proof of work, both to implement and keep secure
should increase security as miners are less likely to risk their stakes for fraudulent reasons.	The majority of current miners will not be able to afford the stake requirements and will have to create staking groups to share the risk

How Proof of Work Mining Works

Although this Document refers exclusively to Bitcoin, it can also applied to any other proof of power blockchain (possibly with some tweaks)

Bitcoin mining is a term given to the process of randomly generating numbers, a "nonce" (number used once) to feed alongside the block into the SHA-256 hashing algorithm so as to make the hashed value below a dynamically set threshold.

- Example SHA256 hash:

000000e943a990e64b08bd3bafc7c1b3fde497e92670f78cd8e9eb27529706f2

Because SHA-256 strings do not change in length, always being 64 hex-digits long (4 bits per hex-digit means the total is 256 bits, hence SHA-256) no matter how much data is fed into them, the value of the hash can be used to represent how much work (computing power) has been put into finding a nonce for this block.

In Bitcoin's case it requires the hash of the block to be lower than the hash target, which is a dynamic value defined by the below equation.

$$Hash\ Target_{new} = Hash\ Target_{current} \times \frac{Average\ time\ taken\ to\ generate\ last\ 2016\ blocks_{(mins)}}{10_{(mins)}}$$

Source - <http://blog.geveo.com/Blockchain-Mining->

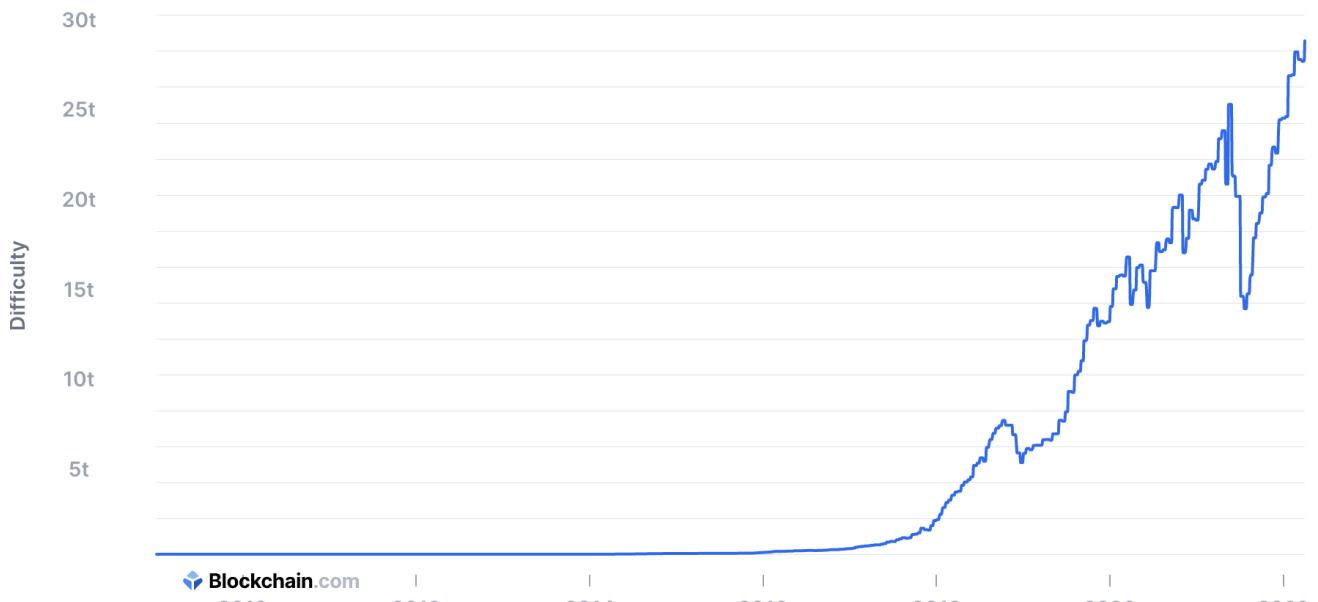
Difficulty#:~:text=The%20current%20Bitcoin%20blockchain%20requirement,not%20about%20the%20leading%20zeros.

2016 Blocks are used because if a block should be generated once per 10 minutes, that's 6 per hour and there's 336 hours per fortnight, therefore multiplying 6 by 336 we get 2016.

Therefore by lowering the target that a block's hash needs to be below, the amount of computing power to generate a nonce that meets this requirement grows proportionally. For example, that means that if the hash has to have an additional hex 0 at the start the hash target has become 1/16th of its previous value and therefore has become 16 times harder to complete.

Network Difficulty

A relative measure of how difficult it is to mine a new block for the blockchain.



Source - <https://www.blockchain.com/charts/difficulty>

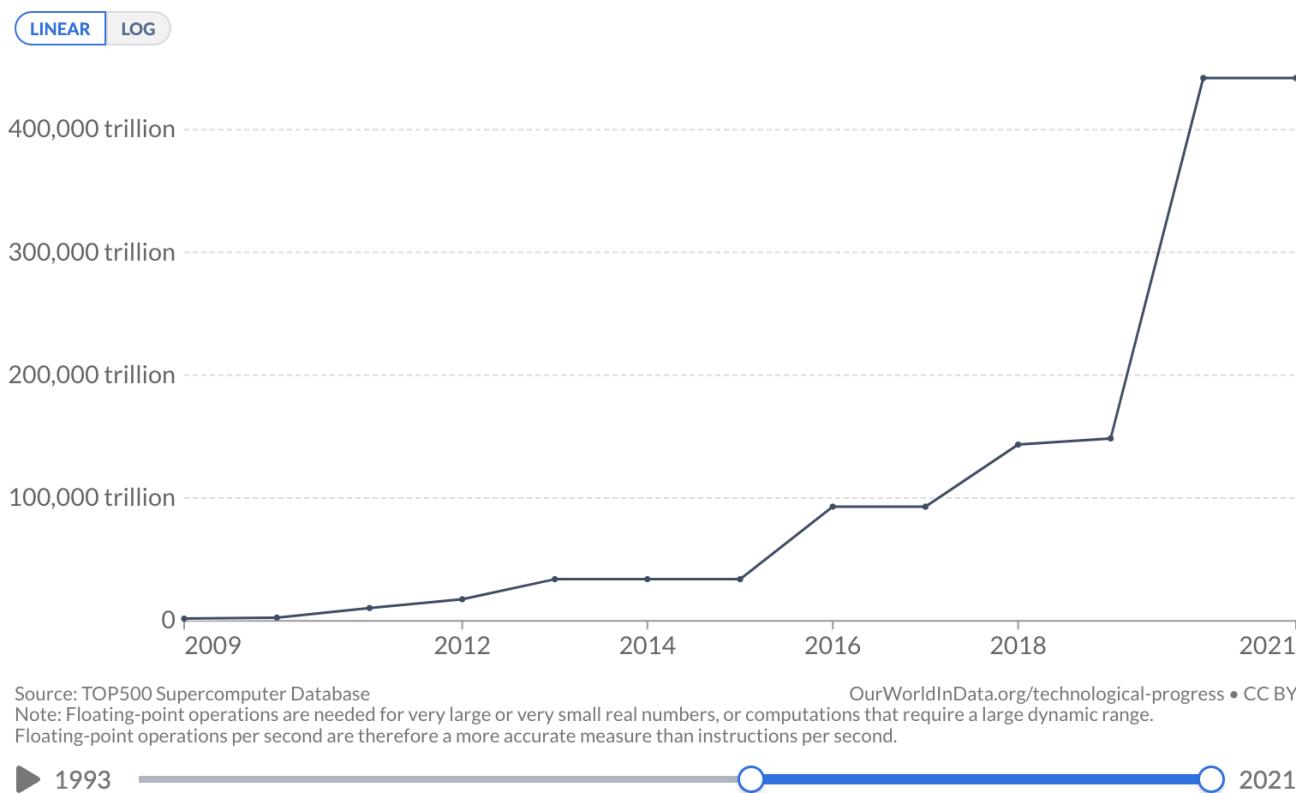
The above graph shows the relative difficulty of mining a block on the bitcoin blockchain compared to mining the first block. As of Monday 4th April 2022 it is approximately 28,600,000,000 times harder to mine a block.

It is also important to remember that alongside the popularity of bitcoin growing between its creation in 2009 and now, in 2022, that computing power has also grown substantially.

The computational capacity of the largest supercomputers

Number of floating-point operations carried out per second by the largest supercomputer in any given year.

Our World
in Data



The above graph showcasing that the computational capacity of the worlds largest supercomputers has grown by a factor of 246, a very substantial increase in power in only 13 years.

Therefore as bitcoin has shown it can scale with popularity and computing power very well, and should only stop scaling this well once it gets to the point upon which the hash of new blocks are entirely zero, which would make it impossible for the network to increase it's mining difficulty, however the amount of computing power required to do this is so large it is infeasible to imagine and it is more likely that the cryptocurrency ceases use before then. (This is because the more zeros required at the start of the hash the exponentially harder it is to generate)

1.4 Features of Proposed Solution

The solution will contain three main parts.

Due to how large this file became whilst working on it I have split the solution up into separate files for each section so as to make it easier to navigate and read this part of the project.

The Protocol

This is what describes how the blockchain will operate, how it is put together and how [nodes](#) will operate. It is what will be described throughout the rest of this section.

The Node Software

This is the software that runs on a computer contributing to the network and uses the rules created in the protocol to generate, validate and view blocks and transactions within the blockchain. Since this is effectively the software form of the protocol it will not be described in this section.

The Web-portal

This simply acts as an introduction to the project and will host some key information, a link to the documents of this project of which you are currently reading, a link to view the source code for the entire project and hopefully some other features which I will mention at the end of this section.

Limitations

Performance

Although I would like to make it so that any person with a spare computer could help compute towards this blockchain as a node and receive as much of a reward as someone with a top of the line server, due to the nature of this project being computationally complex, the more powerful a computer is, the more useful it is to the system.

This is because nodes need to not only do thousands of checks, data requests, node-to-node communications and general computations per block validation, but they need to be able to do so fast enough to be a part of the validation of a block that makes it onto the blockchain in order to actually receive a reward.

Therefore the blockchain system itself will be limited by the capabilities of its nodes and the nodes will be limited by the relative performance of themselves to the other nodes on the network.

1.4.1 The Protocol

1. Creating Data

This section of the document refers to the creation and requirements of the data that is stored within the transactions and blocks that make up the blockchain on the whole, the discussion of those transactions and blocks is done in [part 2 of this document, storing data](#).

1.1 Required Fields

Whenever new data, whether it be coins, a certificate of ownership, or anything else is created, it is essential that it contains a field that states who created it. This allows anything to be added to the blockchain as if it was a giant public database, but prevents someone from being able to claim they own something that they don't. An example of this is the main currency of this system, the mono, a mono will only be valid and acceptable if its creation flag is attributed to the system. This means that the mono must be created by the system itself, and because typically a creation will be attributed using the creator's public key/id this can only happen when a new block is generated, and such monos should only be accepted as true and valid if when they were created by the system the amount of them created in a single block is below the limit set. How this limit is calculated is described in the below section.

1.2 How many of a piece of data can be generated.

This limit of how many of a data object can be created both per block and in total can be dictated through either a constant value, a function dictated in the protocol, or algorithmically by an algorithm stored in the origin block.

Each version has its own benefits and drawbacks:

A Constant Value

Benefits	Drawbacks
Very simple to implement	Cannot change with pace of block creation of item, etc, which could lead to massive devaluing item/data.
Can be set in the protocol to prevent nodes from having to go searching for the data.	Requires an item/data it is regulating prior to the origin of the blockchain as declared within the protocol (rules of the blockchain) itself.
It is very easy to understand how regulation is going to work for a constant - meaning there's virtually no risk of letting an unknown bug ruin user's wallets/finances.	Would require a change to the protocol blockchain in order to introduce a new regulator which could lead to a fork of blockchain.

Algorithm defined in the protocol

Benefits	Drawbacks
Can be set in the protocol to prevent nodes from having to go searching for the data.	Item/Data being regulated must have been designed prior to/during the creation of the blockchain so that the algorithm can be included in the protocol.
Simple to implement	Would require a change to the protocol blockchain in order to introduce a new regulator which could lead to a fork of blockchain.
Can adapt to the size of the blockchain in a way that a constant value cannot.	If algorithm is not written correctly, it could cause massive issues for users later on.
	Algorithms are much larger than constants, so this could result in an abundance of data just regulating items/data that doesn't necessarily need to be stored in the protocol definition.

Algorithm defined in a block

Benefits	Drawbacks
Can be introduced at any point of the blockchain's life span without risking a fork.	Due to not being set in the protocol no this data must go and fetch the algorithm.
Doesn't need to be stored in the protocol, meaning reduced complexity.	More Complicated to implement
Nodes can fetch and throw away algorithms as needed, meaning they don't have to always store them if they don't use them very often. (unlike a protocol defined regulation.)	If algorithm is not written correctly, it could cause massive issues for users later on.
Can adapt very fluidly to the network's needs as it grows and fluctuates	

(crypto.com, 2022)

Due to the above specifications, I will be aiming to add support for algorithms defined in blocks, and initialise the algorithms for base commodities within the origin block, however initially the limits will just be defined as constants as it is much quicker to setup and test with.

This choice also allows for the blockchain to progress into the future without an individual to control it. This is due to the process of voting, which is something the final product would ideally contain, however since that is a feature that requires a lot of pre-requisites I will only be adding it to the project if there is time, although it will be theorised in cycles as they are reached.

2. Storing Data

The data being stored, transferred and owned on the monochain mentioned above will be structured using two fundamental concepts, the block and the transactions that make up the block.

2.1 The Transactions

Transactions represent groupings of data being sent from one wallet to another, whether that is the "system wallet" which creates currency or a user sending something they own to another user.

The Structure of Transactions and Digital Signatures

Transactions on this blockchain will abide by a few rules, rules which form the system that allow for digital signatures to take place and therefore transactions on the whole.

These rules are the following:

- Wallets (and therefore users) are identified by their public keys
- The transactions themselves are signed using a wallet's private key and then their signature and transaction combo can be verified using that wallet's public key. The point of this is to only allow the sender to create transactions where they send an item, but anyone can check and verify these transactions.
- Transactions should include a timestamp of when they were created by the sender, as the block also contains a timestamp, if the transaction is included on a block that is timestamped outside of a set time frame then the transaction should be counted as null and void.
- Transactions should then also obviously include where the coins/proof of ownership should be sent to, and what is being sent.

Following these rules gives us an idea of what each transaction should look like:

Sender's Public Key	Transactions	Recipient's Public Key	Timestamp	Sender's Signature
b94d27b9	[{type: coin, quantity: 50}]	j45n63m3	Thu June 17 2022 19:06:50 GMT	821a643d5ebf1ee9

In JSON format that would look something like the following:

```
{
  Sender: "b94d27b9",
  Transaction: [{type: "coin", quantity: 50}],
  Recipient: "j45n63m3",
  Timestamp: "Thu Mar 17 2022 19:06:50 GMT",
  Sender-Signature: "821a643d5ebf18ee9"
}
```

It's important to mention that the Sender-Signed hash of the data does not include itself, otherwise that would create a recursion error and would be impossible to destruct. Instead the Sender-Signed hash is a hash of everything except itself in a separate JSON object than the one in the blockchain.

Therefore, in this scenario the sender would be hashing and signing the following:

```
{
  Sender-Key: "b94d27b9",
  Transaction: [{type: coin, quantity: 50}],
  Recipient-Key: "j45n63m3",
  Timestamp: "Thu Mar 17 2022 19:06:50 GMT",
}
```

This is very important to bear in mind when writing the node mining software as if the whole json transaction is validated it will never validate successfully, and instead the section of the transaction referenced above is the part that should be validated according to the Sender-signed hash.

The other alternative to this is to construct the transaction object in two parts: the signature and the data. This would be structured similarly to how a typical communication request is structured, with a header and a body.

```
{  
  header: {  
    sender: "b94d27b9",  
    signature: "821a643d5ebf18ee9",  
  },  
  body: {  
    recipient: "j45n63m3",  
    Transaction:  
      [{type: coin, quantity: 50}],  
    Timestamp: "Thu Mar 17 2022 19:06:50 GMT",  
  }  
}
```

The final construction has not yet been decided, but it should abide by the above rules.

Types of Transactions and Valid Data

There will initially be four main types of data that can be sent over a transaction:

- Coin - Any type of cryptocurrency.
- Certificate - A certificate of ownership
- Creation - How things are created on the blockchain
- Custom - Any other json data, this is less strict than the other types and therefore less secure.

Coin

Coins

This is what is commonly referred to as a cryptocurrency, and is the data that represents money and allows the whole blockchain to run. There will be one key coin that will be the coin the default node software will be paid in, however if a user was to create their own node software that abides by the blockchain's protocols they could accept any kind of coin they wished as payment.

```
{
  type: coin,
  data: {
    id: "mono",
    created-by: "system",
    quantity: 50
  }
}
```

Certificate

Certificate

This represents a certificate of ownership and is what allows [Nfts](#) to be supported on this blockchain, the important thing to know about these is that they can hold multiple owners depending on which part of the certificate you are looking at, and they should only be allowed to be edited or "sent" by the wallet that owns that part of the certificate.

Creation

Creation

This is what is used to "create" something on the blockchain, if this is custom then it can just receive a check for compulsory properties (type) and then assumed to be valid, it's up to whoever is implementing it to figure out how to go from here.

For the official currency, MonoCoins, during testing the number of coins will initially just be set to 10000 per day, with the exact value to be calculated to reward miners with to be calculated from 10000 divided by the number of blocks produced in the last 24 hours, ignoring decimal places to round down to the nearest integer. This means that the minimum number of coins rewarded per block will be 1 coin per block, assuming that a block is created once per 10 seconds, which is

the fastest blocks will be able to be created initially. However during later development the blockchain should adopt some form of algorithm to control both of these parameters in a better way.

TrustElo is what is sent from a leader node to a worker node after a block has been added to the blockchain, it is calculated during the calculation of the block but since it is a part of that block it will not take effect until that block is added to the blockchain. It can be positive or negative, but has a fixed value of +1 for a useful/valid contribution and -1 for an invalid/harmful contribution, it is up to a leader nodes own discretion to add this to transactions and that's why they have fixed values, to prevent a leader node from being able to destroy/boost the TrustElo of each node in its pool if a bad actor gained control of it. This means that in theory the trust elo system only works whilst there are more "real" values being contributed per time unit than "fake" values by bad actors.

Custom

Custom

This just represents any other form of data and could be used for individual use or organisational data that is wished to be stored and processed through the blockchain. An example structure of the data that this would represent could be as following:

```
{  
  type: custom,  
  data: {  
    "Hello world!"  
  }  
}
```

2.2 The Blocks

Blocks are what construct a blockchain, they are groupings of transactions stored with a bunch of meta data that allows them to be chained together in such a way that if any block previous to a specific block is edited, it can be noticed with very few computations so as to ensure the security and confirmation of a set of transactions having actually occurred.

Storing Blocks

The more blocks a node stores, the better. However they do not need to store them all, as long as nodes know multiple other nodes who store a copy of each block there isn't any major downside to a node only storing a portion of the total blocks in the blockchain.

Yet there is a catch to this, if all nodes begin to only store the most recent blocks then the current existence of the blockchain and all user's wallet contents will be fine and continue as normal but the transaction history for the blockchain will be erased.

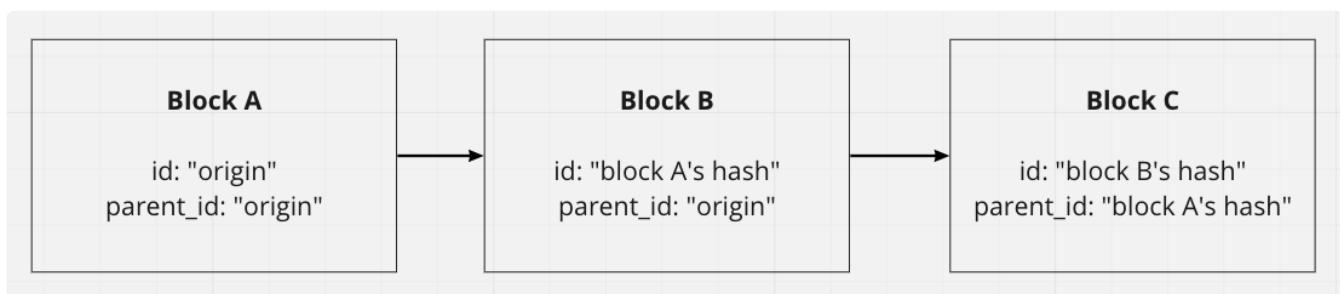
This could be looked at as a benefit of the system because it technically increases anonymity whilst retaining function, but in certain cases it could be an issue. The reason this is an accepted risk is because the storage gains and general computational benefits are great enough as a network like this scales and this risk is low enough that it is a worthy risk to take.

Chaining Blocks

The blocks will need to be chained together in such a way that if a single one of them is edited, all blocks which connect to it prove that it has been mutated and the edited block was not the original block of which they connected to. To do this, the blocks will contain an id field, this id will be the hash of the previous block's and they will also include a parent_id field, this is the id of that previous block.

This means that the parent_id can be used to find the parent block of which a child block is chained, then by hashing the parent block and comparing it to the child block's id it can be validated that the pairing has not been tampered with - or if the hash does not equal the child block's id then the pairing has been tampered with and the chain has been invalidated.

The blocks should also contain the number block that they think they are, as this makes validation and navigation easier for nodes, as if a different node supplies them with block 347 in response to a request when the node thinks that there's only 43 blocks in existence then they need to either confirm the larger quantity of blocks or disregard the other node as untrustworthy.



Ensuring Blocks are secure

This is vital to the survival and usefulness of a blockchain, and in the case of the MonoChain will be done in two ways:

1. Identifying blocks using the hashes of their parents to ensure if a block is tampered, all hashes after that block become invalid and the chain is broken.
2. [The Consensus protocol "Proof of Worth"](#) is used to cut down on the amount of times each node has to validate transactions from each other node.

The first method, identifying and chaining the blocks using hashes is detailed in the "["chaining blocks section"](#)" further up this page.

The second method, is a custom designed Consensus Protocol called "Proof of Worth". ([This protocol can be seen in a technical bullet point summary here](#)). The base concept behind this protocol is to use a trust factor based system that stores, monitors and changes the values of each node's "trustworthiness" within an integer value called that node's "trust". This value can be altered up or down by one unit per each block, with the default node mining software being designed to give a node that helps it a trust increase and a trust decrease to nodes that lie to it.

This then means that as nodes make positive contributions to the blockchain their trust ratings will increase and as bad actors make negative contributions their trust ratings will decrease. Hence when a user wishes to create a transaction, they will be able to choose to send their transaction to a well rated node, which will likely have an increased fee and waiting time, or a worse rated node, which will likely have a lower fee and waiting time. However this also increases the risk of their transaction not be validated properly and require re-sending.

To learn more about the custom consensus protocol being hypothesised, I have created a bullet point summary of the theory of how it would work in the file "[Proof of Worth - the bullet point summary.](#)"

State Root

This is a concept that has been inspired by Ethereum's feature of the same name and allows the state of all wallets and stores in the blockchain to be tracked simply using a hash of the current state of the entire system in each block. The idea here is that nodes should hold the latest copy of all the contents of every user's wallets, and that a hash of this state root should be stored in every new block such that when a node connects to the network they can get collect the state root from a different node, verify it to the latest block's state root hash and if it is valid, then continue computing from there.

This massively lowers the amount of computations a node has to do per transaction validation, because without this nodes would have to travel back throughout the entire blockchain looking for proof of ownership of an item that is to be transacted.

(Ethereum, 2015)

Referencing Owned Data

Whenever something happens to a piece of data that requires proof of ownership by a user, such as sending it to someone else, the node calculating whether or not this is valid must travel down the blockchain until it finds enough of that data being sent to the user as the user is attempting to send to someone else. This is to stop users trying to duplicate data because it shows the user actually has the data/items they want to transfer currently in their wallet, but it could result in a node travelling throughout the entire blockchain if a user tries to send an item that they don't own.

1.4.1.2 Proof of Worth - the bullet point summary.

This is the original concept for proof of worth, my custom designed consensus protocol for this project.

(i) This is just a summary.

This is the original concept for proof of worth, my custom designed consensus protocol for this project. The point of this is not to be fully fleshed out and just shows the original planning for this protocol which will be elaborated on in later development cycles.

In order to create the proof of worth census that this blockchain will run on we need to first initialise the first Leader node, where a leader node is just any public accessible, static nodes that allow worker nodes to connect to them privately. Read about leader vs worker nodes in more detail further down this page.

- Initial leader node is created.

- New worker nodes send a request to the leader node to be put into the active worker nodes pool.
- These worker nodes are added to the active worker nodes pool, with a predetermined constant "trust factor" (this represents their trustworthiness) and because these worker nodes are attached to a wallet in order to receive their payments they can also just store this "trust factor" in their wallet,
- **(IMPORTANT - The final blockchain protocol will not allow for the transfer of this trust factor by the node who's trust factor is being transferred and can only be changed by an integer value of 1 per block).**
 - This effectively means that when a new block is mined, the block creator can change any other node's trust factor by 1 unit, whether that is +1 to represent a positive image or -1 to represent a negative image.
- Now that we have a leader node and a pool of active worker nodes with a trust factor ratings we can start putting blocks on the blockchain, to do this a user sends a transaction request to a leader node, and first the leader node checks that the amount designated to itself is above the percentage of the final transactions that it wants, then it is piled into the leader node's pending transaction pool.
- Once the transaction pool is large enough that a block can be constructed from its contents the leader node announces that it has a new block that needs work and publishes all the contents of the block to each worker node.
- Each worker node then starts validating all of the data it has received and then constructs the block using only valid transactions, and sends this block back to the leader node.
- The leader node then awaits for enough nodes to send back their data so that it has a large enough quantity of the calculated block but not too many so as to prevent the user to have to wait too long for a transaction to go through.
- Now that the leader node has a large quantity of what should theoretically be the same block, it compares all of the worker node's submitted blocks and if over a large majority (90%?) are identical then that will be the final block to be submitted to the blockchain.
- The leader then broadcasts this block to the blockchain and it is added.
- This then completes all of the transactions which were pending and therefore also results in the leader node receiving all of the transaction fees that it required.
- Now that the leader node has "the block's mining revenue" it needs to distribute it to its worker nodes, which it does so by first taking its own cut (this is up to the leader node's own choice but will be 25% on the initial example node software).
- Then it takes the remaining pile of crypto and distributes it equally to every node that helped construct the block

Leader nodes vs Worker nodes

The reason for the leader vs worker split is to allow as many computers as possible to be able to contribute to the blockchain whilst ensuring they are still rewarded for their work.

The leader/worker dynamic helps this as the requirements for becoming a leader node are about as high as typical for becoming a block contributor for other chains, whilst the requirements for becoming a worker node are much lower.

Requirements for each type of node

~~Requirements for each type of node~~

The approximate requirements for a leader node will be the following:

1. An internet connection with low enough latency to support quick computations and data transfers.
2. A computer/server powerful enough to handle large quantities of data quickly.
3. A public DNS/IP address to allow worker nodes to connect to.
4. Two open ports, one for http and one for web-socket connections, the http port for communicating to other leader nodes and the web-socket port for talking to worker nodes in realtime.
5. [Ideally] Able to run the node software 24/7 so as to ensure that worker nodes that choose to work for this leader node can always mine whenever they want. Although there's no way of actually enforcing this but it will help the leader node grow their personal connections so is somewhat socially enforced.

The approximate requirements for becoming a worker node:

1. An internet connection for connecting to a leader node.
2. A computer that supports any form of the node software.

Summary

A leader node is one that operates at all times, allows worker nodes to connect to the network through itself and has a publicly accessible http and web-socket connection but will be rewarded for these requirements with a greater share of block rewards.

A worker node is one that can operate whenever it wants and will contribute by helping a leader node do any calculations they need doing, then will be rewarded with a lower share of block rewards.

Benefits of this system compared to the alternatives:

Bitcoin

Does not waste as much computing power to "mine" a block as a proof of work system such as bitcoin, as it requires an ideal pool of 100-1000 worker nodes per block unlike bitcoin which relies on a pool of tens of thousands of nodes for every block.

Ethereum

Does not rely on worker nodes being owned by users who already have a large investment in the system

unlike a proof of stake model, which in Ethereum's case is a massive hurdle for nodes. "To participate as a validator, a user must deposit 32 ETH into the deposit contract and run three separate pieces of software: an execution client, a consensus client, and a validator." ([Wackerow, 2022](#)) At time of writing 32 ETH is £52657.46 GBP which is absolutely ridiculous hurdle and is the reason why Ethereum only managed to implement this recently which masses of users as opposed to when it was conceived.

1.4.2 The Node Software

The point of the node software is to enforce the rules of the protocol and allow a computer to contribute and use the monochain, therefore the features of this software can be split into two categories: communication between computers and enforcing the protocol rules.

Communication

The communication layer needs to allow both the worker and leader nodes to communicate such that the same software can be used by either type and only require a configuration change rather than a separate piece of software.

One possible way to do this is through the use of separate websocket and http servers built into the software. This could be done through other methods but it is the simplest and most commonly used solutions so that will be what is used for this project.

This then means that a '[leader](#)' node will need to host a http server to communicate with other leader nodes and a websocket server to allow 'worker' nodes to connect to it. The benefit of this is that the worker node does then not have to host any form of server itself and can instead just connect to a leader node through the websockets module of the software, meaning that the node can also be run from anywhere regardless of if the computer it is being run on is publicly or privately accessible.

This then means that in order for communication to work properly across the network there will need to be a solid and robust 'communication layer' which acts to communicate between protocols across the entire network and as such this will be almost all of the initial development since the protocol stage cannot be built until this is finished. It is expected that the communication layer will be approximately half of the development of the entire project.

The Protocol

Once the communication layer is implemented and sufficient, the protocol can be implemented within the software. This will be done by taking the rules defined in the documents relating to the protocol and any new rules planned in further development cycles and implementing these rules to the data structure of the communication and storage of data for the blockchain.

This means that it will be important to write the node software in a language where the types of the data can be clearly defined, such as typescript, so as to prevent any errors in accidental definitions of incorrect data types that do not perfectly follow these rules.

1.4.3 The Webportal

Summary

The webportal will be how users of all types will first interact with the blockchain and its components, it will be used to introduce users to the project, give some basic uses of the blockchain and give further information to those who want it.

Since the webportal has quite a wide range of possibilities, I will be building it in two stages:

- Stage 1. Primary Sections
- Stage 2. Advanced Sections

In this case the primary sections will be developed early on in the projects development and are essential, with the secondary sections being important but not being truly essential to the project's use and mostly reliant upon other parts of the project having reached a specific stage of development.

Stage 1. Primary Sections

Initially the following pages can be built into the webportal that are not reliant on the node software or the protocol being at any stage of development and are primarily information based pages.

Home Page

This will be the page users first see when they open the site and will contain a very brief summary of the project along with some method of navigation to get to other parts of the project/site quickly.

'Learn More' page

This page should be built to explain some of the more technical side of the project in more detail such that technical users who are interested in learning about the inner details of the project and its structuring can do so.

Other Navigation

There should also be some form of link to the source code as well as any other relevant external documents so that users can find anything that is relevant to the project from this webportal.

Stage 2. Advanced Sections

Then as the project progresses and reaches a point at which transactions can be completed through the use of the node software the sections in this stage should begin to be implemented.

Wallet Page

This will allow users to generate or load their wallets to see the contents and what they are storing without having to launch a node and check manually. This will also help to reduce the entry point of the project for non-technical users since it is much easier to login to an online wallet than use a dedicated software one.

Marketplace Page

This will allow the purchase and sale of data, currency and other blockchain items to other users on the network. This is important since initially there will be no third party tools or marketplaces for this blockchain and hence in order to make it useful and demonstrate its capabilities there should be a very basic example of a marketplace that could then be elaborated by other developers if they saw it as interesting enough to warrant doing so.

1.5 Computational Methods

Computational thinking is a type of thinking used to figure out how a problem can be solved. The process of computational thinking can be summarised as formulating and thinking about a problem from such a perspective that it can be solved algorithmically and thus can be solved through the use of creating an algorithmic solution that can be reused on similar problems.

Using the five main computational methods subtitled in the sections below allows the creation of a computationally efficient and reusable solution.

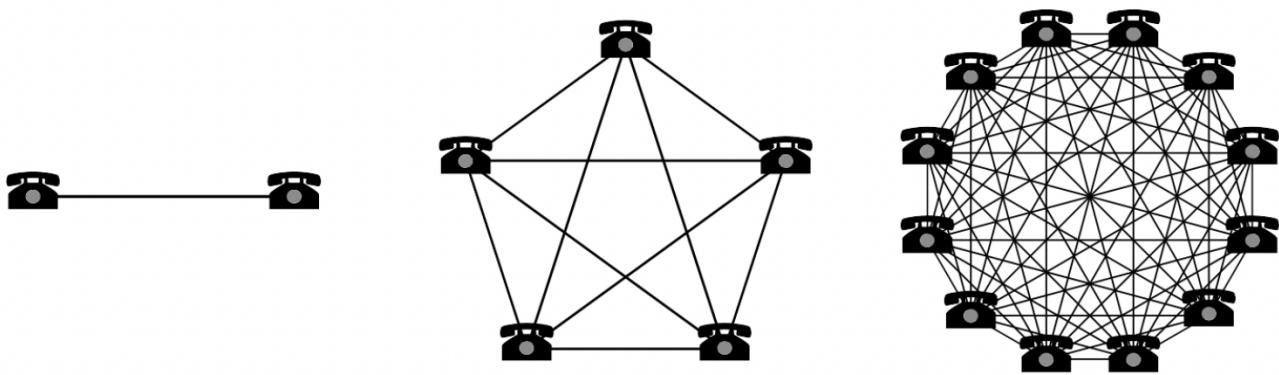
- It is extremely hard to create a decentralised currency system based in the physical world, so hard in fact, that it's effectively impossible, this is why the solution is a purely digital one.
- Ownership certificates must be publicly available so that anyone can see who owns anything on the blockchain, this means that these certificates must be easily and always accessible, which is why they will be hosted digitally, and any node can request the data from any other node.

Thinking Abstractly and Visualisation

Abstraction is a term used in computer science and computational thinking to refer to the process of removing unnecessary details from a problem in order to focus only on the core details that actually matter when creating a solution to that problem. Thinking abstractly is very important in blockchain design because each node may hold thousands of blocks in its storage and each block on a blockchain has the potential to hold thousands of transactions which may hold multiple data transfers. This means that any unnecessary data stored on the blockchain can be amplified massively in the final product.

It's also very important when designing the blockchain so as to remove a lot of the complications of an over-engineered and unnecessarily complicated development cycle and let you focus only on what actually matters so as to create a working and (hopefully) finished network.

- The base transaction protocol, this needs to contain the main types of data transfer supported by the network, but attempting to solve for all possible types of transaction would be nearly impossible and would result in a lot of excess rules that would only be used in very very rare scenarios.
- Communication within nodes, it's important that this communication system ignores any excess bulk data that could be seen as useful but actually isn't that important. This is because as the system grows, the amount of inter-node communications increases exponentially (similar to the network effect).



A diagram showcasing the network effect and how it affects the growth of communications within a system. Source - https://en.wikipedia.org/wiki/Network_effect

Thinking Ahead

Thinking ahead is the computational method used to consider what requirements the problem's solution will require and how the current and early stages of the solution will need to build towards those requirements. Thinking ahead is important as it allows you to figure out roughly how the blockchain should look and the key features it needs to meet the problem identified. This is then also useful for figuring out what the approximate steps should be to reach this solution.

To ensure my project is finished in time and meets all of the requirements I have set myself in the [features of proposed solution](#) I will need to ensure that I plan how I'm going to meet all the requirements very early on, as everything in this project will be based upon a few concrete ideas that act as the foundations for the code and general protocol.

An example of this is the consensus protocol, which is a very key point to how the blockchain functions and because the consensus protocol I have decided to use is designed by myself I have to figure out how I want the general process to work in order to implement this in the rest of the blockchain. [This can be seen in my bullet point summary of the original concept for Proof of Worth.](#)

Thinking Procedurally and Decomposition

Decomposition is the process of splitting up a major, large and complex problem into lots of minor, little and simple problems that can be procedurally solved in order to build up to solving the major, large problem.

Procedural thinking is extremely important in blockchain design as the entire system is built upon a decompositional structure (The blockchain splits up into blocks which split into transactions, etc) and in order to build a blockchain system the structure must be built procedurally in order to ensure that each structure works with its substructures.

During the design of the blockchain system, I will be focusing on individual aspects of the system itself, which should build together to make the whole blockchain work as required. The requirements for this system are defined in my [success criteria](#), with some specifics being defined technically in the [Features of proposed Solution](#).

Thinking Logically

Logical thinking is used to understand and decide what inputs/conditions will create what outputs/outcomes. This is very important in blockchain design as if there are any logic errors it could mean that some transaction or block should be classified as invalid but is classified as valid (or vice-versa) could result in someone losing or gaining ownership of an item/currency that has real world worth when they shouldn't have.

- When a bad actor attempts to pass an invalid transaction or block as valid it should be identified and not added to the blockchain.
- When a transaction is requested it should be ensured it is valid and meets the current protocol or it could cause an issue for the user.
- When a node identifies an invalid item it should be classified as such and thrown away, rather than being allowed to stay in the system regardless of its invalid status.

Thinking Concurrently

When designing a blockchain, it is very important the software which runs it is designed to be able to handle multiple processes concurrently. This can be achieved either through the illusion of switching between many processes rapidly, which works in modern systems due to how fast the modern computer can complete the average program's requests; or can be achieved through multi-threaded programming.

For my system, there are two key components, and both require concurrent programming.

The web portal

This can be concurrently programmed quite easily, as it is a website running purely on a frontend server so will just send the required code and files to the user's computer for them to execute, effectively running it concurrently on all systems that visit the site at the same time.

The Node Software

The node software will need to be able to process transactions being sent to it, answer incoming connection requests from other nodes and a bunch of other processes, all at once. The way this is solved is through creating the node software to work similarly to an api server, this means that every time any other node or service communicates with a node, it will generate a new thread to process this message and deal with it concurrently to any other processes.

1.6 Success Criteria

Node Software / Protocol

Individual blocks

Ref. No.	Criterion	Justification	Reference
1	Blocks must refer to the previous block in the chain.	This prevents any user from being able to insert a block in-between two pre-existing blocks and is what forms the "chain" itself.	The Blocks
2	Blocks must be detected as invalid and disregarded by the node software.	If the node software cannot tell when a block is invalid, the whole system is effectively useless as if a bad actor manages to bypass the initial validation they could successfully hijack the whole system	The blockchain
3	Blocks must be detected as valid and approved by node software	If the node software mistakenly disproves of valid blocks they could punish good nodes/users, discouraging them from continued use of the system.	Keeping Block Creation Secure

Network Communication

Ref No.	Criterion	Reason
4	Nodes should have an open api server running using http that contains enough pathways to complete all the needed types of communication.	This is what allows nodes to communicate which results in how the computations will be shared and how blocks will be updated to nodes around the blockchain
5	Node should have web-socket connection abilities for server or client connections.	This will allow 'private' nodes who do not have a public dns/ip address to connect to the network without having to set up a public address like is needed for a http server.
6	Nodes should be able to add and remove each other to their communication lists which should be stored locally	This prevents a node from having to rebuild its network every time it connects to the network, and instead just connect to the nodes it already knows
7	Nodes should be able to store a local "grudge" object that just remembers who the node has a good or bad experience with to change their trust level when they create their next block.	This allows Node to remember who they've found trustworthy and who they shouldn't trust again so they can send things that require higher security (such as transaction initialisations) to more trustworthy nodes.
8	Nodes should send through a digital signature with each communication.	This ensures that each node knows who they're talking to and ensures that they don't hold a grudge against the wrong wallet.
9	Nodes should be able to remember each other through a combination of their wallet address and their ips/domains	This is so that when a node wants to communicate with a node running under a specific wallet they can communicate with the ips and domains they have associated with that node in the past instead of having to ask around other nodes until they find an active communicator to that wallet/node.

General features

Ref. No.	Criterion	Reason
10	A configuration handler	This allows for the loading and saving of node settings so that the node doesn't have to be setup up every time it launches.
11	The configuration handler should be usable and accessible to technical users with non-technical users being able to use it with additional help.	This ensures that technical users can custom the node software to work in any way in which they wish for it to work, with non-technical users being able to do the same if they really need to through the use of some external help or guides.
12	Nodes should have some kind of web based access dashboard.	This allows the user to see what their node is currently doing and interact with it directly.
13	The node dashboard should be easily accessible and usable by all forms of users.	This is the most basic side of the node software and it is essential that it is easily usable so that users of any type can understand whether their node is running and if so can interact with it as easily as possible.

Non-functional

Ref. No.	Criterion	Justification
14	The node software should not crash.	Since the idea of this software is that it should be setup, launched and then left alone for long periods of time whilst it completes calculations and transactions, hence a crash of the system could cause the software to be offline for long periods of time.
15	The software should be capable of receiving multiple messages per second.	This is such that the software is capable of handling multiple connections and sending data at a rate at which matches or exceeds the rate at which data is inputted into the system so as to prevent a backlog of requests from being formed.
16	The software should attempt to retry any failed processes (such as loading a file) and should only exit after multiple failed attempts.	Since certain processes are bound to fail occasionally, it is important that the node software re-attempts these processes a few times to ensure it is not due to itself and is instead due to something outside of the software's control (no internet, not setup properly, etc)

Webportal

Page types

Ref. No.	Page	Justification
17	A "home" page that summarises what the project is about.	Gives users a quick and brief explanation of what the project is so they can decide whether or not they want to continue using it.
18	An "info" page that explains the project in more detail.	This will give users a more detailed understanding of what is actually going on in the project and how it works, whilst attempting to solve any questions that may have been generated by the home page.
19	A "downloads" page that lets the user download the node software.	This is how users will get access to the node software and will help contribute to the network, thus is essential for increasing the ease of use of the project to a reasonable degree.
20	A "wallet" page that allows the user to see any data or items stored within their digital wallet.	This makes it much easier for casual users to see what they 'own' and thus interact with the network from a browser without having to download anything.

Non-functional

Ref. No.	Criterion	Justification	Reference
21	Non-technical users must be able to identify what the point of the project is just from the homepage of the website.	<ul style="list-style-type: none"> • Stakeholders include typical internet users. 	Stakeholders
22	Users should be able to get to the majority of what they would want to get to within 3 clicks.	The easier it is for users to get to where they want to go, the more likely they are to do whatever they're trying to do.	Stakeholders
23	The web-portal should be available and working on a variety of device sizes including mobile and desktop.	This allows users with any kind of internet enabled device to be able to use the site, as users tend to be distributed across different device types.	Stakeholders
24	The web portal should run without crashing under any circumstance	Since the web portal will be a website and ran upon a public server, if it were to crash, a major stakeholder (the whole internet) would not be able to get access to the site.	Stakeholders

1.7 Hardware and Software Requirements

Development Requirements

The following are the requirements of a system in order to develop for the blockchain system described in these documents.

Webportal Development

Software used	Reason for Use
Yarn	A package manager used to install and run the necessary packages used by this repository.
Node.js	A javascript interpreter used to actually run the software.
Google Chrome	A web browser is needed to test the webportal and ensure that users can actually do everything they need. (Access their wallet, download the node software, etc)
Webstorm	A programming environment used to write the code, acts as a text editor as well as having a terminal component for developing, building and packaging the code.

Node Development

Software Used	Reason for use
Yarn	A package manager used to run the necessary commands and build scripts for this software
V Compiler	This is the compiler for the programming language V, which is what the node software is written in
VS Code	This is a programming environment that has support for the Vlang coding extension and syntax highlighting
libssl-dev	Required to build to binary on ubuntu using openssl 1.1.1
libpq-dev	Required to build database packages to binary using PostgreSQL.
Mingw-w64	Used to cross-compile the v program into a windows executable

The below guidelines are based upon all the tools specified in the software used above and

Hardware	Minimum Hardware Requirement	Recommended
CPU	4 vCPUs	4 Cores
Memory (RAM)	2GB	8GB
Storage	25GB	25GB
OS	Windows 7+ (64 bit)	Latest 64 Bit Version of Windows, macOS or Linux

(Cloud 66, n.d.) (WebStorm Help, 2022)

User Requirements

The typical user will only need a web browser in order to use the network, that is because in order to view their wallet, make transactions and use the blockchain they will need access to the website accompanying this project (<https://monochain.network>). This means this type of user (a web based user) can use either a typical computer or a mobile device, provided it supports a modern web browser.

However, users who intend to use their computing power as a node for the network will need to be able to run the node software created as part of this project. The final version of this software will be written in V ([V lang](#)), which gets compiled down to C and then machine code, therefore although it could theoretically be run on a desktop computer and a mobile device, that would make the code too unnecessarily complicated as it doesn't really make sense in this scenario and so it will only be available for windows, macOS and Linux

Web based Users

Chrome

OS	Requirements
Mac	<ul style="list-style-type: none">• OS X El Capitan 10.11 or later
Linux	<ul style="list-style-type: none">• 64-bit Ubuntu 18.04+, Debian 10+ 15.2+, or Fedora Linux 32+• An Intel Pentium 4 processor or later SSE3 capable
Windows	<ul style="list-style-type: none">• Windows 7, Windows 8, Windows 10 or later• An Intel Pentium 4 processor or later SSE3 capable
Android	<ul style="list-style-type: none">• Android Marshmallow 6.0 or later

Safari

OS	Requirements
Mac	<ul style="list-style-type: none">• Any Mac running Mac OS X Leopard 10.5.8, Mac OS X Tiger 10.4.11 and Security Update 2009-002• Mac with an Intel processor or a PowerPC G4, or G3 processor and built-in FireWire port• 256MB of RAM• Top Sites and Cover Flow on Mac require a Quartz-Extreme compatible card.

Firefox

OS	Requirements
Windows	<ul style="list-style-type: none"> • Windows 7, 8, 10 or 11 (32 or 64 bit) • Pentium 4 or newer processor that supports SSE2 • 512MB of RAM / 2GB of RAM for the latest version • 200MB of hard drive space
Mac	<ul style="list-style-type: none"> • macOS 10.12 -> 10.15, 11 or 12 • Mac computer with an Intel x86 or compatible processor • 512 MB of RAM • 200 MB hard drive space
Linux	<ul style="list-style-type: none"> • Required: <ul style="list-style-type: none"> • glibc 2.17 or higher • GTK+ 3.14 or higher • libstdc++ 4.8.1 or higher • X.Org 1.0 or higher • Recommended: <ul style="list-style-type: none"> • X.Org 1.7 or higher • DBus 1.0 or higher • GNOME 2.16 or higher • libxtst 1.2.3 or higher • NetworkManager 0.7 or higher • PulseAudio

(Google, 2012) (Apple, 2009) (Firefox, 2022)

Node Users ("miners")

Node users will need a browser as specified in web based users and will need either an ARM cpu or a x86 based cpu, alongside either a modern unix based OS (macOS & Linux) capable of running "exec" files or a modern windows OS capable of running modern "exe" files.

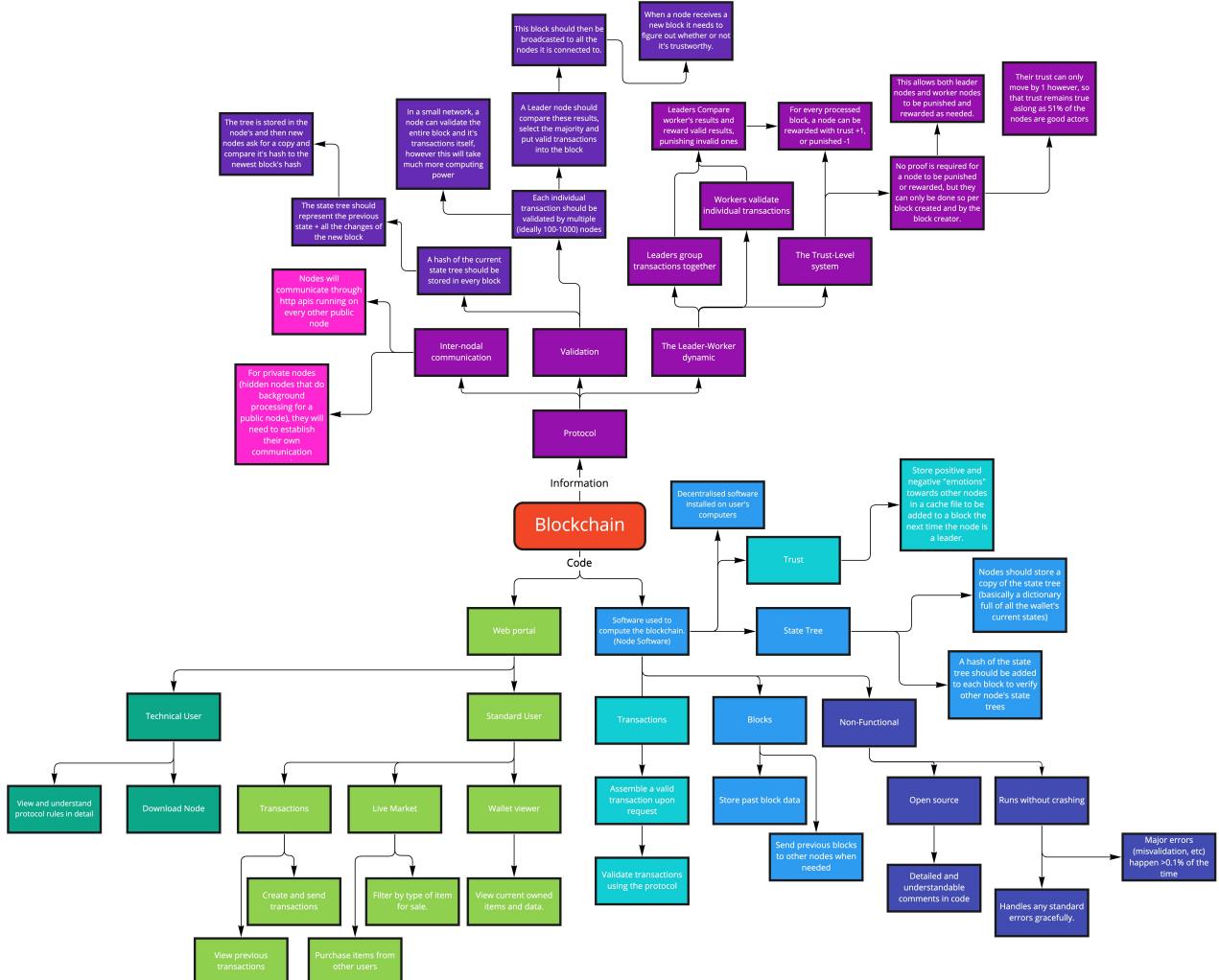
This is because the node software will eventually be compiled to binary packages, and the two types of binary package built by the v compiler are unix executables and windows executables, so the node user must be able to run one of those types of file.

Alternatively they can translate and recompile the node software into any programming language they want, as it will be open source.

2 Design and Development

2.1 Design Frame

Systems Diagram



The diagram above shows the major sectors of this project that I will be creating through out development. The diagram is split into three major sections that then have their own sub-sections to help simplify each part.

The main sections are the following:

1. The Protocol - This is a summary for how the computers on the network will interact, construct data and validate it in order to create what will be known as the MonoChain.
2. The Web-portal - This is the website that will allow users to interact with the MonoChain through a few basic functions which are shown in the diagram.
3. The Node Software - This is what people who want to contribute to the MonoChain through the validating of transactions and data in order to "mine" coins on the network will use on their computers/servers.

Throughout the development stage I will initially focus mainly on the Node Software and the Protocol, and then once they are up to the point at which the network somewhat works I will also start developing the webportal as when the network doesn't work the portal doesn't have much of a use. I have also decided to break down the diagram into those specific subsections in order to roughly align with the [Success Criteria](#).

Usability Features

Usability is an essential aspect to my project as I want it to be usable and understandable to both developers who have a prior understanding of blockchain technology and non-developers who have never used it before.

Effective

I need to ensure that users can do what they are trying to accomplish as easily as possible and part of this will include configuring their settings and generally using the software. The effectiveness of the usability is also relevant when discussing how intuitive the project is to use, which usually comes in the form of questioning how much assistance a user needs from documentation and other people before they figure out how to use part of the project.

Aims

- Ensure that the user can use the majority of the project without having to get much support and thus the software is intuitive.
- Make the software as easy to understand as possible such that the user has high autonomy

Efficiency

The Efficiency part is concentrated on the speed and accuracy to which a user can complete whatever they are trying to do. This equates to making the UI for both the web-portal and the node software as simple and easy to navigate as possible. In my case, part of this will be also based upon how important the information/part of the project the user is trying to get to is, as if it is important and commonly accessed it should take very few clicks to get to but less important information that isn't accessed that much is okay to take more clicks.

Aims

- Ensure users can get from one point on the website to any other within 3-7 clicks, with 3 being for major sections such as the home page and 7 being the more detailed sections such as the documentation for the configuration file for the Node software.
- Allow users to setup a node and have it start running within as small a time period as possible.

Engaging

Making a blockchain and its mining software engaging is not a simple task, however in order to keep the project on the whole interesting to the user the focus will be the web-portal. This is because this is the part of the project a user will visit before deciding whether to actually use the rest of the project and since the node software is designed to be left alone and let to run so it doesn't matter as much how engaging it is.

To keep the web-portal engaging, it will require bright, colourful graphics and design to attract users alongside short slices of information that tell them all they need to be told without overwhelming them with information. However individuals who are already interested in the software and wish to understand the technical side better should also be able to find out as much as they want to know, which will be done through select pages of which contain a much higher quantity of information.

Aims

- Create a simple, clear Homepage for the web-portal.
- Stick to a simple, repetitive art style so the user recognises all parts of the web-portal as being a part of the project.
- Create more detailed sections/pages of information for users who wish to understand the technical detail better.

Error Tolerant

No part of the project should crash completely, as this can be terrible for user's opinions of the project.

If the web-portal was to crash it would likely result either in the user coming back later or just not using that section of the website as due to the nature of being a client sided website it is very difficult to crash the entire site.

However, if the node software was to crash it could have much worse effects, as it is designed to be left to run and users should only be expected to check on it once every few days, hence a crash could result in days of missed processing which is not only bad for the network but could also lead to users giving up on the project completely.

Hence to solve this problem I will be writing the code to catch and prevent as many run-time errors as possible, and to handle any errors which do occur without crashing which will be made easier through the use of a compiled language that runs most of its error checking during compilation.

Aims

- The Node should never crash completely, parts of it may stop working until the user comes to check on it but it shouldn't completely stop processing.
- The Webportal should have no code-breaking bugs, with the only errors that make it through to production being styling/graphics based bugs.

Easy To Learn

The solution should be easy to use and not be over complicated. To do this, I will create a simple command line based interface for the node program, which contains only the options and input required without confusing the user. Then for the web-portal, the locations of buttons and the transportation to different parts of the website should remain constant through updates and are categorised so that users only have to worry about the sections they actually want to visit and aren't overwhelmed with choice.

Aims

- Minimise the amount of user input required to operate the Node software.
- Categorise web-portal transportation methods to simple, easy to understand groupings such that users are not overwhelmed with choice but instead can make a few simple choices to get to where they need to go.

Pseudocode for the Node project

The main function

The main function of which operates the node will be very simple, due the general design of the software being split up into lots of different files and functions to be reused in different parts of itself and to abstract as much of the program as possible.

```
MODULE main
IMPORT server
IMPORT configuration

PROCEDURE main():
    OUTPUT "Launching MonoChain Mining Software"
    config = configuration.get_config()
    server.start(config)
END PROCEDURE
```

The server start function

This is the function for which the server initialises a connection to the network and sets up all the routes for other nodes and programs that use the network to request to.

```

MODULE server

init_ref = "https://nano.monochain.network"

FUNCTION start(config):
    // start server on a new thread
    api = RUN http_server ON PORT config.port

    // wait to make sure server is up before proceeding
    WAIT 2 SECONDS

    // ping the node's entry point to connect it to the network
    server.start_handshake(init_ref, config)

    // bring server process back to main thread
    api.wait()

END FUNCTION

```

A generic route on the server

This shows the basic layout of code for a route within the api server which is started in the main program.

```

module server

HTTP ROUTE ("/", GET) index(request):
    // the request object contains all data from a traditional http request.

    // some data would then be calculated
    DATA = "This is the api route for a node running on the Monochain network."

    // this data is then sent back to whoever requested the route.
    RETURN app.text(DATA)

END ROUTE

```

Pseudocode for the Webportal

The webportal will be built using react, next.js and typescript, the way in which these technologies can be used (and will be used in this project) allows for each web page to be placed in a separate file. Therefore each file would be fairly similar in terms of structure yet very different in terms of content, so the below pseudocode contains some example components with some example css parsing to represent the structure of a webpage in this format.

```
FUNCTION Page() {  
    // any code that is put here is just typescript code  
    // it will be executed prior to the webpage being sent  
    example = "hello world"  
  
    RETURN (  
        <Generic Component  
            width = "full"  
            height = "full"  
            textAlign = "center"  
            padding-x = 10  
            flexDirection = "column"  
        >  
        <Text>  
            {example}  
        </Text>  
    </Generic Component>  
);  
};  
  
EXPORT Page;
```

2.2.1 Cycle 1 - The Framework

Design

Objectives

In this first cycle I am to build out the frameworks for my project. This will include: a web portal for showcasing the project, letting users create wallets and send/receive data to and from their wallets, explaining the rules of the blockchain and the interfaces that can be used to interact with it; a node program that will act as a program that users can download and run to host the blockchain on their device and contribute to the project; and a utils package that holds code and types needed by both the web-portal and the node.

Because of the fact that this relies on three separate packages and it may need additional packages in the future the ideal solution for this code base is a monorepo, therefore the first thing I will need to do is set up a monorepo that allows the packages to be built and used separately, to do this I will be using yarn workspaces and lerna.

- Setup a monorepo for hosting the packages in a single repository.
- Build a very basic initial version of the web-portal for testing deployment.
- Deploy the initial version of the web-portal.
- Build a very basic version of the node program that says "hello world" in the console
- Figure out the best way to let users download and run the node program.
- Attach a way to download the node program to the web-portal.
- Successfully download and run the node program from the webportal.

Usability Features

- Easy to read text - the web portal will be built using a white background and black text, this contrasts the two greatly and makes the text a lot easier to read.
- Navigation - when using either the node software or the web portal the controls should be easy to understand and navigate, in the web portal's case it's important everything a user could want should be accessible within as few clicks as possible.

Key Variables

Variable Name	Use
workspaces (/package.json)	Tells lerna and yarn workspaces where the packages directory is and therefore where to build programs from.
references (/tsconfig.json)	Tells typescript where references across package should occur - allows web portal and node to share types and interfaces through the utils package.
href (within web-portal download page)	Tells the browser where the node package download is hosted, and therefore what file to download

Pseudocode

Demo Node program:

```
OUTPUT "hello world!" //outputs a string to the console to confirm it's running
```

Download Feature on webportal:

```
<React component> // This refers to the page of which the download button exists
  <Link href="https://path/to/node/download" download>
    // The Link paired with the download flag above ^ tells the browser
    // to download the file referenced in the 'href' field to the users computer.

    <Text>
      Download Node
    <End Text>
    // The text component just shows some text within the link box to the user

  <End Link>
<End React component>
```

Development

Most of the development for this cycle was just setup, to get everything I need for this project up and running and building out the structure of the codebase in order to make the actual programming as smooth as possible.

Outcome

Web Portal

In the pages directory I have created four primary web pages:

- [download.tsx](#)
- [index.tsx](#)
- [info.tsx](#)
- [wallet.tsx](#)

Node Software

Challenges

The main challenge I faced in this cycle of development was deciding how to make the node software available for download, originally I attempted to setup a file server and upload files to that in order to download the software easily, however I soon realised that this would be a lot of setup to just host one file.

Luckily I soon realised that you can download files very easily from websites using the "download" flag in chakra-ui (the component package I'm using - it's very similar for html/css) and because I had setup the repositories in a monorepo it was possible to build and zip the node software from the node's package, then copy it into the public folder within the website's package and have the download link from the website just point to the zipped software within it's own public folder.

This then means that upon building the node software, the typescript in which it is written is compiled to javascript within the "dist" folder (dist meaning distribution), this dist folder is then zipped and finally the zipped folder is copied to the web portal's public folder.

This is done using yarn commands which effectively just let you string together a series of bash commands that are local to your project, the commands themselves can be found within the "package.json" file within the node package. (The current package.json is shown below)

```
{  
  "name": "@namespace/node",  
  "version": "1.0.0",  
  "main": "dist/index.js",  
  "types": "dist/index.d.ts",  
  "scripts": {  
    "start": "node dist/code/index.js",  
    "clean": "rimraf dist/code && rimraf tsconfig.tsbuildinfo && mkdir dist/code && echo \"cle  
    \"clean-zip\": \"rimraf node.zip && echo \"cleaned node.zip\"",  
    "prepack": "yarn build",  
    "build": "yarn clean && yarn clean-zip && yarn compile && yarn zip && echo \"zipped node\\\"  
    \"compile\": \"tsc && cp './package.json' ./dist/code/ && echo \"compiled\\\"",  
    "zip": "zip ../webportal/public/node.zip -9r ./dist/ && echo \"zipped\\\"",  
    "lint": "eslint './src/**/*.{ts,tsx}' --max-warnings=0"  
  }  
}
```

By following this code we can see that once "yarn build" is run it runs:

1. "yarn clean" - which removes the last code distribution and typescript build info from the file system, then recreate the distribution code file structure before echoing that it has finished cleaning the distribution files.
2. "yarn clean-zip" - which removes the previous zip file and then echoes that the command has successfully run
3. "yarn compile" - which compiles the typescript to javascript, copies the current package.json to the distribution folder and echoes to let us know it's successfully completed.
4. "yarn zip" - which zips the newly compiled distribution folder to the webportal's public folder and echoes to confirm it's finished.

Testing

Tests

Test	Instructions	What I expect	What happens
1	Run the webportal on localhost.	Webportal launches and text is displayed on the screen	As
2	Deploy the web-portal by git pushing to Github and Vercel should automatically deploy.	Website launches successfully on frontend server and when navigated to displays the same text as on localhost.	As
3	Run node program.	An output of "hello world"	As
4	Download the node program by clicking on the download button on the web-portal	The Node program should be downloaded to the test computer	As

Evidence

Test 1 - Web portal running on localhost:

MonoChain

A blockchain based around the JSON format, allowing you to store and own any data that can be stored in a javascript object.

This means that not only can MonoChain support small transactional data, but can also be used for large data storage.

Meaning that not only does the MonoChain support cryptocurrencies and traditional nfts (links to images stored on other external databases and servers), but also ownership certificates from other sites if they choose to support MonoChain.

This would allow for other, external sites, that have nothing to do with the MonoChain, to build their entire storage directories on the MonoChain, provided they are willing to run their own nodes on the network to process their transactions, or pay fees to willing nodes to run these stores of data instead.

Web portal running as expected on localhost:3000 when "yarn next" is ran in /packages/webportal

```
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
event - compiled client and server successfully in 328 ms (622 modules)
```

Console output for the webportal.

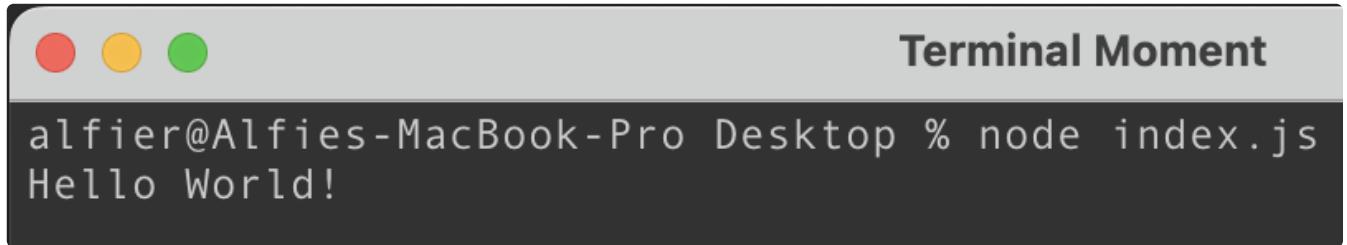
Test 2 - Deploy and Run the webportal on an external server:

The screenshot shows a Vercel deployment dashboard for the MonoChain webportal. On the left, there's a preview window showing the MonoChain homepage with its mission statement and storage capabilities. On the right, detailed deployment information is provided:

- DEPLOYMENT:** monochain-webportal-3szgg6oq2-alfieran.vercel.app
- DOMAINS:** www.monochain.network +3
- STATE:** CREATED
- Ready** (green status indicator)
- CREATED** (time since deployment)
- BRANCH:** master
- added the other two pages** (description of recent changes)

Vercel successfully running the webportal on <https://monochain.network>

Test 3 - Running the node program:

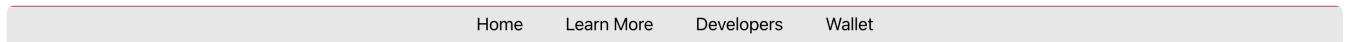


A screenshot of a terminal window on a Mac OS X system. The window title is "Terminal Moment". The terminal content shows the command "node index.js" being run, followed by the output "Hello World!". The window has the standard OS X title bar with red, yellow, and green buttons.

```
alfier@Alfies-MacBook-Pro Desktop % node index.js
Hello World!
```

This shows the node software running within my terminal.

Test 4 - Downloading the node program



Downloads

Node (Mining software)

Although it is recommended to either write your own miner or use a third party node program written in a lower level language, I have included a zipped node program that runs using javascript.

That means that to use this program you must first download node.js and unzip the file. It can then be started by navigating to where-ever you downloaded the node program to, opening the "dist" folder that should be created after it was unzipped and typing "node ./code/index.js" into your terminal.

[Download Node Software](#)

The top image shows the page that the node software can be downloaded from, with the lower image showing the zip file being downloaded.

2.2.2 Cycle 2 - Changing Node Language to V Lang

Design

Objectives

If I want to let users contribute using the node software and continue down the javascript & node.js route they will have to download the software, download node, then all of the libraries I used for the project, which is far from ideal.

Therefore I'm going to be switching to a different programming language and a different distribution method. The conditions I need this programming language to meet are the following:

- Can be compiled with libraries to limit the amount of setup end users have to do.
- Can be compiled to arm and x86 on macOS, Linux and Windows.
- Ideally has type safety
- Runs Fast
- Has networking capabilities
- Has modern features

The best language I could find for those specifications was V ([aka Vlang](#)) which is a type based language based upon C.

There are both upsides and downsides to using Vlang instead of typescript, the largest upsides are how much faster than typescript it is and that it can be compiled to binary so that end users won't have to worry about downloading additional packages/files. The downsides come more in the form of general development, where some functionality is likely to contain more bugs and there will likely be a lot less documentation because Vlang is such a new programming language.

Rewriting what I've done so far.

Now that I've decided to switch from typescript to Vlang I will first need to rewrite everything I've done in this new language. However since this isn't actually that much, I will also be setting up the api server in this cycle which will allow you to go to the ip address that the node is running on and using a http get request (what your browser sends when you open a webpage) to receive the message "Hello world".

Summary of the objectives for this cycle

- Create a V program that runs.
- Setup the api using the default Vlang library: "Vweb"
- Allow the index to be requested using http and respond with "Hello World"
- Compile the V program to a macOS executable
- Configure the build files to compile the V program and put it in the public folder within the web server project so it can be downloaded.

Development

Setting up Vlang

Setting up the V compiler is actually pretty basic, all that is needed is to clone the git repo (<https://github.com/vlang/v>), run the make file and then follow the instructions given. I then also setup the symlink, which is what allows the compiler to run just by typing "v" in the terminal rather than having to run the path to the executable.

This can be verified to have been setup properly using "v --version" in the terminal, see tests.

Writing the Vlang program

In order to keep the code clean and easy to understand, I'm going to split the different parts of the node software into different files and then different modules to group those files together. In this dev cycle I will be setting up the first two modules, main and server.

(i) I accidentally forgot to add this cycle's V files to the git repository until a later cycle as I started using vsCode as the program editor for the node instead of WebStorm, which doesn't automatically add new program files to git like I'm used to.

This means you can see a lot of the same code in the git commit where I realised this was an issue ([here](#)) but the code mentioned in this page had been modified to include parts of cycle 3 and 4 by then.

The Main module

This will be the part of the node software which handles and operates all of the other modules, all the major data in this program should flow through the main module. Initially it will need to:

- Import the server module
- output to the console to let the user know the node is running
- launch the server

This will look something like:

```
// Pseudocode
module main
import server

procedure main():
    output "A Welcome Message"
    port = (an integer value between 0 and 25565)
    server.start(port)
end procedure
```

After converting this Pseudocode to V code we get the following:

```
// Vlang Code - packages/node/src/main.v

module main // declares which module this part of the code is
import server // imports the server module referenced above

fn main() {
    // output to the terminal to let the user know it's running,
    println('***** MonoChain Mining Software *****')
    port := 8001

    // then start the webserver
    server.start(port)
}
```

The Server module

Now that I have built out a basic main file, it needs the server module to allow the server.start(port) function to actually work, this is the point of that module.

- Have a start function which takes a port number as an input and launches a web server on that port.
- Setup a "Hello World" function at the index of the server ("/")

```
// Pseudocode
module server    // declare which part of the project this is
import webServer // import a http server module

function start(port):    // starts running the webserver
    webServer.run()

function index_fn():    // returns 'Hello World!' to the domain it runs on
    return "Hello World!"

webServer.page("/", index_fn)    // assign the index_fn to "/" on the server.
```

This looks slightly differently in V due to how the default api server works but it does the same thing.

```
// Vlang Code - packages/node/src/server/main.v

module server // declare which part of the project this is
import vweb // import the http api server library.

struct App { // generate key constants that represent the server
    vweb.Context
}

pub fn start(port int) { // this launches the server
    vweb.run(&App{}, port)
}

pub fn (mut app App) index() vweb.Result {
    return app.text('Hello World!')
} // this is how a webpage is represented in V
```

The New Build Configuration

Because the previous version of this code was setup in a different language, it means that all the build configuration files are now incorrect, correcting this is fairly easy, as yarn - the configuration and package manager I use - allows support for any terminal commands to be executed through its scripts function.

This means I just have to figure out what terminal commands will need to be run differently to the typescript version and adjust them accordingly. Then finish off the config files by removing any unnecessary junk left over by the old config.

Compiling

I've changed quite a lot in the config file, but most of it is fairly basic so I've chosen to only highlight the biggest change, which is how the program is compiled.

```
// Old Configuration
"compile": "tsc && cp \"./package.json\" ./dist/code/ && echo \"compiled\"",
```

```
// New Configuration
"compile": "v ./src/ && mv ./src/src ./dist/code && echo \"compiled\"",
```

As you can see there isn't a massive difference between the two, with the only actual difference being that the program is compiled using "v ./src/" rather than "tsc", and that the resulting file is now moved using "mv ./src/src ./dist/code" rather than being compiled in that location and partially copied using "cp \"./package.json\" ./dist/code/" although they both result in a compiled version of the program being in "./dist/code".

! It's important to remember here that unlike typescript, which just compiles to javascript, V compiles to binary, this means that this compiled version of the project can only be run on an arm based macOS device, or whatever else it is compiled on.

Users who don't use a device of that exact type (most of the expected user pool), can however clone the git repository and compile the program on their own device and it should work without any issues!

Challenges

The main challenges for this cycle have been learning the syntax and flow of a completely new programming language, since I have never actually used V for a project before this one I spent a lot of time in [the documentation](#) and searching the internet for solutions to obscure and unusual compiling issues (which I didn't mention in the testing as it turned out to be an issue with how I downloaded the compiler and not the code I wrote.).

This wasn't a huge issue but it did certainly add some friction to the development process, and I'm hopeful that by the end of this project I should've proficiently learnt how to use Vlang to the extent at which I'd say I can code well in it.

Setup and Configuration Testing

Test	Instructions	Expected	Actual Result	Passed/Failed
1	run "v --version" in the terminal	Output of "V" then some version number and a hash of that update.	"V 0.3.0 426421b"	Passed
2	run "yarn start" in the node package's source.	The node software to be compiled and ran.	As expected	Passed
3	run "yarn build" in the node package's source	The node software to be compiled, zipped and put into the webportal's public folder.	As expected	Passed

Test 1 - Checking that the V compiler is installed properly

```
alfier@Alfies-MacBook-Pro ~ % v --version
V 0.3.0 426421b
```

Testing to ensure the V compiler is installed on my laptop properly.

Test 2 - Seeing if the re-written software runs as expected

```
alfier@Alfies-MacBook-Pro node % yarn start
***** MonoChain Mining Software *****
[Vweb] Running app on http://localhost:8001/
```

Welcome message displayed alongside an output to declare the server has started

Test 3 - Building the node software

```
alfier@Alfies-MacBook-Pro node % yarn build
cleaned dist/code
cleaned node.zip
compiled for native OS of this system
adding: dist/ (stored 0%)
adding: dist/ExitCodes.txt (deflated 46%)
adding: dist/.DS_Store (deflated 94%)
adding: dist/code/ (stored 0%)
adding: dist/code/node-native (deflated 59%)
adding: dist/README.txt (deflated 48%)
zipped
zipped node
```

Building the software for the native OS of the system it is ran on - in this case mac.

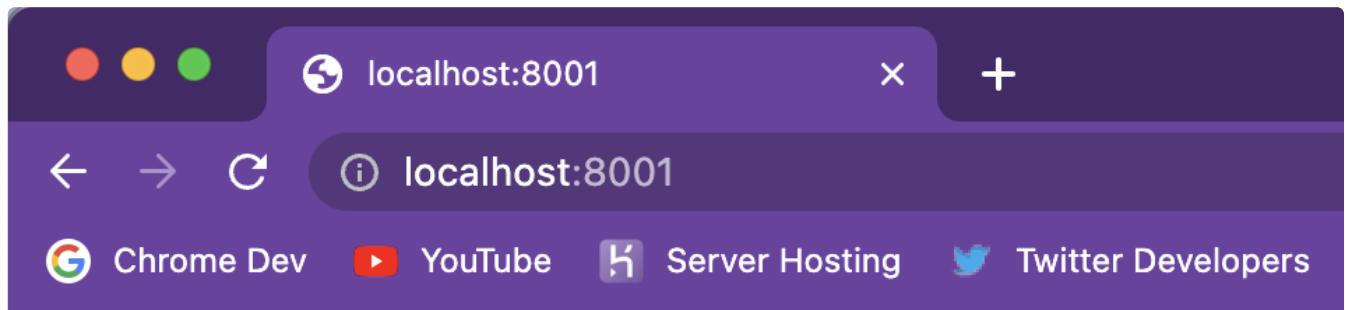
Program Testing

Test	Instructions	Expected	Actual Result	Pass/Fail
4	start the program using "yarn start"	Outputs a welcome message	"***** MonoChain Mining Software *****"	Pass
5	Web api should launch on port 8001	"Hello World" on "http://localhost:8001"	As expected	Pass

Evidence for Program Tests

Test 4 - See Evidence for Test 2

Test 5 - Web api launched at "http://localhost:8001"



Download Testing

Test	Instructions	Expected	Actual Result	Pass/Fail
6	Navigate to "https://monochain.net work/download"	See updated download instructions	As expected	Pass
7	Download and unzip the node software	Should download and unzip to an executable file	As expected	Pass
8	Run the node software on an arm based macOS device	Should run as tested in program tests	As expected	Pass

Evidence for Download testing

Test 6 - Updated download instructions

Downloads

Node (Mining software)

This is an example node software written in Vlang, it is recommended that you clone the git repository and compile it yourself so that you can edit any pieces of code you wish to and to ensure that it runs properly on your system as V compiles to an executable that will be different for different systems. The below zip file contains an arm64 macOS executable called "node-native" (native because that's the native OS for the laptop I do the dev for this project on) and an x86 windows executable called "node-windows". If you don't use either of these systems then you must clone and compile the git repo yourself.

[Download Node Software](#)[Clone the repo](#)

2.2.3 Cycle 3 - Configuration Handler

Design

Objectives

For this cycle the main objective is to turn the node software into something more aligned with what a user would actually be able to use, as although it will mainly be computer oriented people using this part of the project having it easy to understand is still a big benefit.

- Implement a configuration handler and setup tool.
- Allow users to generate new configuration files as needed
- Save and load configuration files whenever they are needed

Usability Features

Configuration handler

For the configuration handler it will originally start off very simply and will probably be somewhat a pain to use, but the hope is that by laying out a ground framework now that contains the console interface that will be used to interact with the configuration handler to be in plain, easy to understand English. This will give users the best chance at being able to run through the configuration setup with as little friction as possible.

- Plain, easy to read English text based interface.
- Allows user to respond in a variety of ways ("yes", "y", "confirm", "approve", etc).

Key Variables

Variable Name	Use
user_config	to store the user's current configuration settings for various parts of the system. (which port to use for the server, etc)
config_path	the default place for user configurations to be stored by the software for loading/saving configs.

Pseudocode

The handler will be built with three main functions:

- `get_config` meant for collecting the configuration for use by other parts of the program.
- `create_configuration` meant for generating a new configuration for the user if they don't already have one.
- `save_config` meant for saving the current version of the configuration to the file.

Getting the configuration.

`get_config` will allow any part of the software to load the newest version of the configuration, ideally this will be from memory but will likely just be from the file to start with. It will look something like the following:

```
// Psuedocode - get configuration function

CONST config_path = "./monochain/node.config" // the key variable described earlier.

FUNCTION get_config():
    // load the config from the defualt file location and decode with json.
    user_config = file.read(config_path, "json")

    // check if the config loaded properly
    IF user_config DOES NOT EXIST:
        // if it hasn't, let the user know and start generating a new one.
        OUTPUT "No config detected, or error occoured."
        user_config = new_config()
    END IF

    // if generating a new config failed aswell, then exit with code 100
    IF !user_config.loaded
        OUTPUT "Failed to create a new configuration file.\nExiting..."
        exit(100)
    ENDIF

    // if we make it to this part of the code then the config is ok
    OUTPUT "\nConfig Loaded..."
    return user_config
END FUNCTION
```

Generating a new config

`create_configuration` is a pretty simple function that collects data from other functions - such as the `ask_for_port` one also in the pseudocode below - and formats that data into a configuration object before saving it and returning the config to wherever it has been called from.

```

// Psuedocode - create a new configuration

FUNCTION create_configuration():
    // create the config object
    config = {
        // let's the program know the config loaded successfully
        loaded: true
        // generate the port to use in the function below
        port: ask_for_port()
    }

    // save the config object to a file
    save_config(config)
    return config
END FUNCTION

FUNCTION ask_for_port():
    // ask the user for the port
    OUTPUT "What port would you like to run your node on?"
    port = INPUT

    // check the port is valid
    IF port > 65535 || port < 1:
        // if it's not valid, tell the user and ask for the port again
        OUTPUT "That port does not exist! You might want to enter a number between 1 and 65535"
        return ask_for_port()
    END IF

    // return the port
    return port
END FUNCTION

```

Saving the configuration

This function simply saves the configuration to a file so that it can be loaded/updated by other parts of the program as needed, and will keep its contents if the node/computer running the program is shut down or stopped.

The code will look something like the following:

```

// Psuedocode - save the configuration

FUNCTION save_config(config)
    // convert the data into something safe to be saved into a file
    data = json.encode(config)

    // write the encoded data to the disk at the path ./node.config"
    file.write("./node.config", data)
END FUNCTION

```

Development

Most of the development for this cycle went pretty smoothly, which was very nice!

This was mainly due to the module being relatively simple, so most of the development was simply converting the pseudocode and concept specified above into V code.

In order to do this I first created the "configuration" module, which is done simply by creating a new folder with the name of the module and adding the line `module configuration` to the top of any files that are part of that module.

Once the module was setup I created three files for the code to help separate it out and make it easier for anyone looking at the code base to see what was going on, the files were:

- `configHandler.v` - This is the main file of the module and contains the `get_config()` function mentioned earlier, this then calls on a function from the `configFileHandler.v` file below and if there is a config file setup, loads it, and if not then calls on the `configCreator.v` file to create a new configuration.
- `configCreator.v` - This just houses some basic user interfacing functions that ask for various pieces of data required to generate a new configuration and then turns that into a new configuration object and saves it using the `configFileHandler.v` file below.
- `configFileHandler.v` - This is a basic file used to load, save, and check the existence of configuration files, any file read through this is type safe due to the language being used (VLang) reading json data using the expected object type and rejecting the file if it is incorrectly formatted. This means that all the data loaded to and from files using this part of the module are the correct object types and **shouldn't** cause any formatting crashes in other parts of the program.

Outcome

Because there is actually quite a lot of code in this module, I will just include some major functions such as the ones mentioned in the pseudocode.

Getting the config file

```
// Vlang Code - "./packages/node/src/configuration/configHandler.v"

pub fn get_config() UserConfig {
    // the function used below can be found in the configFileHandler file
    // it is also in the final code subsection further down this page
    mut user_config := load_config()

    if !user_config.loaded {
        println("No config detected, or error occurred.")
        // the following function asks the user if they want to generate
        // a new config and if so takes them to the "create_configuration" function.
        user_config = new_config(0)
    }

    if !user_config.loaded {
        // if something went wrong, quit
        eprintln("Failed to create a new configuration file, unexpected value was $use
        exit(100)
    }

    // nothing went wrong! So load the next part of the program.
    println("\nConfig Loaded, launching API server...")
    return user_config
}
```

This version of the code can be found on [Github Commit bc45df98e7](#).

Generating a new config

```

// Vlang Code - "./packages/node/src/configuration/configCreator.v"

pub fn create_configuration() UserConfig {
    // build the config object using subfunctions
    config := UserConfig{
        loaded: true
        port: ask_for_port(0)
    }

    // save the config
    save_config(config, 0)
    return config
}

fn ask_for_port(recursion_depth int) int {
    // this just asks the user what port they'd like to run their node on
    // then sends that data back to the config creator.
    mut port := (read_line("What port would you like to run your node on (default: 8001)?"
        eprintln("Input failed, please try again")
        utils.recursion_check(recursion_depth, 2)
        return ask_for_port(recursion_depth + 1)
    }).int()

    if port > 65535 || port < 1 {
        eprintln("That port does not exist! You might want to enter a number between :"
        utils.recursion_check(recursion_depth, 3)
        return ask_for_port(recursion_depth + 1)
    }

    return port
}

```

This version of the code can be found on [Github Commit bc45df98e7](#).

Saving the configuration files

This includes the `save_config` function as expected, but also the `load_config` function which was originally planned to just be built into the `get_config` function from earlier.

```

// Vlang Code - "./packages/node/src/configuration/configFileHandler.v"

pub fn save_config(config UserConfig, recursion_depth int) bool {
    // setup a failure tracker
    mut failed := false

    // encode the data
    data := json.encode(config)

    // this shouldn't be here but it is in the commit referenced so I left it in.
    // it doesn't error, it'll just never get run.
    if failed {return false}

    // save the file
    os.write_file("./node.config", data) or {
        // if this gets run then something went wrong.
        eprintln('Failed to save file, trying again.')
        if recursion_depth >= 5 {
            // if it's already failed multiple times then quit
            eprintln("Failed to save file too many times, continuing with program")
            failed = true
        } else {
            // if no recursion issues, then just try again
            failed = !save_config(config, recursion_depth + 1)
        }
    }

    if !failed {
        println("Successfully saved configuration file.")
    }
    return !failed
}

pub fn load_config() UserConfig {
    if os.exists("./node.config") {
        // config file already exists, make sure we can open and decode it

        config_raw := os.read_file("node.config") or {
            // something went wrong opening the file, return null
            eprintln('Failed to open config file, error $err')
            return UserConfig{ loaded: false }
        }

        user_config := json.decode(UserConfig, config_raw) or {
            // something went wrong decoding the file, return null
            eprintln('Failed to decode json, error: $err')
            return UserConfig{ loaded: false }
        }

        return user_config
    }
}

```

```
        return UserConfig{ loaded: false }  
    }
```

This version of the code can be found on [Github Commit bc45df98e7](#).

Testing

Tests

Test	Instructions	What I expect	What happens
1	Create a new configuration file using "create_configuration()"	Console logs to confirm the operation is commencing and a configuration file to be generated.	As
2	Load the new config file using "load_config()"	Config file should be loaded and returned	As
3	Load the new config file using "get_config()"	Config file should be loaded and returned	As

Evidence

```
Running test: create configuration  
What port would you like to run your node on (default: 8001)?  
$:8000  
Successfully saved configuration file.  
test: create configuration passed
```

Test 1 Passed

```
Running test: load configuration from file  
test: load configuration from file passed
```

Test 2 Passed

```
Running test: load configuration using get config  
Config Loaded, launching API server...  
test: load configuration using get config passed
```

Test 3 Passed

2.2.4 Cycle 4 - Compiling for Windows

Design

Objectives

The main objective in this cycle is to allow the software to be compiled and ran on multiple operating systems, apart from just macOS. The key OS here is windows, but I will also be trying to get compilation working for Linux.

- Compile for Windows
- Test the current software on Windows and ensure it works properly
- Compile for Linux
- Test the software on Linux and ensure it works properly

Usability Features

For the compilation, I would like to make it so that the user just selects which OS they are using on the website and download only the correct version, however that would take up too much time at this point of the project and can be added later on, therefore for now the user will just download all available versions of the program and run the one that their system can handle.

- Make it clear which version of the program the user should run based on their system using a brief summary.
- Not crash the computer if the user runs the wrong version.

Development

This was a lot more tricky than I had first anticipated, since once I started working on this cycle, the configuration was all running fine and had passed all the tests I ran it through on my laptop, so to start compiling for multiple OS all I thought I had to do was add some flags to the compiler inputs but sadly it was not quite that simple.

The V docs suggested it was as simple as I hoped (available [here](#)), but it turns out that because the laptop I have been using to create this project runs using an ARM64 chip as opposed to a traditional x86 one, there were some more issues.

What was supposed to happen:

All that I was supposed to have to do was add two new functions to my builder settings, one of which specified the OS to be Linux as follows:

```
v -os linux ./src/
```

And one for Windows:

```
v -os windows ./src/
```

Then the code should've been compiled, packaged with the other versions and uploaded to the webportal ready for download.

What actually happened:

Windows

Running the windows compilation appeared to work fine at first, however it did end up having some weirder issues regarding text inputs not being collected properly.

I managed to fix this fairly easily by forcing the results of the inputted data to be a string using `string()` and that seemed to be the end of that error, although because an arm64 macOS and x86 windows is so fundamentally different I expect to see more errors of this kind later on in the project or during [user testing](#) when a variety of systems are tested using the same program.

```
● alfier@Alfies-MacBook-Pro node % v -os windows ./src/
Cross compiling for Windows...
'/Users/alfier/WebstormProjects/A-Level-Project/packages/node/src/src.exe' has been successfully compiled
Compiling for windows
```

Linux

Linux however, did not want to work at all.

```
● alfier@Alfies-MacBook-Pro node % v -os linux ./src/
Cross compilation for Linux failed (first step, cc). Make sure you have clang installed.
builder error: /tmp/v_501/src.13014226172111377526.tmp.c:1597:2: error: VERROR_MESSAGE Header file <sys/syscall.h>, needed for module `crypto.rand` was not found. Please install the corresponding development headers.
#error VERROR_MESSAGE Header file <sys/syscall.h>, needed for module `crypto.rand` was not found. Please install the corresponding development headers.
^
/tmp/v_501/src.13014226172111377526.tmp.c:1705:2: error: VERROR_MESSAGE Header file <sys/un.h>, needed for module `net` was not found. Please install the corresponding development headers.
#error VERROR_MESSAGE Header file <sys/un.h>, needed for module `net` was not found. Please install the corresponding development headers.
^
/tmp/v_501/src.13014226172111377526.tmp.c:1759:2: error: VERROR_MESSAGE Header file <arpa/inet.h>, needed for module `net` was not found. Please install the corresponding development headers.
#error VERROR_MESSAGE Header file <arpa/inet.h>, needed for module `net` was not found. Please install the corresponding development headers.
^
/tmp/v_501/src.13014226172111377526.tmp.c:26325:20: error: use of undeclared identifier 'SYS_getrandom'
    int _t1 = syscall(SYS_getrandom, buffer, bytes_needed, 0);
4 errors generated.
```

Errors following the Linux compilation attempts.

(The above screenshot is actually from later on in the project which is why some cryptography modules are installed, but the error is very similar to the ones I was seeing during this cycle)

A lot of the errors I received in the approximate week I spent trying to debug and fix these compilation errors were claimed to be dependency and clang issues, however I had ensured I had all the correct dependencies installed so either they weren't being loaded correctly, or the CPU architecture of my laptop seemed to not be playing very nice with the Linux compiler system built into Vlang. Either of those are valid reasons since according to similar issues I could find on the internet, a lot of the people facing this issue have a laptop similar to mine.

I then tried compiling and running the software directly on a Linux machine I had and it worked perfectly, so since Linux users tend to also be a lot more used to git and terminal usage, they should be able to just follow the documentation in the project's readme and compile it themselves, hence after this point I decided just to compile for macOS and windows and let other users compile the project themselves.

This is something I would like to change in the future if I have the time, but realistically it will probably remain like this.

Testing

Tests

Test	Instructions	What I expect	What actually happens	Pass/Fail
1	Compile for Windows.	Node program to compile and execute normally.	Node program compiled and ran but with weird input bugs.	Fail
2	Compile for Linux.	Node program to compile and execute normally.	Node program did not compile.	Fail
3	Recompile for Windows with input changes.	Node program to compile and execute normally.	Node program compiled and ran seemingly normally.	Pass
4	Run the Windows version on a non-windows system.	Computer running incorrect program should report some kind of error and close the program.	Using a MacOS device it is as expected.	Pass

Evidence

Sadly I cannot provide evidence for **test 1** as I forgot to collect evidence of the test before fixing the errors resulted from and could not recreate the error without rewriting a large chunk of the configuration handler.

Evidence for **test 2 & 3** can be found above in the screenshots within the area "[What actually happened](#)".

Test 4

```
☒ alfier@Alfies-MacBook-Pro node % ./dist/code/node-windows.exe  
zsh: exec format error: ./dist/code/node-windows.exe
```

Terminal reports a format error but does not crash the window.

2.2.5 Cycle 5 - Signatures and Key Generation

Design

Objectives

The next module that needs implementing and introducing to the code base is the cryptography side, this is the part of the code that allows a node to sign things and to validate other wallet's transactions and signatures.

This will be done through either the RSA or DSA algorithm, as these are the most common and well-known signature algorithms and I can be sure that they work without having to design and test my own signature algorithm.

- Choose between the RSA and DSA algorithm
- Implement the algorithm key generation using either a pre-made library or a custom built solution
- Generate key pairs for the chosen algorithm
- Build a verification function to ensure the key pair was generated properly by signing some random data and checking it.

Usability Features

- The ability to generate keys as needed if the user doesn't have any to reduce friction.
- Once key pair has been either inputted or generated the user shouldn't have to worry about them during the Node's run time.

Key Variables

Variable Name	Use
private_key	This is a private key that should not be shared and is used to sign data.
public_key	This is a publicly available key derived from the private key that can be used to validate that a specific private key produced a signature. Can also be used as an identifier for a wallet.

Choosing a Signature Algorithm

Before this part of the program can be designed and programmed, it is first required to choose a signature algorithm for generating signatures and validating transactions and therefore blocks.

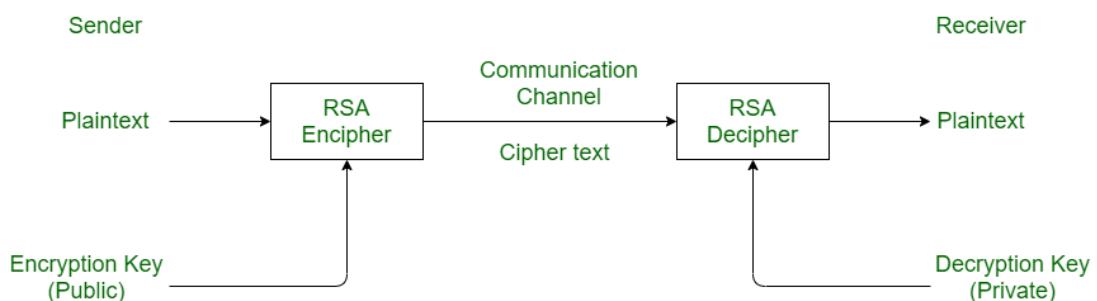
The main two ways to do this are using either the RSA or DSA algorithms, both of which are considered

sate and secure algorithms as of 2022, and can both be used for use case of which I will be using them for in this project.

Summary:

- Primarily used for secure data transmission
- Developed in 1977 by **Ron Rivest, Adi Shamir and Leonard Adleman**.
- Uses the factorisation of product of two large primes for its mathematical security.
- Faster Encryption than DSA
- Slower Decryption than DSA

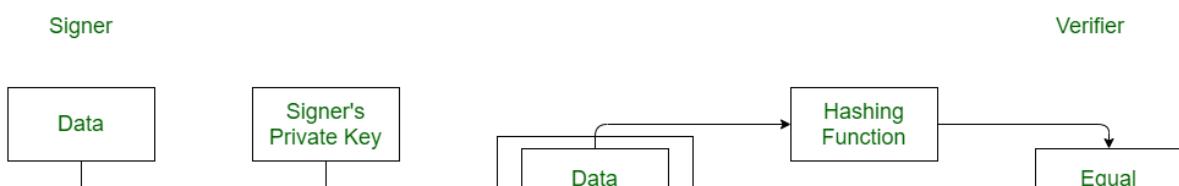
A basic diagram of how RSA works. ([GeeksforGeeks, 2020](#))

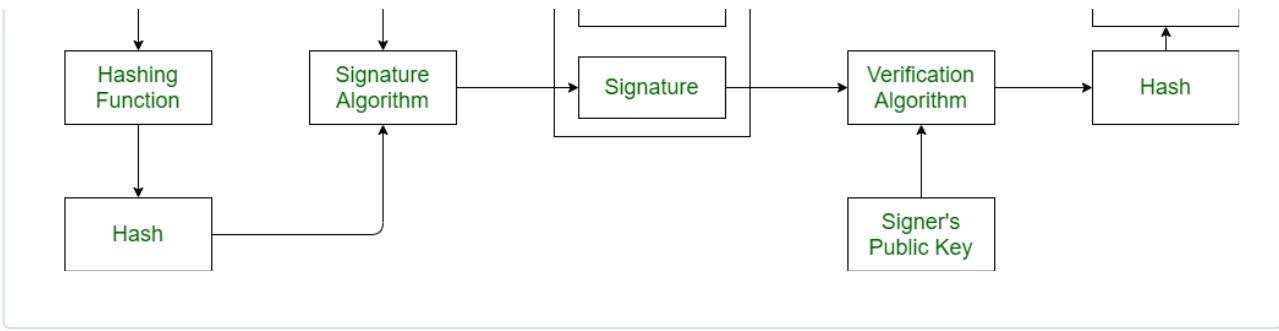


Summary

- Primarily used for secure digital signature and verification
- Developed in 1991 by **National Institute of Standards and Technology (NIST)**.
- Uses modular exponentiation and discrete logarithm to ensure mathematical security.
- Slower encryption compared to RSA.
- Faster decryption than RSA.

A basic diagram of how DSA works. ([GeeksforGeeks, 2020](#))





Due to DSA being primarily used and tested for signature and verification thus having been tested for the use case of which I will be using it for and having faster decryption than RSA - which will be used a lot more than encryption during the project - I will be using the DSA algorithm for this project.

Choosing How to Implement The Algorithm

Now that the DSA algorithm has been selected it's time to decide how to implement it: either through a library or from scratch.

This is a choice made possible by the fact that the DSA algorithm is open source and built using entirely mathematical functions so should theoretically be able to be implemented into any language.

However after looking into how the DSA algorithm works and what the V Language can handle, an issue is uncovered, one of the ways in which DSA stays secure is through the use of massively large prime numbers - currently recommended to be 2048 bits in length - however due to how Vlang integrates variables, in order to reach this size the number needs to be split up into batches of byte long values which would make processing the data required to use DSA a lot more complicated than it already is.

To clarify this doesn't mean it is impossible to write your own copy of the DSA algorithm in Vlang, it's just a lot more complicated than it would be in most other languages and puts it outside the time scope for this project.

Therefore as the 'from-scratch' method is off the table due to the complexity and therefore time scope I will instead be implementing DSA using the prebuilt `crypto.ed25519` library, which includes a generate, sign and validate function all built in, although some wrapper functions will still need to be created to handle the loading/saving of these keys and ensuring they supply the data needed in the program and not an optional return like they do currently.

Design

All that is required in this stage is to create some wrapper functions for the libraries pre-existing functions to ensure that the data supplied to the rest of the node is what we expect it to be, as well as a custom `validation` function which will be used to check that a public and private key pair match such that some data can be signed and then validated using that pair.

Pseudocode

Wrapper functions:

```
// Pseudocode
```

```

IMPORT crypto.ed25519 as dsa

FUNCTION gen_keys():
    TRY:
        // generate the keys using the dsa module
        public_key, private_key = dsa.generate_key()

        // validate the keys to check they were generated correctly
        validate_keys(public_key, private_key)

        RETURN public_key, private_key
    CATCH:
        // if anything goes wrong in the "TRY" section, this will run
        OUTPUT "Error generating keys"
        EXIT
    END TRY
END FUNCTION

```

```

FUNCTION sign(private_key, data):
    TRY:
        // sign the data supplied using the private key
        signature = dsa.sign(private_key, data)
        RETURN signature
    CATCH:
        OUTPUT "Error signing data"
        EXIT
    END TRY
END FUNCTION

```

```

FUNCTION verify(public_key, data, signature):
    TRY:
        // check if a public key matches the signature created from some data.
        verified = dsa.verify(public_key, data, signature)
        RETURN verified
    CATCH:
        OUTPUT "Error verifying data"
        EXIT
    END TRY
END FUNCTION

```

Validation function

```
// Pseudocode
```

```

FUNCTION validate_keys(public_key, private_key):
    // data is just some test binary data
    data = "Hello, world!".bytes()

    signature = sign(private_key, data)
    verified = verify(public_key, data, signature)

    IF (verified == FALSE):
        // Since private key pairs should only be used by the node itself
        // if something breaks then the node's keys are invalid.
        OUTPUT "Signature verification failed"
        EXIT
    ELSE:
        OUTPUT "Signature verified"
        RETURN TRUE
    END IF
END FUNCTION

```

Development

Converting these designs to actual code is actually pretty simple and is just a case of turning the pseudocode into something more resembling V code. The largest change here is converting from the 'try-catch' structure of error handling into V's 'or' based handling.

One other thing worth mentioning in this section is the use of error codes, if you look at any part of the code that includes a function called `exit` you will see a number being passed as a parameter, this number represents a specific type of error and can be looked up in the `ExitCodes.txt` file supplied with the distribution files.

For example, at this specific point in the project the 'Cryptography Errors' section within `ExitCodes.txt` is the following:

```

// Cryptography Errors
130 - Failed to validate keypair
140 - Error signing data
145 - Error validating data
150 - Error generating new keypair

```

Outcome

All the code in this cycle is within the file `/packages/node/src/modules/cryptography/main.v` found [here](#).

Wrapper functions

```
// vlang - src/modules/cryptography/main.v
```

```

module cryptography
import crypto.ed25519 as dsa

pub fn gen_keys() (dsa.PublicKey, dsa.PrivateKey) {
    public_key, private_key := dsa.generate_key() or {
        eprintln("Error generating keys")
        exit(150)
    }
    validate_keys(public_key, private_key)

    return public_key, private_key
}

pub fn sign(private_key dsa.PrivateKey, data []u8) []u8 {
    signature := dsa.sign(private_key, data) or {
        eprintln("Error signing data")
        exit(140)
    }
    return signature
}

pub fn verify(public_key dsa.PublicKey, data []u8, signature []u8) bool {
    verified := dsa.verify(public_key, data, signature) or {
        eprintln("Error verifying data")
        exit(145)
    }
    return verified
}

```

Validation function

```

// Vlang - src/modules/cryptography/main.v

pub fn validate_keys(public_key dsa.PublicKey, private_key dsa.PrivateKey) bool {
    data := "Hello, world!".bytes()
    signature := sign(private_key, data)
    verified := verify(public_key, data, signature)
    if !verified {
        eprintln("Signature verification failed")
        exit(130)
    } else {
        println("Signature verified")
        return true
    }
}

```

Challenges

The main challenge in this cycle was the attempted creation of a custom built DSA algorithm without using the library that I ended up using. I didn't mention this much in the [`Choosing How to Implement The Algorithm`](#) as all the reasoning in that sector for choosing a prebuilt library based system is correct, it's just how I came to understand those reasons that are the challenge.

This included the initial setup for a key generator, a probable prime generator (used to generate large numbers that are probably prime numbers to a high degree of probability) and some other utility files.

To see all the code written, and eventually scrapped, visit [the archive file here](#).

Testing

Tests

Test	Instructions	What I expect	What actually happens	Pass/Fail
1	Generate a key pair.	Keys to be generated and validated successfully without crashing.	As expected	Pass
2	Sign some random input data.	A signature to be generated without crashing.	As expected	Pass
3	Validate the data from test 2.	The data generated from test 2 should be validated successfully.	As expected	Pass

Evidence

Test 1 - Key generation tests

```
● alfier@Alfies-MacBook-Pro ~ % v run ./test.v
Testing Key Pair Generation
Signature verified
Key Pair Generation Test Passed
```

Test 2 - Key signature test

```
● alfier@Alfies-MacBook-Pro ~ % v run ./test.v
```

```
● alfier@Alfies-MacBook-Pro:src % v run ./test.v
Testing Signing
Signature verified
Generated a new key pair successfully
Signed a message successfully
Signing Test Passed
```

Test 3 - Key verification test

```
● alfier@Alfies-MacBook-Pro:src % v run ./test.v
Testing Verification
Signature verified
Generated a new key pair successfully
Signed a message successfully
Verified a signature successfully
Verification Test Passed
```

2.2.6 Cycle 6 - Setting up Inter-Nodal Communication

Design

Objectives

Now that a lot of the structure of the project has been built up and I have a bit more of an idea where I want to head with the design of the software it's time to start actually putting together the actual networking side of the blockchain. This is arguably the most important part as it was what allows the blockchain to function properly and securely.

Therefore for this cycle the objectives are to setup some kind of basic communication between nodes, which I will be doing through the use of a ping/pong API which will allow nodes to request other nodes to sign some example data ([using the cryptography functionality from the last cycle](#)) to check that they are online and owned by the wallet they claimed to be owned by.

- Create a new api route "/pong"
- Read messages sent to the api route and sign the date data sent to them
- Respond with the signature, the public key used by the node and the original message so the node who ponged can be sure that the response is correct

Usability Features

- When a pong request is received, the node should log it in the console.
- When incorrect data is fed to the route it should not crash the program so as to ensure the program keeps running for as long as possible without user intervention.
- All other errors within this route should result in soft-errors that are logged to console but do not crash the program.

Key Variables

Variable Name	Use
PingRequest	Holds a public key and message and represents the data received in the request.
PongRequest	Holds the data to be sent in response to a ping request, includes: the key supplied in the ping request; the receiver's public key; the message supplied originally; the signature of the message by the receiver.

Pseudocode

Since all of the objectives in this cycle revolve around the code I will simplify this part of the cycle by introducing the pseudocode all in two sections, the api route which will be requested by other nodes and the request function used by those nodes.

The Api Route

Due to there being so many different ways that http routes can be created based upon the language being used, this pseudocode is a little bit adjusted up to fit my requirements but should hopefully be clear enough that it doesn't matter,

```

// Psuedocode

// import functions from modules made in previous cycles
IMPORT configuration
IMPORT cryptography

// This is meant to indicate a url of form "http://---/pong/[req]"
// where req is what is fed into this "function"
HTTP_ROUTE pong (req string):

    TRY:
        req_parsed = json.decode(req)
    CATCH ERROR:
        OUTPUT "Incorrect data supplied to /pong/:req"
        RETURN HTTP.code(403)
        // a code 403 means "forbidden" in http terms
    END TRY

    // collect the configuration object from that module
    self = configuration.get_config()

    OUTPUT "Received pong request, data supplied:" + req_parsed

    // create an object that represents the response
    response = {
        pong_key: self.pub_key
        ping_key: req_parsed.ping_key
        message: req_parsed.message
        signature: cryptography.sign(self.priv_key, message)
    }

    // encode the response to be http safe
    data = json.encode(response)
    // return it to the requester
    RETURN HTTP.text(data)
END HTTP_ROUTE

```

The Request Function

```

// Psuedocode

FUNCTION ping(ref, this):
    // ref should be an ip or a domain
    message = "gfhajbsfhka" // should be random or datetime or something
    req = {
        ping_key: this.self.key,
        message: msg
    }

    OUTPUT "Sending ping request to $ref"

    // fetch domain, domain should respond with their wallet pub key/address, "pong" and a
    req_encoded = json.encode(req)

    TRY
        raw = http.get("$ref/pong/$req_encoded") or {
            }
    CATCH
        OUTPUT "Failed to ping $ref, Node is probably offline."
        RETURN false
    END TRY

    TRY
        data = json.decode(PongResponse, raw.body)
    CATCH
        OUTPUT "Failed to decode response, responder is probably using an old node version"
        RETURN false
    END TRY

    OUTPUT ref + " responded to ping request with pong" + data

    // signed hash can then be verified using the wallet pub key supplied
    IF (data.message == msg && data.ping_key == this.self.key):
        OUTPUT "Should also verify signature but I haven't implemented DSA yet"
        RETURN true
    END IF

    // if valid, return true, if not return false
    RETURN false
END FUNCTION

```

Development

Theoretically this cycle shouldn't have been that complicated but there were some weird issues with how V's default web api (V web) works and how data that will be needed on every request should be handled.

This is relevant as for the 'pong' route and a decent quantity of other routes, the keys will be needed to sign/validate different data and as each route request creates a new process to handle it, the keys need to find some way to make their way into these processes without too much inefficiency or overhead.

Outcome

The Api Route

```
// V - /src/modules/server/main.v

struct App {
    vweb.Context
}

['/pong/:req']
pub fn (mut app App) pong(req string) vweb.Result {
    req_parsed := json.decode(PingRequest, req) or {
        eprintln("Incorrect data supplied to /pong/:req")
        return app.server_error(403)
    }

    this := configuration.get_config()

    println("Received pong request.\n data supplied: $req_parsed \n Raw data supplied $req

    res := PongResponse{
        pong_key: this.self.key
        ping_key: req_parsed.ping_key
        message: req_parsed.message
        signature: cryptography.sign(this.priv_key, req_parsed.message.bytes())
    }

    data := json.encode(res)
    return app.text(data)
}
```

The Request Function

```

// V - /src/modules/server/handshake.v

module server
import configuration
import json
import net.http

// This is to establish a handshake between two nodes and should be done everytime two nodes connect
pub struct PongResponse {
    pong_key []u8
    ping_key []u8
    message string
    signature []u8
}

pub struct PingRequest {
    ping_key []u8
    message string
}

pub fn ping(ref string, this configuration.UserConfig) bool {
    // ref should be an ip or a domain
    msg := "gfhajbsfhka" // should be random or datetime or something
    req := PingRequest{ping_key: this.self.key, message: msg}

    println("Sending ping request to $ref")
    // fetch domain, domain should respond with their wallet pub key/address, "pong" and a signed hash
    req_encoded := json.encode(req)
    raw := http.get("$ref/pong/$req_encoded") or {
        eprintln("Failed to ping $ref, Node is probably offline. Error: $err")
        return false
    }

    data := json.decode(PongResponse, raw.body) or {
        eprintln("Failed to decode response, responder is probably using an old node version")
        return false
    }

    println("$ref responded to ping request with pong $data")

    // signed hash can then be verified using the wallet pub key supplied
    if data.message == msg && data.ping_key == this.self.key {
        println("Should also verify signature but I haven't implemented DSA yet")
        return true
    }

    // if valid, return true, if not return false
    return false
}

```

Challenges

The main challenge in this cycle, as mentioned in the introductory paragraph was how to get the keys for signing and validating data into the request process.

This was more complicated than I anticipated because although it may appear as if you can feed custom inputs to the `app` object which is the object that the route exists on (can be seen in the outcome code), all these values are wiped on every new request so the result is just an empty value.

So after trialing that method multiple times and realising the only way to fix that issue would be to clone the `V` web package, edit the five lines of code that cause this to happen and then recompile it - which wouldn't be too hard but would massively inflate the compile times and amount of code needed to be compiled - I moved onto trialing exporting the configuration as a constant generated on code compilation, but that would give all parts of the program access to the private key which is not great for security and therefore also not an option.

Eventually this detour led to just calling the `configuration.get_config()` function on each new request, although this does lead to loading from the save file a lot and since most of the configuration object isn't actually needed in most requests, just the keys, this will very likely be changed again in a future cycle but it'll work for now.

[Update: The refactor mentioned in this section is completed as of [Cycle 8 - Node Refactor](#)]

Testing

Tests

Test	Instructions	What I expect	What actually happens	Pass/Fail
1	Send a ping request to "https://nano.monochain.network"	The requester to successfully complete the ping request.	As expected	Pass
2	Receive a ping request as "https://nano.monochain.network"	The receiver to successfully receive and process the request.	As expected	Pass

Evidence

Sadly I forgot to collect evidence of the test in action during this stage of development, however the code of which was written in this cycle is still mostly in tact during the current stage of development and just contains more modules that haven't been written at this point.

Therefore the below evidence was generated during [Cycle 11 - Remembering Nodes](#) and contains excess outputs but does show the process working.

Sending handshake request to <https://nano.monochain.network>.
Message: 2022-09-08 10:54:27.488402

<https://nano.monochain.network> responded to handshake.
Reading file: ./monochain/config.json

Config Loaded...

Reading file: ./monochain/refs.json

Verified signature to match handshake key

Handshake with <https://nano.monochain.network> successful.

Evidence generated using newer version of code

2.2.7 Cycle 7 - Improved Webportal (Mobile Support)

Design

Objectives

Now that basic inter-nodal communication, cryptography and some other essentials are setup for the node software I think it's a good time to work on the webportal a bit more and I also to happen to have noticed that at this point the webportal does not render very well on mobile, so that's a good place to start.

- Fix the rendering issues on mobile
- Implement a separate navigation bar on mobile that is easier to use
- Improve the home page.
- Add some navigation buttons on the homepage (Inspired by ethereum.org)

Usability Features

- The site should have a consistent colour scheme so as to make the whole site recognisable and easily usable by users.
- The site should fit on a variety of screen sizes and resolutions so as to ensure as many users can access the site as possible.

Key Variables

Variable Name	Use
mobile	A boolean value which tells the navigation component whether to render the mobile or desktop view.
window	An object that stores the width and height of the browser window to be used to calculate the value for the mobile variable.
breakpoint	An integer value which when the width of window is less than, the mobile navigation bar is enabled.

Pseudocode

The code for the rendering issues are mainly just css changes to ensure that as many browser sizes as possible are supported, therefore there's not really any pseudocode to be written.

However for the new navigation bar and the new buttons on the home screen there is some actual code to write.

Starting with the new navigation bar, this will introduce two 'react hooks' that allows the 'mobile' variable referenced in key variables to be generated every time the window changes size. The first one will be called 'useWindowSize()' and will be kept in a separate file to the navigation bar file incase it is useful in other parts of the website later on, and the second hook being a basic one that re-renders any time the window variable referencing the 'useWindowSize()' hook changes.

The 'useWindowSize()' hook

```
// useWindowSize Hook
FUNCTION useWindowSize():
    // generate the window size variable
    windowHeight = {
        width: 0
        height: 0
    }

    // Handler to call on window resize
    FUNCTION handleResize():
        // Set window width/height to state
        window = {
            width: window.innerWidth,
            height: window.innerHeight,
        }
    END FUNCTION

    RELOAD_ON_CHANGE (window):
        IF (window != "undefined"):
            // Add event listener to call handleResize everytime the window is resized.
            window.addEventListener("resize", handleResize);

            // Call handler right away so state gets updated with initial window size
            handleResize();

            // Remove event listener on cleanup
            RETURN window.removeEventListener("resize", handleResize);
        END IF
    END RELOAD_ON_CHANGE

    RETURN windowHeight;
END FUNCTION
```

The mobile navigation bar hook

```
// initialise the nav bar as desktop
mobile = false

// set the breakpoint as a width of 700 pixels
breakpoint = 700

RELOAD_ON_CHANGE (window.width):
    // reloads any time the window.width changes

    IF (window.width < breakpoint):
        mobile = true

    ELSE IF (window.width > breakpoint):
        mobile = false

    END IF
END
```

Development

Most of the development for implementing these changes were pretty simple as the logic for deciding whether to use the mobile or desktop nav bar is very simple and the buttons have basically no logic, so although it looks like a lot of code, most of it is just styling and layering to make it actually look nice and reusable in different parts of the website.

- ⓘ The outcome of these changes can be visited here:

<https://monochain-webportal-5ph33emep-alfieran.vercel.app>

Although if I switch web hosting providers at somepoint between writing this (26/10/22) and when you are reading it then the link will probably not work anymore.

Outcome - Navigation Bar

useWindowSize() Hook

This simply just generates a few variables and functions that will run and change the value of the `windowSize` variable as the `window` object - an object supplied by the browser and not written by myself - changes in size and shape.

```

import { useState, useEffect } from "react";

// Hook
export default function useWindowSize() {
  const [windowSize, setWindowSize] = useState<{
    width: undefined | number;
    height: undefined | number;
}>({
  width: undefined,
  height: undefined,
});

// Handler to call on window resize
function handleResize() {
  // Set window width/height to state
  setWindowSize({
    width: window.innerWidth,
    height: window.innerHeight,
  });
}

useEffect(() => {
  // only execute all the code below in client side
  if (typeof window !== "undefined") {
    // Add event listener
    window.addEventListener("resize", handleResize);

    // Call handler right away so state gets updated with initial window size
    handleResize();

    // Remove event listener on cleanup
    return () => window.removeEventListener("resize", handleResize);
  }
}, []); // Empty array ensures that effect is only run on mount
return windowSize;
}

```

Mobile hook

Since this code is part of the NavBar component, there's a lot more code that isn't relevant here so this only shows the relevant code, but the entire file can be found [here](#). As shown below this code simply changes the state of the 'mobile' variable as the width of the window goes above or below the 'breakpoint' variable - currently set to 700 pixels.

```

const NavBar = () => {
  // there is some other code ran here

  // generate the required constants
  const window = useWindowSize();
  const breakpoint = 700;
  const [mobile, setMobile] = useState<boolean>(true);

  // create the hook
  useEffect(() => {
    if ((window.width || 0) < breakpoint) {
      setMobile(true);
    } else if ((window.width || 0) > breakpoint) {
      setMobile(false);
    }
  }, [window.width]);

  // then moves onto to the code for some other hooks & the actual components
}

```

The mobile and desktop rendering splitter

Once again, this code is part of the NavBar component, but since that file is so big and including all of it would make it unclear which parts have been changed, so only relevant sections have been included here.

The top section shows the desktop navigation bar, with the div that houses it having the parameter `hidden={mobile}` such that as the mobile view is toggled it will disappear from the user's view and then the mobile navigation component will appear - shown by the parameter `hidden={!mobile}`

```

<div hidden={mobile}>
  <Flex css={NavBarCss} justifyContent={"center"}>
    {Links.map((data) => {
      if (data.type === "component") {
        return (
          <Flex flexDir={"column"} key={data.name}>
            <Button
              bg={"/"}
              my={0}
              mx={LinkMarginAmount}
              p={0}
              _hover={LinkHover}
              _active={LinkHover}
              onClick={() => {
                data.state.change(!data.state.value);
              }}
            >
              <Text
                fontSize={"xl"}
                fontWeight={"normal"}

                h={"full"}
                w={"full"}
                lineHeight={"base"}
                pt={1.5}
              >
                {data.name}
              </Text>
            </Button>
          </Flex>
        );
      } else {
        return (
          <Link
            key={data.name}
            href={data.href}
            mx={LinkMarginAmount}
            isExternal={!!data.external}
            _hover={LinkHover}
          >
            <Text fontSize={"xl"} h={"full"} w={"full"} pt={1.5}>
              {data.name}
            </Text>
          </Link>
        );
      }
    ))}
  </Flex>
{Links.filter((item) => item.type === "component").map((data) => {
  if (data.type === "component") {
    return (
      <Collapse key={data.name} in={data.state.value} dir="top">

```

```
        <data.comp />
    </Collapse>
);
}
})}
</div>
<MobileNav hidden={!mobile} />
```

Mobile Navigation component

This is the component that is shown to users who are on mobile devices - or any browser with a window width of less than 700 pixels and simply represents a toggle-able side menu that can open and close as the user wishes. This is in exchange for the desktop navigation bar which has all the options along the top of the screen and is not toggle-able.

```
import {
  Button,
  Drawer,
  DrawerBody,
  DrawerContent,
  DrawerHeader,
  DrawerOverlay,
  Flex,
  Link,
  Text,
  useDisclosure,
} from "@chakra-ui/react";
import { NavBarCss } from "../styles/navBar";
import { scaling_button } from "../styles/buttons";
import React from "react";

const MobileNav = (props: { hidden?: boolean }) => {
  const Links: { name: string; href: string }[] = [
    { name: "Home", href: "/" },
    { name: "Learn More", href: "/info" },
    { name: "Wallet", href: "/wallet" },
  ];
  const { isOpen, onOpen, onClose } = useDisclosure();

  if (props.hidden) return null;
  return (
    <>
      <Flex
        css={NavBarCss}
        justifyContent={"space-between"}
        alignItems={"center"}
      >
        <Flex css={scaling_button()} pl={2}>
          <Link _hover={{ textDecoration: "none" }} href="/">
            <Text m={2}>Home</Text>
          </Link>
        </Flex>
        <Button css={scaling_button()} onClick={onOpen} px={3}>
          <Text m={2}>Menu</Text>
        </Button>
      </Flex>

      <Drawer placement="right" onClose={onClose} isOpen={isOpen}>
        <DrawerOverlay />
        <DrawerContent>
          <DrawerHeader borderBottomWidth="2px">
```

```

<Flex
  dir={"row"}
  w={"100%"}
  justifyContent={"space-between"}
  alignItems={"center"}
>
  <Text>MonoChain</Text>
  <Button css={scaling_button()} onClick={menuOnClose}>
    X
  </Button>
</Flex>
</DrawerHeader>
<DrawerBody>
  {Links.map((linkObj) => (
    <Link
      href={linkObj.href}
      _hover={{ textDecoration: "none" }}
      key={linkObj.name}
    >
      <Flex
        borderWidth={"2px"}
        borderRadius={"lg"}
        my={3}
        p={2}
        fontSize={"lg"}
        bg={"rgba(0,0,0,0.1)"}
        _active={{ bg: "rgba(0,0,0,0.2)" }}
      >
        <Text>{linkObj.name}</Text>
      </Flex>
    </Link>
  )));
</DrawerBody>
</DrawerContent>
</Drawer>
</>
);
};

export default MobileNav;

```

Outcome - Homepage buttons

Since the buttons to be produced will all be identical apart from their contents and where they send the user, the button only needs to be coded once and can then be looped over for each item in an array of contexts for the button to show.

The context info for the buttons

This is simple an array of objects which abide by the type 'gridItemProps' such that they all have exactly the same parameters and are easily usable by the button code.

```
type gridItemProps = {
  link: string;
  title: string;
  subtitle: string;
};

const gridItems: gridItemProps[] = [
  {
    link: "/wallet",
    title: "Wallet",
    subtitle: "Create and manage your wallet.",
  },
  {
    link: "/info",
    title: "Learn More",
    subtitle: "Learn more about the MonoChain.",
  },
  {
    link: "/download",
    title: "Setup a Node",
    subtitle:
      "Download a node to support the MonoChain in exchange for a fee.",
  },
  {
    link: "https://github.com/AlfieRan/A-Level-Project",
    title: "View the Code",
    subtitle: "View the open source code for the default Node and webportal.",
  },
];
```

The button code

This turns the data supplied by each of the objects within the 'gridItems' array into a button with the context supplied by that object.

```

<Grid gridTemplateColumns={[ "repeat(1, 1fr)", "repeat(2, 1fr)" ]}>
  {gridItems.map((item) => (
    <GridItem
      key={item.title}
      m={2}
      maxW={"100%"}
      w={[ "4xs", "md" ]}
      h={[ "2xs", "xs" ]}

    >
    <Link
      href={item.link}
      _hover={{ textDecoration: "none" }}

      w={"100%"}
      h={"100%"}

    >
    <Center
      w={"100%"}
      h={"100%"}
      p={2}
      bg={"rgba(100,100,255,0.2)"}
      transitionDuration={"0.15s"}
      _hover={{

        bg: "rgba(100,100,255,0.3)",
        transform: "scale(1.03)",

      }}
      _active={{

        bg: "rgba(100,100,255,0.1)",
        transform: "scale(0.97)",

      }}
      borderRadius={"xl"}
      flexDir={"column"}

    >
    <Text fontSize={"2xl"} fontWeight={"semibold"}>
      {item.title}
    </Text>
    <Text fontSize={"lg"}>{item.subtitle}</Text>
  </Center>
  </Link>
  </GridItem>
  ))}
</Grid>

```

Challenges

The main challenge of this cycle was simply just tweaking css enough to ensure that the new ui looks good on as many devices and resolutions as possible, which involved changing my simulated resolution using the chrome developer tools to ensure that it looked good.

Testing

Tests

Test	Instructions	What I expect	What actually happens	Passed?
1	Navigate throughout the website on a desktop/laptop device	The website to stay "intact" with no components becoming hidden or breaking.	As Expected	<input type="checkbox"/>
2	Navigate throughout the website on a phone/small device	The website to stay "intact" with no components becoming hidden or breaking.	As Expected	<input type="checkbox"/>
3	Open the site on desktop device.	The desktop styled navigation bar should appear along the top of the screen.	As Expected	<input type="checkbox"/>
4	Open the site on mobile device.	The mobile styled navigation bar should appear with a menu button to open and close the bulk of the contents.	As Expected	<input type="checkbox"/>

Evidence

Test 1 & 3

This showcases the evidence for both test 1 and 3, both of which are regarding the site on a desktop device as is emulated here.



Test 2 & 4

This showcases the evidence for both test 2 and 4, both of which are regarding the site on a mobile device as is emulated here. (In this case an iPhone 13 Pro Max was chosen to be emulated.)



2.2.8 Cycle 8 - Node Refactor

Design

Objectives

After beginning to plan out the slightly more advanced inter nodal communication ([which should be in a later cycle not too much later than this one](#)) I have come to the realisation that housing the cryptography and configuration module in the same place is probably not a good idea.

This is because after a few experiments I realised that without editing and recompiling it, the V-Web module that I'm using to host the api routes for the nodes currently prevents the loading of custom non-public variables into the web object to be used within those routes. What that means is that I can't setup the node's cryptography keys to be passed around the node whenever it receives a new api request without making them public and accessible by all parts of the program (which isn't great for security) and hence need to find a work around.

There are two main options here:

1. Reload the configuration object every time the program needs the cryptography keys to do anything.
2. Store the keys as a separate object and just reload them when they're needed.

Since separating the configuration and key objects helps reduce the amount of data that needs to be loaded and saved per time unit and is better for security, I will be going with the second option.

- Separate the keys into their own object.
- Allow the private key to remain private and unusable from any non-direct functionality
- Refactor any sections of code that need to be updated to support this new method
- Fix any newly generated errors and get back to at least what was working before this refactor

Usability Features

- Keys are stored in a seperate file so that users can copy and paste just their key file to run additional nodes without having to manually enter their key data.
- Ensure that all files for this project are stored within one "monochain" folder so as to keep them contained and not overrun the user's file system.

Key Variables

Variable Name	Use
Keys	This is the keys object that holds the private and public keys as-well as some functions for using these keys (sign, verify, etc).
key_path	This is a string that represents the path of the file where the keys object is stored and loaded from.

Pseudocode

The keys object

This is the object that stores both the private, public key and the two key functions that need access to private key. The first being the sign function which signs data using the private key, and the validate function that ensures the public and private key within the object are matching and work together.

```

// Psuedocode
IMPORT cryptography AS dsa
// dsa is used as the name since that is the cryptography algorithm being used.

OBJECT Keys:
    priv_key
    pub_key

    FUNCTION validate_keys(this):
        // defines some random data to check a key pair with
        data = "Hello, world!"

        // signs that data using the private key
        signature = this.sign(data)

        // validate that data with that signature and key pair
        verified = verify(this.pub_key, data, signature)

        IF (verified == false): // if verification failed
            OUTPUT "Signature verification failed"
            EXIT      // exit the program
        ELSE:
            // verification succeeded
            OUTPUT "Signature verified"
            RETURN true      // valid so return true
        END IF
    END FUNCTION

    FUNCTION sign(this, data) {
        // wrap the sign function to prevent having conditional data throughout program
        TRY:
            signature = dsa.sign(this.priv_key, data)
            RETURN signature
        CATCH:
            OUTPUT "Error signing data"
            EXIT
        END TRY
    END FUNCTION
END OBJECT

FUNCTION verify(public_key, data, signature):
    // wrap verify function to prevent having conditional data throughout program.
    TRY:
        verified = dsa.verify(public_key, data, signature)
        RETURN verified
    CATCH:
        OUTPUT "Error verifying data"
        RETURN false
    END TRY
END FUNCTION

```

The keys handler

This is what allows the keys to be stored, loaded and generated into the keys object.

```

// Psuedocode
IMPORT utils
IMPORT cryptography AS dsa
IMPORT json

FUNCTION failed_to_get_keys(key_path):
    OUTPUT "Could not load keys from file, would you like to generate a new pair?"
    IF utils.ask_for_bool():
        new_keys = gen_keys()
        failed = utils.save_file(key_path, json.encode(new_keys))

        IF failed {
            println("Cannot continue, exiting...")
            EXIT
        END IF

        RETURN new_keys
    ELSE
        OUTPUT "Cannot operate without a keypair, Exiting..."
        EXIT
    END IF
END FUNCTION

FUNCTION get_keys(key_path):
    raw = utils.read_file(key_path)

    IF (!raw.loaded):
        RETURN failed_to_get_keys(key_path)
    END IF

    TRY:
        keys = json.decode(Keys, raw.data)
        OUTPUT "Keys loaded from file."
        RETURN keys
    CATCH:
        RETURN failed_to_get_keys(key_path)
    END TRY

END FUNCTION

FUNCTION gen_keys():
    TRY:
        public_key, private_key = dsa.generate_key()

        keys = {
            pub_key: public_key,
            priv_key: private_key,
        }

        keys.validate_keys()

        RETURN keys
    
```

```
CATCH:  
    OUTPUT "Error generating keys"  
    EXIT  
END TRY  
END FUNCTION
```

Development

The end result for this code involves two files, one of which handles the creation and functions of the keys object, and the other handles the loading, saving and creation of that object.

The benefit of this is that it allows the private key to remain as a private property so that the only functions that can access it are those that are a part of the keys object and used for signing data. Therefore although if part of the program was infected with some kind of malware it would allow that malware to sign data/transactions which is obviously not good, it wouldn't give them access to the keys directly hence making it harder for the bad actor responsible for the malware to copy the keys and gain complete control of the wallet.

Outcome

The keys object.

```

// /packages/node/src/modules/cryptography/main.v
module cryptography      // this module
import crypto.ed25519 as dsa      // an external module used to handle all the dsa stuff - too cool

pub struct Keys {
    priv_key []u8    //dsa.PrivateKey
    pub:
        pub_key []u8    //dsa.PublicKey
}

type KeysType = Keys

pub fn (this Keys) validate_keys() bool {
    data := "Hello, world!".bytes()          // defines some random data to check a key pair
    signature := this.sign(data)    // signs that data using the private key
    verified := verify(this.pub_key, data, signature)           // validate that data was signed correctly
    if verified == false { // if verification failed
        eprintln("Signature verification failed")
        exit(130)      // exit with error code 130
    } else {           // verification succeeded
        println("Signature verified")
        return true     // valid so return true
    }
}

pub fn (this Keys) sign(data []u8) []u8 {
    // wrap the sign function to prevent having conditional data throughout program.
    signature := dsa.sign(this.priv_key, data) or {
        eprintln("Error signing data")
        exit(140)
    }
    return signature
}

pub fn verify(public_key dsa.PublicKey, data []u8, signature []u8) bool {
    // wrap verify function to prevent having conditional data throughout program.
    verified := dsa.verify(public_key, data, signature) or {
        eprintln("Error verifying data")
        return false
    }
    return verified
}

```

The keys handler.

```

// V code from /packages/node/src/modules/cryptography/keyHandling.v
module cryptography
import utils
import crypto.ed25519 as dsa
import json

fn failed_to_get_keys(key_path string) (Keys){
    println("Could not load keys from file, would you like to generate a new pair?")
    if utils.ask_for_bool(0) {
        new_keys := gen_keys()
        failed := utils.save_file(key_path, json.encode(new_keys), 0)
        if failed {
            println("Cannot continue, exiting...")
            exit(215)
        }
        return new_keys
    } else {
        eprintln("Cannot operate without a keypair, Exiting...")
        exit(1)
    }
}

pub fn get_keys(key_path string) (Keys) {
    raw := utils.read_file(key_path, true)

    if !raw.loaded {
        return failed_to_get_keys(key_path)
    }

    keys := json.decode(Keys, raw.data) or {
        return failed_to_get_keys(key_path)
    }

    println("Keys loaded from file.")
    return keys
}

pub fn gen_keys() (Keys) {
    // This is just a wrapper function to prevent the rest of the code having to deal with
    // throughout the rest of the program

    public_key, private_key := dsa.generate_key() or {
        eprintln("Error generating keys")
        exit(150)
    }
    keys := Keys{
        pub_key: public_key,
        priv_key: private_key,
    }

    keys.validate_keys()
}

```

```
    return keys  
}v
```

Challenges

There were two key challenges to overcome during this development phase, the first one being a challenge I knew I'd have to overcome and the second being a very annoying bug. I've split these challenges into two separate sections for the sake of ease of reading.

Managing the storage of multiple files

Since the configuration settings are already saved in their own file and I am now going to be saving the keys in a separate file and I don't want to keep dumping files into the user's home directory, it's probably a good idea to start storing files within a folder to keep everything in one place.

Hence all files will now be stored within the `/monochain/` folder so as to improve organisation. As a byproduct of this, I needed to write some form of script to check if the 'monochain' folder actually exists before anything was saved to it and if not, create the folder and then save a file to it.

This resulted in the adaptation of the current `save_file` function (which previously just wrapped the `os.write_file` function so as to reduce error handling) to the following:

```

// This is within the 'utils' module
pub fn save_file(path string, data string, recursion_depth int) (bool) {
    mut failed := false

    // splits the requested path into an array of sub-directories
    dirs := path.split("/")
    if dirs.len > 1 {
        // this means the file is in a folder

        // assemble the directory path
        mut dir := ""
        mut i := 0
        for i < dirs.len - 1 {
            dir += dirs[i] + "/"

            // check if path exists
            if !os.exists(dir) {
                // the directory doesn't exist, so we need to create it
                os.mkdir(dir) or {
                    // something went wrong creating the directory, return
                    eprintln("[utils] Failed to create the directory at path")
                    failed = true
                    return true
                }
                println("[utils] Created directory: " + dir)
            }
        }

        //increment to the next sub-directory in the path
        i++
    }

    os.write_file(path, data) or { // try and write data to file
        // if it failed, run the recursion depth checker to ensure there hasn't been too many tries
        eprintln('[utils] Failed to save file, error ${err}\nTrying again.')
        if recursion_depth >= 5{
            eprintln("[utils] Failed to save file too many times.")
            failed = true
        } else {
            // if the recursion depth is less than 5, just try again.
            failed = save_file(path, path, recursion_depth + 1)
        }
    }

    return failed
}

```

This code also allows for the creation of sub-directories within the main folder as it simply loops through all sections of the file path. An example of this would be the creation of a file at path `/monochain/subfolder1/subfolder2/file.txt` which is supported with this code.

Overcoming a tricky SIGSEGV error

One of the major issues with Vlang that I have been dealing with over the last few development cycles is the SIGSEGV error. This is an error that occurs when a program tries to access a memory address that it has not been given permission to access by the operating system and one that occurs for a variety of different reasons.

The main problem with this error is that because of how the Vlang compiler deals with these errors is to just tell the developer that one has occurred without any hints of why or where it could've happened, meaning that as soon as a SIGSEGV error occurs the developer is completely on their own to solve it.

Although I have encountered a few of these errors before, they have all been after changing only a few lines of code and hence I can simply tweak those few lines of code until the error is fixed. However in this case the error occurred after I had written an entire file of code and included it to be compiled, so I had to spend a few hours checking each line of code individually until eventually I discovered that the error wasn't even directly in that file at all and was instead caused by a type declaration loop across two separate files.

This type declaration loop was caused due to a struct being defined based upon a type, then that type being defined based upon the struct.

```
Struct A {TypeB}
```

```
Type B = Struct A
```

This was annoying because in most other languages if this was to occur, the compiler would give you a nice error message due to it being fairly easy to catch prior to compilation, but in Vlang that was not the case.

Solving this error after discovering this was pretty simple, the challenge was just finding out the cause of the error in the first place.

Testing

To assist in the testing of this module I created a batch of automated tests

Tests

Test	Instructions	What I expect
1	Generate a key pair.	The program not to exit, which will happen if the key pair cannot be generated.
2	Generate a key pair and sign some data.	The program not to exit, which will happen if the key pair cannot be generated or sign the data.
3	Generate a key pair, sign and then verify some data.	The program not to exit and to verify that the message matches the signature.

Evidence

```
● alfier@Alfies-MacBook-Pro: ~ v -stats test ./src/modules/cryptography
---- Testing...
----- V -----
  source code size: 30683 lines, 822578 bytes
generated target code size: 30053 lines, 1134573 bytes
compilation took: 583.787 ms, compilation speed: 52558 vlines/s
running tests in: /Users/alfier/WebstormProjects/A-Level-Project/packages/node/src/modules/cryptography/crypto_test.v
[testing] Testing Key Pair Generation
[cryptography] Signature verified
[testing] Key Pair Generation Test Passed
OK [1/3] 60.720 ms 11 asserts | main.test_gen_keys()
```

Evidence for test 1 being successful.

```
[testing] Testing Signing
[cryptography] Signature verified
[testing] Generated a new key pair successfully
[testing] Signed a message successfully
[testing] Signing Test Passed
OK [2/3] 79.540 ms 17 asserts | main.test_sign()
```

Evidence for test 2 being successful

```
[testing] Testing Verification
[cryptography] Signature verified
[testing] Generated a new key pair successfully
[testing] Signed a message successfully
[testing] Verified a signature successfully
[testing] Verification Test Passed
OK [3/3] 83.608 ms 19 asserts | main.test_verify()
Summary for running V tests in "crypto_test.v": 47 passed, 47 total. Runtime: 223 ms.
```

Summary for all V _test.v files: 1 passed, 1 total. Runtime: 1072 ms, on 1 job.

Evidence for test 3 being successful

Automation Code

Test 1

```
pub fn test_gen_keys() {
    println("[testing] Testing Key Pair Generation")
    cryptography.gen_keys()
    // because the gen_keys function hasn't exited yet, we know that the keys are generated
    println("[testing] Key Pair Generation Test Passed")
    assert true
}
```

Test 2

```
pub fn test_sign() {
    println("[testing] Testing Signing")
    keys := cryptography.gen_keys()

    println("[testing] Generated a new key pair successfully")
    message := time.utc().str().bytes()

    signed := keys.sign(message)
    println("[testing] Signed a message successfully: $signed")
    println("[testing] Signing Test Passed")

    // because the sign function hasn't exited yet, we know that the signature is valid
    assert true
}
```

Test 3

```
pub fn test_verify() {
    mut failed := false
    println("[testing] Testing Verification")
    keys := cryptography.gen_keys()

    println("[testing] Generated a new key pair successfully")
    message := time.utc().str().bytes()

    signature := keys.sign(message)
    println("[testing] Signed a message successfully")

    if cryptography.verify(keys.pub_key, message, signature) {
        println("[testing] Verified a signature successfully")
    }
}
```

```
        println("[testing] Verification Test Passed")
        failed = false
    } else {
        println("[testing] Verification failed")
        println("[testing] Verification Test Failed")
        failed = true
    }

assert failed == false
}
```

2.2.9 Cycle 9 - Protocol Updates

Design

A lot of the work so far has been based around the software, in the form of the node program and the webportal, however there is a third portion of this project which is equally valuable to its creation, the monochain protocol itself.

Hence in this cycle I will be expanding some of the concepts initialised in the analysis of this project and therefore this cycle will take a very different form to the other cycles in the project.

Now the reason I'm only just talking about the protocol this late into the project is because a lot of the initial sections were theorised and planned in the [analysis part of this project](#). However due to the complexity of a blockchain protocol, and hence the monochain protocol, this was only the major framework of the protocol and the details need to be ironed out as the project proceeds.

In this cycle the two sections of the protocol to be looked more into are:

- The regulation of the speed at which blocks are created
- The initial theorisation of voting on blockchain parameters.

Objectives

As specified in the above there will be two main areas of the protocol to be examined and improved. Those areas being:

- Regulating block production speed
- Voting on blockchain parameters

Let's start with the regulation of the speed at which blocks are produced

In order for the blocks to be properly propagated amongst all the nodes in the network, blocks cannot just be created thousands of times per second and have it hoped that all the nodes receive them, hence some kind of regulation method needs to be created to limit the amount of blocks created per time period. Initially this will be purely a conceptual, protocol change, but time permitted should be implemented into the node software in a future cycle.

- Conceptualise how blocks are going to be regulated
- Don't use a constant timing based method (e.g. only allowing a new block every 10s) as this could lead to multiple nodes publishing new blocks as soon as that time period completes, causing forks.

Leading on from here is the voting concept

Voting is essential to any decentralised power structure and since this blockchain will be a decentralised structure that will most likely require updates and adjustments into the future, it makes sense for the monochain to allow voting to change the specifics of the fundamentals of which it is based upon. The requirements for this are that the specific parameters that control the functionality and specifics of the blockchain should be able to be changed and tweaked through voting by users of the system.

- Votes should have the power to change parameters of the blockchain, e.g. transactional speed.
- Votes should allow anyone with a wallet to vote.

Development

Transaction Speed

In order to ensure blocks can be created whilst there are low transaction rates yet restricting blocks from being created too quickly, each block may not be created more than once per 10 seconds, with each block having a maximum of 5000 transactions, setting the maximum transactions per second to 500, far more than Ethereum 1.0's 30 per second although much lower than Visa's average of 1700 per second and Ethereum 2.0's theoretical 100,000 per second.

This will be achieved by setting the number of seconds after the last block creation at which a block can be registered to be:

```
time = 5 + 30000(transactions + 50)^-1
```

Where the transactions variable is the number of transactions in this new block, promoting miners to add as many transactions into a block as possible, such to reach this time constraint quicker by lowering it.

An upgrade to increase this could be introduced in the future through the use of the system's wallet, as this rule will be included into the blockchain as a smart contract owned by the system's wallet. However, this will need to be done through voting.

Voting

Voting is essential to the way the Monochain works, as it creates a truly decentralised system of which no individual party has control of whilst ensuring the system can adapt and change into the future.

The way in which this will work is the following:

1. A mining Node will register a vote by creating two new voting "wallets", where the public key for the wallets will be in the format "vote:[uid]:true" and "vote:[uid]:false" and there is no private key.
2. Users will then vote by sending a "vote token" to one of the two wallets in the pair, where they do not need any vote tokens to send a vote token, they just cannot send a token to the same vote more than once and cannot send a vote token to both "wallets".
3. If a vote receives less than 100 votes or 75% of the quantity of votes from the previous vote then it is immediately cancelled as invalid (although as this is a system rule it can be changed through voting).
4. If a vote receives a majority of no votes then the vote is cancelled and nothing happens, if the vote receives a majority yes vote then the change it wishes to put into place is granted and that change is made to the system wallet.
5. Initially votes cannot exceed more than once per 1209600 seconds (two weeks), but this can be set to be controlled by some algorithm once the methodology is proved to work

2.2.10 Cycle 10 - Basic Inter-Nodal Communication

Design

Objectives

The objective for this cycle is to turn the [basic inter-nodal communication from Cycle 6](#) into something a little more advanced, allowing Nodes to start using post requests rather than get requests so that the data can be supplied within the body of the request, and other similar quality of life upgrades.

Alongside this there is also a very bad security flaw with the way in which Nodes sign data sent to their pong routes, and that is that they will just sign anything sent their way. This means that node A could just ask node B to sign a transaction that gives all of B's assets to A and then B would just do it and send the signed transaction back to A. This is obviously bad and needs to be fixed.

- Make nodes check the data sent to their pong route and only sign it if it is a date/time string.
- Convert pong request from a get request to a post request.
- Move pong request data from the query of the request to the body.
- Convert ping/pong terminology into a "handshake" with sufficient renaming

Usability Features

The main usability feature introduced within this cycle is the renaming and structuring of the "ping/pong" functions and routes within the node software to a "handshake" route which more accurately describes what is actually going on.

Key Variables

Variable Name	Use
config	Stores all the configuration object's data, such as the node's reference to itself, its port and other essential information.
keys	Contains the keys object, which has the ability to sign, validate and verify data.

Pseudocode

Only signing date-time objects

Luckily implementing this, and dramatically improving the node software's current security, should actually be pretty easy. All the code will need to do is attempt to parse the message received within the handshake (ping/pong) request as a date-time object and if it succeeds, then the message is okay to be signed and if not then fail the request and do not sign it.

Parsing date-time objects is different for every language and standard library so for this pseudocode the function `Parse_Date` will represent a function which takes a string in the format `YYYY-MM-DD hh:mm:ss` and either returns a date-time object in some format or errors out.

```
// Pseudocode

FUNCTION Valid_Message(message):
    TRY:
        date = Parse_Date(message)
        return date

    CATCH ERROR:
        return false

    END TRY
END FUNCTION
```

Converting the handshake route from a get request to a post request

As both the conversion from a get request to a post request and converting from using queries to using the body to send data is so similar I'll use the below examples to summarise both changes.

Api Route

Because this is based upon code from a previous cycle where I introduced the nodal communication, I've copied that pseudocode as the base and then edited it from there. [Visit here to see the original Pseudocode](#)

```

// Pseudocode

// import functions from modules made in previous cycles
IMPORT configuration
IMPORT cryptography

// In this case request has changed to be the body of the http request
HTTP_POST_ROUTE handshake (request):

    TRY:
        req_parsed = json.decode(request)
    CATCH ERROR:
        OUTPUT "Incorrect data supplied to handshake"
        RETURN HTTP.code(403)
        // a code 403 means "forbidden" in http terms
    END TRY

    OUTPUT "Received handshake request from node claiming to have the public key"
    + req_parsed.initiator_key

    // using the function defined earlier in this cycle
    time = Valid_Message(req_parsed.message)

    IF time == False:
        OUTPUT "Incorrect time format supplied to handshake by node claiming to be"
        + req_parsed.initiator_key

        // this is where a negative grudge would then be stored but that's for a
        // future cycle.

        RETURN HTTP.code(403)
    ENDIF

    // time was okay, so store a slight positive grudge
    OUTPUT "Time parsed correctly as:" + time

    // how the keys are used has also changed due to the node refactor.
    config = configuration.get_config()
    keys = cryptography.get_keys(config.key_path)

    // create an object that represents the response
    response = {
        responder_key: self.pub_key
        initiator_key: req_parsed.initiator_key
        message: req_parsed.message
        signature: keys.sign(message)
    }

```

```
data encode then response to be http safe  
  
// return it to the requester  
OUTPUT "Handshake Analysis Complete. Sending response..."  
return HTTP.text(data)  
END HTTP_ROUTE
```

Requester Function

```

// Psuedocode

IMPORT time
IMPORT json
IMPORT http

IMPORT cryptography

FUNCTION start_handshake(ref, this):
    // ref should be an ip or a domain

    message = time.now() // set the message to the current time since epoch
    // message = "invalid data" // Invalid data used for testing

    request = {initiator_key: this.self.key, message: msg}

    OUTPUT "Sending handshake request to" + ref + ".\nMessage:" + message
    // fetch domain, domain should respond with their wallet pub key/address, "pong" and a signed hash
    request_encoded := json.encode(request)

    TRY:
        raw_response = http.post("$ref/handshake", req_encoded)
    CATCH:
        OUTPUT "Failed to shake hands with" + ref + ", Node is probably offline."
        RETURN FALSE
    END TRY

    if raw_response.status_code != 200 {
        OUTPUT "Failed to shake hands with" + ref + ", may have sent incorrect data, or node is offline"
        RETURN FALSE
    }

    TRY:
        response = json.decode(HandshakeResponse, raw.body)
    CATCH:
        OUTPUT "Failed to decode handshake response, responder is probably using an old version of the protocol"
        RETURN FALSE
    END TRY

    OUTPUT "\n" + ref + "responded to handshake."

    // signed hash can then be verified using the wallet pub key supplied
    IF (response.message == message && response.initiator_key == this.self.key):

        IF (cryptography.verify(response.responder_key, response.message.bytes(), response.signature)):
            OUTPUT "Verified signature to match handshake key\nHandshake with $ref successful"
            RETURN TRUE
        END IF

        OUTPUT "Signature did not match handshake key, node is not who they claim to be"
        // this is where we would then store a record of the node's reference/ip address
        RETURN FALSE

```

```

    END IF

    OUTPUT "Handshake was not valid, node is not who they claim to be."
    OUTPUT response
    // if valid, return true, if not return false
    RETURN FALSE
END FUNCTION

```

Development

The development for this cycle was primarily based upon changing the code, pushing the changes, downloading the new code and running it on two nodes and seeing if either of them had communication issues. This was fairly repetitive but due to having ssh access to both servers It wasn't too time consuming and allowed me to just keep testing the new code until it stopped having errors, which lowered the error rates a lot due to so many tests as I was programming.

Outcome

Sigining Date-Time Objects

Converting the pseudocode to actual code for this was pretty simple, although I ended up keeping the parsing within the main handshake route function rather than splitting it out into a separate function, so that's why the code isn't in a function below.

```

// V code - within the "pong" route in /src/modules/server/main.v

// with this version of the node software all messages should be time objects
time := time.parse(req_parsed.message) or {
    eprintln("Incorrect time format supplied to /pong/:req by node claiming to be $req_parsed")
    return app.server_error(403)
}

// time was okay, so store a slight positive grudge
println("Time parsed correctly as: $time")

```

This code can be found in [this commit here](#), although bear in mind that the code above removed some in-code comments meant for future use in a different cycle that is not in development and is just being planned at the moment.

Converting the handshake route from a get request to a post request

The Api route

This is the endpoint at which a node will send a http POST request to in order to request a handshake response in accordance to the details sent in the request.

```

// V code - within the "handshake" route in /src/modules/server/handshake.v

['/handshake'; post]
pub fn (mut app App) handshake_route() vweb.Result {
    body := app.req.data

    req_parsed := json.decode(HandshakeRequest, body) or {
        eprintln("Incorrect data supplied to /handshake/")
        return app.server_error(403)
    }

    println("Received handshake request from node claiming to have the public key: $req_pa

    // THIS IS THE SECTION WRITTEN UNDER THE TITLE "Signing Date-Time Objects"
    // with this version of the node software all messages should be time objects
    time := time.parse(req_parsed.message) or {
        eprintln("Incorrect time format supplied to handshake by node claiming to be $time")
        return app.server_error(403)
    }

    // time was okay, so store a slight positive grudge
    println("Time parsed correctly as: $time")
    // THIS IS THE SECTION WRITTEN UNDER THE TITLE "Signing Date-Time Objects"

    config := configuration.get_config()
    keys := cryptography.get_keys(config.key_path)

    res := HandshakeResponse{
        responder_key: keys.pub_key
        initiator_key: req_parsed.initiator_key
        message: req_parsed.message
        signature: keys.sign(req_parsed.message.bytes())
    }

    data := json.encode(res)
    println("Handshake Analysis Complete. Sending response...")
    return app.text(data)
}

```

The Handshake Requester

This code handles the assembly of a handshake request, then the http request and finally the handling of the returned data then returns a boolean value that represents whether the handshake was successful or not.

```

// V code - /src/modules/server/handshake.v

pub fn start_handshake(ref string, this configuration.UserConfig) bool {
    // ref should be an ip or a domain
    msg := time.now().format_ss_micro() // set the message to the current time since epoch
    // msg := "invalid data" // Invalid data used for testing
    req := HandshakeRequest{initiator_key: this.self.key, message: msg}

    println("\nSending handshake request to ${ref}.\nMessage: $msg")
    // fetch domain, domain should respond with their wallet pub key/address, "pong" and a signature
    req_encoded := json.encode(req)

    raw := http.post("$ref/handshake", req_encoded) or {
        eprintln("Failed to shake hands with $ref, Node is probably offline. Error: $raw")
        return false
    }

    if raw.status_code != 200 {
        eprintln("Failed to shake hands with $ref, may have sent incorrect data, responder is probably offline")
        return false
    }

    data := json.decode(HandshakeResponse, raw.body) or {
        eprintln("Failed to decode handshake response, responder is probably using an old version of the protocol")
        return false
    }

    println("\n$ref responded to handshake.")

    // signed hash can then be verified using the wallet pub key supplied
    if data.message == msg && data.initiator_key == this.self.key {
        if cryptography.verify(data.responder_key, data.message.bytes(), data.signature) {
            println("Verified signature to match handshake key\nHandshake with $ref is valid")
            return true
        }
        println("Signature did not match handshake key, node is not who they claim to be")
        // this is where we would then store a record of the node's reference/ip address
        return false
    }

    println("Handshake was not valid, node is not who they claim to be.")
    println(data)
    // if valid, return true, if not return false
    return false
}

```

This version of the code can be found [here](#).

Challenges

The main challenge faced in this cycle was surrounding responding to the handshake requests, as in a future cycle I plan to complete a check after a node receives a request to check if they have seen the requester before and if not send them their own request to get info on them.

This involved planning for the future and examining how to get the reference of the node that sent a request, however my initial plan, which was just to examine the http headers failed because due to how my primary node (<https://nano.monochain.network>) uses cloud-flare tunnelling to increase security so all nodes that send requests appear to have the same ip -> one of cloud-flare's servers. Therefore if I was having that problem then other people probably would as well so after some testing and probing I decided not to try and cover that in this cycle and it will instead be covered in a future cycle.

Testing

Tests

Test	Instructions	What I expect	What happened
1	Completing a correct and valid handshake using a time object as the message.	A series of console logs on both nodes confirming that the handshake was successful.	As expected
2	Completing an invalid handshake using the string "invalid data" as the message.	For both nodes to record the handshake as unsuccessful and log such to the console.	As expected

Evidence

Test 1 Evidence - Valid data in a valid handshake

```
***** MonoChain Mining Software *****
Reading file: ./monochain/node.config

Config Loaded...
[Vweb] Running app on http://localhost:8000/

Sending ping request to https://nano.monochain.network.
Message: 2022-08-31 16:53:45.043892

https://nano.monochain.network responded to ping request.
Verified signature to match pong key
Handshake with https://nano.monochain.network successful.
```

The handshake initiator

```
Config Loaded...
Reading file: ./monochain/keys.config
Keys loaded from file.
Received pong request from node with public key: [0xa1, 0xe9, 0xf7, *, $, P, 0x80, 0xc6, 0xb5
, U, M, 0xf7, `e, 0xeb, 0x8f, /, 0xa0, 0xf3, `t, `t, 0x9c, _, B, 0xea, 0xd7, c, R, N, X
, 0xef, 0x9a, 0xa7]
Time parsed correctly as: 2022-08-31 16:53:45
```

The handshake Recipient

As shown, both the initiator and recipient successfully agreed on the handshake, with both showing the same time message to prove this was the same handshake as the test.

Test 2 Evidence - Invalid data in the handshake

```
***** MonoChain Mining Software *****
Reading file: ./monochain/node.config

Config Loaded...
[Vweb] Running app on http://localhost:8000/

Sending ping request to https://nano.monochain.network.
Message: invalid data
Failed to ping https://nano.monochain.network, may have sent incorrect data, repsonse body: 500 Internal Server Error
```

The handshake initiator

```
Config Loaded...
Reading file: ./monochain/keys.config
Keys loaded from file.
Received pong request from node with public key: [0xa1, 0xe9, 0xf7, *, $, P, 0x80, 0xc6, 0xb5
, U, M, 0xf7, `e, 0xeb, 0x8f, /, 0xa0, 0xf3, `t, `t, 0x9c, _, B, 0xea, 0xd7, c, R, N, X
, 0xef, 0x9a, 0xa7]
Incorrect time format supplied to /pong/:req by node claiming to be [0xa1, 0xe9, 0xf7, *, $,
P, 0x80, 0xc6, 0xb5, U, M, 0xf7, `e, 0xeb, 0x8f, /, 0xa0, 0xf3, `t, `t, 0x9c, _, B, 0xe
a, 0xd7, c, R, N, X, 0xef, 0x9a, 0xa7]
```

The handshake Recipient

As shown both nodes classified the handshake as failed, with the initiator predicting that incorrect data may have been sent (although that is not guaranteed as another error may have occurred but in this case we know it was the case of invalid data); and the Recipient logging an incorrect time format supplied and who it was claimed to be supplied by.

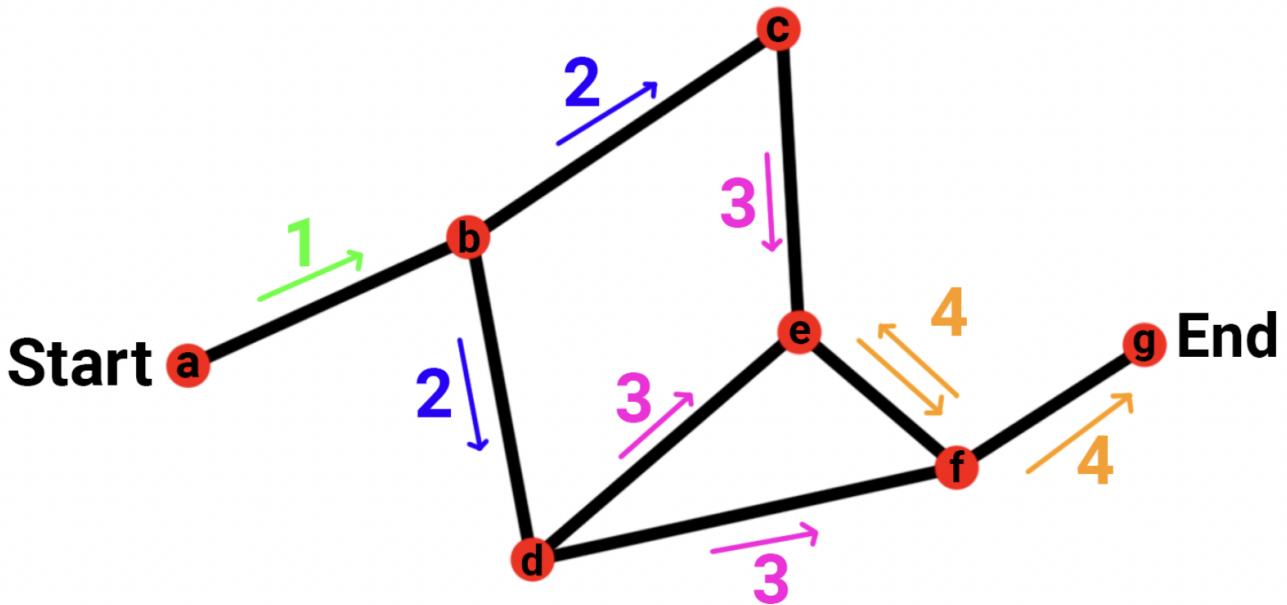
2.2.11 Cycle 11 - Remembering Nodes

Design

Objectives

The next step to building a network in which nodes can successfully communicate across is for the node software to remember which nodes it has come into contact with previously so that it can send messages it receives from other nodes onto its remembered nodes in the future.

This is what will create the networking effect that allows for messages to flow throughout the entire network without having to have any form of central server.



A diagram showing how communication across the network works.

The diagram above shows how such a message flow would work, with node 'a' wanting to send a message across the network which will end up with every node having read it, including node 'g' at the opposite side. The way in which this will work can be explained using a few stages:

1. Node 'a' sends the message to all nodes it's aware of, in this case only node 'b', this is represented by green arrow and number.
2. Node 'b' receives the message, checks it's valid, approves it and then forwards it on to all the nodes it knows (apart from the node it just came from - 'a'), this is blue.
3. Nodes 'c' and 'd' receive the message from 'b', validate it and then forward it on to the nodes they know of, this is pink.
4. Nodes 'e' and 'f' both receive the message, validate it and then begin to send it on to any node they know of that didn't send them the message. Although in this case they don't know that the other node has already seen the message so they send it on anyway. Since node 'f' is also connected to node 'g' this does mean that 'g' is also sent the message. This is represented by the colour orange.
5. All nodes have now seen the message and therefore the cycle is complete.

Therefore the primary objectives this cycle are:

- Nodes should store references to other Nodes.
- Nodes should send handshakes to nodes upon connecting to the network.
- Nodes should send a handshake back to any node that sends them one that they haven't seen before.

Usability Features

- Storing the node data locally so that nodes will only send messages to a smaller pool of nodes rather than to every node on large, publically accessible lists speeds the network up and limits the number of duplicate messages.
- Once the network reaches the point at which it is large enough that every node knows atleast two other nodes, we can be sure that if any one node shuts down or goes offline any messages will still be distributed across the network.

Key Variables

Variable Name	Use
References	This is the references object which holds all the known references that a node has encountered as well as some functionality to check if the node is aware of a specific reference, add a new reference, etc.
ref_path	This is the file path of the file at which the references object is stored and loaded from. This is stored as a parameter within the configuration object.

Pseudocode

Handling References

This piece of code handles the initial definition of the 'References' object and the loading, saving generation of that object. Like most of the rest of the key objects in the codebase the object is saved and loaded in json.

```

IMPORT utils
IMPORT json

// This is a simple way of storing the node's memory of other nodes that it's encountered.

OBJECT References:
    path
    keys // this maps a reference the pub key that it runs using.
    blacklist // this is a list of pub keys that we've already seen and do not trust

    FUNCTION save(this):
        // "this" refers to the object itself.
        // create a new object to ignore blacklisted keys
        raw := json.encode({
            path: this.path
            keys: this.keys
        })
        utils.save_file(this.path, raw, 0)
    END FUNCTION

END OBJECT

FUNCTION new(file_path):
    ref = NEW References{
        path: file_path
        keys: {},
        blacklist: {}
    }

    ref.save()
    RETURN ref
END FUNCTION

FUNCTION get_refs(file_path):
    raw = utils.read_file(file_path)
    refs = new(file_path) // incase no file or error

    IF (raw.loaded != false):
        TRY:
            // convert the json data to a References struct.
            refs = json.decode(References, raw.data)
        CATCH:
            // if the json is invalid, create a new one.
            refs = new(file_path)
        END TRY
    END IF

    RETURN refs
END FUNCTION

```

Using the Reference object

The functions in this block of code allow for other parts of the codebase to add keys to both the standard reference and the blacklist reference simply by supplying the reference (and key for the main reference type). Then also to check if the references object is already aware of a specified reference for use in areas such as the handshake code expanded in [Cycle 10](#).

```
EXTEND OBJECT References:  
    FUNCTION aware_of(this, reference):  
        // check if the reference is in the blacklist.  
        IF (refs.blacklist[reference]):  
            // we have encountered this reference before and it is blacklisted.  
            RETURN true  
        END IF  
  
        // if it is not blacklisted, check if the reference is in the keys.  
        IF (reference in refs.keys):  
            // we have encountered this reference before and it is not blacklisted.  
            RETURN true  
        END IF  
  
        // we have not encountered this reference before.  
        RETURN false  
    END FUNCTION  
  
    FUNCTION add_key(this, reference, key) {  
        // add the key to the keys map.  
        this.keys[reference] = key  
        // save the references.  
        this.save()  
    END FUNCTION  
  
    FUNCTION add_blacklist(this, reference) {  
        // add the reference to the blacklist.  
        this.blacklist[reference] = true  
        // save the references.  
        this.save()  
    END FUNCTION  
END OBJECT EXTEND
```

Using references when receiving a handshake.

This is a piece of code taken from the handshake responder api route that checks if the node has come into contact with the node sending a handshake request before and if not, starts a new handshake to 'get to know' the new node.

```

OUTPUT "Handshake Analysis Complete. Sending response..."  

// now need to figure out where message came from and respond back to it  
  

refs = memory.get_refs(config.ref_path)  

IF (NOT refs.aware_of(req_parsed.initiator.ref)):  

    OUTPUT "Node has not come into contact with initiator before, sending them a handshake request."  

    // send a handshake request to the node  

    start_handshake(req_parsed.initiator.ref, config)  

ELSE:  

    OUTPUT "Node has come into contact with initiator before, no need to send a handshake request."  

END IF

```

Using references when sending a handshake.

This allows the node to remember if a node successfully completed a handshake request - in which case it is added to the main reference map - or unsuccessfully didn't complete the request - in which case the reference is temporarily blacklisted until the node restarts.

```

refs = memory.get_refs(config.ref_path)  
  

// this verifies that the received handshake is valid  

// signed hash can then be verified using the wallet pub key supplied  

IF (data.message == msg AND data.initiator.key == this.self.key):  

    IF (cryptography.verify(data.responder_key, data.message.bytes(), data.signature)):  

        // handshake was valid.  

        OUTPUT "Verified signature to match handshake key\nHandshake with $ref successful."  
  

        // now add them to reference list  

        refs.add_key(ref, data.responder_key)  

        RETURN true  

    END IF  
  

    // handshake signature was not valid  

    OUTPUT "Signature did not match handshake key, node is not who they claim to be."  
  

    // store a record of the node's reference and temporarily blacklist it  

    refs.add_blacklist(ref)  

    RETURN false  

END IF  
  

// handshake message was not assembled correctly.  

OUTPUT "Handshake was not valid, node is not who they claim to be."  

OUTPUT data  
  

// node is not who they claim to be, so temporarily blacklist it  

refs.add_blacklist(ref)

```

Development

Although the changes introduced in this cycle were fairly simple to do and didn't take many changes to the rest of the program to start being used properly, they are still very substantial, as the addition of referencing at memorising which nodes a node has come into contact with before is what will allow the network to truly grow and become as interconnected as it will need to be to work properly.

Outcome

Handling References

```
module memory

// internal imports

import utils

// external imports

import json

// This is a simple way of storing the node's memory of other nodes that it's encountered.

pub struct References {
    path string
    mut:
        keys map[string][]u8 // this maps a reference the pub key that it runs using
        blacklist map[string]bool // this is a list of pub keys that we've already seen
}

fn (Ref References) save() {
    // create a new object to ignore blacklisted keys
    raw := json.encode(References{
        path: Ref.path
        keys: Ref.keys
    })
    utils.save_file(Ref.path, raw, 0)
}

fn new(file_path string) References {
    ref := References{
        path: file_path
        keys: map[string][]u8{},
        blacklist: map[string]bool{},
    }
}
```

```
    }

    ref.save()

    return ref
}

pub fn get_refs(file_path string) References {
    raw := utils.read_file(file_path, true)

    mut refs := new(file_path) // incase no file or error

    if raw.loaded != false {
        // convert the json data to a References struct.

        refs = json.decode(References, raw.data) or {
            // if the json is invalid, create a new one.

            new(file_path)
        }
    }

    return refs
}
```

Using the Reference object

```

pub fn (refs References) aware_of(reference string) bool {
    // check if the reference is in the blacklist.
    if refs.blacklist[reference] {
        // we have encountered this reference before and it is blacklisted.
        return true
    }

    // if it is not blacklisted, check if the reference is in the keys.
    if reference in refs.keys {
        // we have encountered this reference before and it is not blacklisted.
        return true
    }

    // we have not encountered this reference before.
    return false
}

pub fn (mut refs References) add_key(reference string, key []u8) {
    // add the key to the keys map.
    refs.keys[reference] = key
    // save the references.
    refs.save()
}

pub fn (mut refs References) add_blacklist(reference string) {
    // add the reference to the blacklist.
    refs.blacklist[reference] = true
    // save the references.
    refs.save()
}

```

Using references when receiving a handshake.

```

println("Handshake Analysis Complete. Sending response...")
// now need to figure out where message came from and respond back to it
refs := memory.get_refs(config.ref_path)
if !refs.aware_of(req_parsed.initiator.ref) {
    println("Node has not come into contact with initiator before, sending them a handshake")
    // send a handshake request to the node
    start_handshake(req_parsed.initiator.ref, config)
} else {
    println("Node has come into contact with initiator before, no need to send a handshake")
}

```

Using references when sending a handshake.

```

mut refs := memory.get_refs(config.ref_path)

// this verifies that the received handshake is valid
// signed hash can then be verified using the wallet pub key supplied
if data.message == msg && data.initiator.key == this.self.key {
    if cryptography.verify(data.responder_key, data.message.bytes(), data.signature) {
        // handshake was valid.
        println("Verified signature to match handshake key\nHandshake with $ref succe:

        // now add them to reference list
        refs.add_key(ref, data.responder_key)
        return true
    }

    // handshake signature was not valid
    println("Signature did not match handshake key, node is not who they claim to be.")
    // store a record of the node's reference and temporarily blacklist it
    refs.add_blacklist(ref)
    return false
}

// handshake message was not assembled correctly.
println("Handshake was not valid, node is not who they claim to be.")
println(data)
// node is not who they claim to be, so temporarily blacklist it
refs.add_blacklist(ref)

```

Challenges

Since the actual logic for this part of the software is somewhat simple, this cycle didn't have many algorithmic based challenges but that doesn't mean it didn't have its hurdles. In this case the main hurdle was trying to figure out what the reference for a node was and initially I trialled using ip tracking and cryptography signatures to attach a secure method of ensuring that a node was who they said they were. However this didn't work out as the ip tracking wasn't always perfect and this lead to the other testing servers running this version of the node software semi-randomly blacklisting each other at various points.

The solution to this was to realise that as long as any secure requests/responses contain a signature validating that transmission, the node's reference doesn't really need to be secure and can mainly just be used to remember where nodes are on the internet and then anything else can be done using signed messages to and from the referenced node. This means that if a node is lying about its reference, it won't matter as it will still be signing the messages with a different

Testing

Test Table

Tests

Test	Instructions	What I expect
1	Create a new references object and log it to the console.	An empty references object logged in the console.
2	Create a new references object and log it to the console using the "get_refs" function.	An empty references object logged in the console and created with that object.
3	Add a test reference to the references object and save it to a test file.	A references object with the reference to be saved to the test file.
4	Load the test references object created in test 3 and check if the node is still aware of that node using the "aware_of" function.	The references object to be aware of the test reference.
5	Add a test reference to the references object as a blacklisted node and save it to a test file.	A references object with the reference as a blacklisted node saved to the test file.
6	Load the test references object created in test 3 and check if the node is still aware of that node using the "aware_of" function.	The references object to no longer be aware of the test reference.

Test Code

Test 1

```
const test_path = "./test.json"

pub fn test_new_ref_obj() {
    println("\n\n[Memory Test] Testing the creation of a new Reference Object.")
    obj := new(test_path)
    println("[Memory Test] Reference Object created: $obj")
    assert true
}
```

Test 2

```
pub fn test_get_refs() {
    println("\n\n[Memory Test] Testing the retrieval of references.")
    refs := get_refs(test_path)
    println("[Memory Test] References retrieved: $refs")
    assert true
}
```

Test 3

```
pub fn test_add_ref() {
    println("\n\n[Memory Test] Testing the addition of a reference.")
    mut refs := get_refs(test_path)
    println("[Memory Test] Reference object loaded.")
    refs.add_key("test", []u8{})
    println("[Memory Test] Reference added: $refs")
    assert true
}
```

Test 4

```
pub fn test_aware_of() {
    println("\n\n[Memory Test] Testing the awareness of a reference.")
    refs := get_refs(test_path)
    println("[Memory Test] Reference object loaded.")
    aware := refs.aware_of("test")
    println("[Memory Test] Reference awareness: $aware")
    assert aware
}
```

Test 5

```
pub fn test_add_blacklisted_ref() {
    println("\n\n[Memory Test] Testing the addition of a blacklisted reference.")
    mut refs := get_refs(test_path)
    println("[Memory Test] Reference object loaded.")
    refs.add_blacklist("test")
    println("[Memory Test] Reference added: $refs")
    assert true
}
```

Test 6

```
pub fn test_aware_of_blacklisted() {
    println("\n\n[Memory Test] Testing the awareness of a blacklisted reference")
    refs := get_refs(test_path)
    println("[Memory Test] Reference object loaded.")
    aware := refs.aware_of("test")
    println("[Memory Test] Reference awareness: $aware")
    assert aware == false
}
```

Evidence

Test 1

```
[Memory Test] Testing the creation of a new Reference Object.
[Memory Test] Reference Object created: memory.References{
    path: './test.json'
    keys: {}
    blacklist: {}
}
OK      [1/6]      0.391 ms      1 assert | memory.test_new_ref_obj()
```

Test 2

```
[Memory Test] Testing the retrieval of references.
[utils] Reading file: ./test.json
[Memory Test] References retrieved: memory.References{
    path: './test.json'
    keys: {}
    blacklist: {}
}
OK      [2/6]      0.215 ms      1 assert | memory.test_get.refs()
```

Test 3

```
[Memory Test] Testing the addition of a reference.
[utils] Reading file: ./test.json
[Memory Test] Reference object loaded.
[Memory Test] Reference added: memory.References{
    path: './test.json'
    keys: {'test': []}
    blacklist: {}
}
OK      [3/6]      0.288 ms      1 assert | memory.test_add_ref()
```

Test 4

```
[Memory Test] Testing the awareness of a reference.  
[utils] Reading file: ./test.json  
[Memory Test] Reference object loaded.  
[Memory Test] Reference awareness: true  
OK [4/6] 0.157 ms 1 assert | memory.test_aware_of()
```

Test 5

```
[Memory Test] Testing the addition of a blacklisted reference.  
[utils] Reading file: ./test.json  
[Memory Test] Reference object loaded.  
[Memory Test] Reference added: memory.References{  
    path: './test.json'  
    keys: {}  
    blacklist: {'test': true}  
}  
OK [5/6] 0.487 ms 1 assert | memory.test_add_blacklisted_ref()
```

Test 6

```
[Memory Test] Testing the awareness of a blacklisted reference.  
[utils] Reading file: ./test.json  
[Memory Test] Reference object loaded.  
[Memory Test] Reference awareness: false  
OK [6/6] 0.182 ms 1 assert | memory.test_aware_of_blacklisted()  
Summary for running V tests in "refs_test.v": 6 passed, 6 total. Runtime: 1 ms.
```

2.2.12 Cycle 12 - Sending messages across the network

Design

Objectives

Since the project is getting near the end of the time period allowed for development and the validation/transaction part of the blockchain doesn't exist outside of the theoretical space yet, but the majority communication layer is nearly finished I think it would be a good idea to get general communication working such that by the end of the allowed period of development the project exists in a form that is somewhat usable.

This form would effectively end up being a decentralised messaging service, which although is not quite the full blockchain that I was aiming for in the first place, still utilises a lot of the same networking and logic - due to effectively being a demo of the decentralised communication layer.

To do this, I will need to introduce a few new features that will build upon what I have created in previous cycles:

- Generating message objects that state who created the object and provides a signature to prove it.
- Sending these messages to all references stored on the node.
- Receiving, validating, logging and forwarding these messages.

With these features the messages being sent will be very similar to the transactions that were initially hypothesised in the analysis of this project, however instead of being sent in bulk during block creation, will instead be sent individually whenever nodes feel like it.

This will then allow for the adaption of these messages into blocks in a future cycle if there is enough time to continue on this path, meaning that in any form this is not a detour of the theorised final project, just a way to create a demo that uses everything that has been created so far.

Usability Features

- Receiving messages - Only store valid messages so as to prevent the waste of storage on a user's computer that comes with storing invalid messages.
- Logging - Continuous logging should be made while the program operates so that the user can check what is going on with their node by checking the terminal output/logs.

Key Variables

Variable Name	Use
Broadcast_Message_Contents	This object holds a message received or created for broadcasting, it holds all necessary data such as the sender, timestamp, contents, etc. It's useful for ensuring that messages broadcasted across the message are all consistently typed.
Broadcast_Message	This object wraps the <code>Broadcast_Message_Contents</code> object and contains a signature that confirms that the sender did in fact send and approve a message.

Pseudocode

The API endpoint

This is the endpoint at which other nodes will send their own messages to be broadcasted onwards, although it also includes some logic to ensure that the messages received are indeed valid before the node will send them onwards.

```

// internal imports
IMPORT database
IMPORT cryptography
IMPORT configuration

// external imports
IMPORT json
IMPORT vweb
IMPORT time
IMPORT net.http

//objects
OBJECT Broadcast_Message_Contents {
    sender
    receiver
    data
    time
}

OBJECT Broadcast_Message {
    message
    signature
}

EXTENT OBJECT Web_Server:
    // setup the api route as using the POST http method.
    FUNCTION broadcast_route(this) METHOD POST:
        db := this.db
        body := this.req.data
        decoded := {}

        TRY:
            decoded = json.decode(Broadcast_Message, body)
        CATCH:
            OUTPUT "[Broadcaster] Message received that cannot be decoded: $body"
            RETURN app.server_error(403)
        END TRY

        valid_message = cryptography.verify(decoded.message.sender, json.encode(decoded.message.signature))

        IF (valid_message):
            OUTPUT "[Broadcaster] Message received from $decoded.message.sender is valid"

            // check if message has been received before
            parsed_signature = decoded.signature.str()
            parsed_sender = decoded.message.sender.str()
            parsed_receiver = decoded.message.receiver.str()

            // check if message has been received before
            message_seen_before = db.get_message(parsed_signature, parsed_sender, parsed_receiver)

            IF (!message_seen_before):

```

```

        OUTPUT "[Broadcaster] Have not seen message before.\n[Broadcas

        message_db = {
            timestamp: decoded.message.time
            contents: decoded.message.data
            sender: parsed_sender
            receiver: parsed_receiver
            signature: parsed_signature
        }

        // save to database/file system/etc
        SAVE message_db

        OUTPUT "[Database] Saved message to database."

        OUTPUT "\n[Broadcaster] Received message:\n[Broadcaster] Sender
forward_to_all(db, decoded)
RETURN this.ok("Message received and forwarded.")

END IF

        OUTPUT "[Broadcaster] Have seen message before."
RETURN this.ok("Message received but not forwarded.")

ELSE:
        OUTPUT "[Broadcaster] Received an invalid message"
RETURN this.server_error(403)
END IF

        OUTPUT "[Broadcaster] Shouldn't have reached this part of the code - please re
RETURN app.server_error(403)
END FUNCTION
END OBJECT EXTEND

```

Functionality for internal use

These are the functions that will allow the node to handle sending messages internally and these functions will be called by other sections - such as the dashboard page also being developed in this cycle.

```

FUNCTION forward_to_all(db, msg):
    OUTPUT "[Broadcaster] Sending message to all known nodes."

    // get all known nodes - file/db/etc
    refs = GET ALL node_references

    FOR (ref IN refs):
        // start the send function on a new thread
        GO send(ref.domain, ref.ws, msg)

    END FOR

    OUTPUT "[Broadcaster] Sent message to all known nodes."
END FUNCTION

FUNCTION send(ref string, ws bool, msg Broadcast_Message) bool {
    println("[Broadcaster] Attempting to send message to $ref")
    IF (ws) {
        OUTPUT "[Broadcaster] Websockets are not implemented yet, cannot send message"
        RETURN false
    END IF

    raw_response = {}

    TRY:
        raw_response = http.post("$ref/broadcast", json.encode(msg))
    CATCH:
        OUTPUT "[Broadcaster] Failed to send a message to $ref, Node is probably offl:
        RETURN false
    END TRY

    IF (raw_response.status_code != 200):
        OUTPUT "[Broadcaster] $ref responded to message with an error. Code: $raw_res
        RETURN false
    END IF

    OUTPUT "[Broadcaster] Successfully Sent message to $ref"
    return true
}

```

```

END FUNCTION

FUNCTION broadcast_message(db database.DatabaseConnection, data string){
    send_message(db, data, "".bytes())
}

END FUNCTION

FUNCTION send_message(db, data, receiver):
    OUTPUT "[Broadcaster] Assembling message with data: $data"
    config = configuration.get_config()
    keys = cryptography.get_keys(config.key_path)

    contents = Broadcast_Message_Contents{
        sender: config.self.key
        receiver: receiver
        time: time.now().str()
        data: data
    }

    message = Broadcast_Message{
        signature: keys.sign(json.encode(contents).bytes())
        message: contents
    }

    OUTPUT "[Database] Saving message to database."
    db_msg = {
        timestamp: contents.time
        sender: contents.sender.str()
        receiver: contents.receiver.str()
        contents: contents.data
        signature: message.signature.str()
    }

    // save to a file/database/etc
    SAVE db_msg

    OUTPUT "[Database] Message saved."

```

```
OUTPUT "[Broadcaster] Message assembled, broadcasting to refs..."  
forward_to_all(db, message)  
END FUNCTION
```

Development

An addition that I ended up creating for this cycle that wasn't originally planned but was very useful to test and use the functionality created was a basic html page that was hosted on the http server to allow a user to send and view messages in an easy to use, simple web page.

Paired with all the new broadcasting functionality that allows messages to be shared across the network and the development for this cycle ends up being a lot more substantial than the pseudocode would make it appear, hence the outcome section simply includes two relevant files in their entirety as they were both created during this cycle and both contain relevant code.

The first being the broadcast file that shows all the code written to send messages across the network and forward those that have been received and checked to be valid.

Then the second being the user sided dashboard that helps to provide a nice way to view and contribute to these messages.

Outcome

- ⓘ Due to the majority of the changes in this cycle taking place whilst I was also working on the development of the database addition of the cycle after this one, **the code shown below includes references to an early version of the database module** but would have the same primary logic if it was using the old referencing style.

The broadcast file

This file is split up into three key sections: the setup; the api endpoints; and the internal functions; this is because although it would be briefer to just show the api endpoints and the internal functions, the setup helps to give more context to those two sections so I included it anyway.

Setup

File setup

This shows the setup of the file where various internal and external modules are imported and then two structs used for message generation are setup for use in the sections below.

```
module server

// internal imports
import database
import cryptography
import configuration

// external imports
import json
import vweb
import time
import net.http

//structs
struct Broadcast_Message_Contents {
    sender  []u8
    receiver []u8
    data    string
    time    string
}

struct Broadcast_Message {
    message Broadcast_Message_Contents
    signature []u8
}
```

Api Endpoint

The api endpoint

This is the endpoint at which other nodes will send their own messages to be broadcasted onwards, although it also includes some logic to ensure that the messages received are indeed valid before the node will send them onwards.

```
['/broadcast'; post]
pub fn (mut app App) broadcast_route() vweb.Result {
    db := app.db
```

```

body := app.req.data
decoded := json.decode(Broadcast_Message, body) or {
    eprintln("[Broadcaster] Message received that cannot be decoded: $b
        return app.server_error(403)
}

valid_message := cryptography.verify(decoded.message.sender, json.encode(de

if valid_message {
    println("[Broadcaster] Message received from $decoded.message.send
    // check if message has been received before
    parsed_signature := decoded.signature.str()
    parsed_sender := decoded.message.sender.str()
    parsed_receiver := decoded.message.receiver.str()

    // check if message has been received before
    message_seen_before := db.get_message(parsed_signature, parsed_send

    if !message_seen_before {
        println("[Broadcaster] Have not seen message before.\n[Broadc

        message_db := database.Message_Table{
            timestamp: decoded.message.time
            contents: decoded.message.data
            sender: parsed_sender
            receiver: parsed_receiver
            signature: parsed_signature
        }

        sql db.connection {
            insert message_db into database.Message_Table
        }
        println("[Database] Saved message to database.")

        println("\n[Broadcaster] Received message:\n[Broadcaster] S
        forward_to_all(db, decoded)
        return app.ok("Message received and forwarded.")
    }

    println("[Broadcaster] Have seen message before.")
    return app.ok("Message received but not forwarded.")
}

} else {
    eprintln("[Broadcaster] Received an invalid message")
    return app.server_error(403)
}

println("[Broadcaster] Shouldn't have reached this part of the code - pleas
return app.server_error(403)
}

```

Functionality for internal use.

These are the functions that will allow the node to handle sending messages internally and these functions will be called by other sections - such as the dashboard page also being developed in this cycle.

```

pub fn forward_to_all(db database.DatabaseConnection, msg Broadcast_Message) {
    println("[Broadcaster] Sending message to all known nodes.")

    // get all known nodes
    refs := sql db.connection {
        select from database.Reference_Table
    }

    for ref in refs {
        go send(ref.domain, ref.ws, msg)
    }

    println("[Broadcaster] Sent message to all known nodes.")
}

pub fn send(ref string, ws bool, msg Broadcast_Message) bool {
    println("[Broadcaster] Attempting to send message to $ref")
    if ws {
        eprintln("[Broadcaster] Websockets are not implemented yet, cannot
        return false
    }

    raw_response := http.post("$ref/broadcast", json.encode(msg)) or {
        eprintln("[Broadcaster] Failed to send a message to $ref, Node is p
        return false
    }

    if raw_response.status_code != 200 {
        eprintln("[Broadcaster] $ref responded to message with an error. Co
        return false
    }

    println("[Broadcaster] Successfully Sent message to $ref")
    return true
}

pub fn broadcast_message(db database.DatabaseConnection, data string){
    send_message(db, data, "" .bytes())
}

pub fn send_message(db database.DatabaseConnection, data string, receiver []u8) {

```

```

    println("[Broadcaster] Assembling message with data: $data")
    config := configuration.get_config()
    keys := cryptography.get_keys(config.key_path)

    contents := Broadcast_Message_Contents{
        sender: config.self.key
        receiver: receiver
        time: time.now().str()
        data: data
    }

    message := Broadcast_Message{
        signature: keys.sign(json.encode(contents).bytes())
        message: contents
    }

    println("[Database] Saving message to database.")
    db_msg := database.Message_Table{
        timestamp: contents.time
        sender: contents.sender.str()
        receiver: contents.receiver.str()
        contents: contents.data
        signature: message.signature.str()
    }

    sql db.connection {
        insert db_msg into database.Message_Table
    }
    println("[Database] Message saved.")

}

println("[Broadcaster] Message assembled, broadcasting to refs...")
forward_to_all(db, message)
}

```

The dashboard file

This file houses all the logic and code for the http dashboard that is hosted the same way that the api endpoints are, with a basic token system to allow a node's 'owner' to send messages from anywhere that can access the node's api server

Setup

File setup

This just shows the setup for modules and constants needed by other parts of the code and contains no actual logic.

```
module server

// external modules
import vweb
import crypto.rand
import json
import net.http

// internal modules
import configuration
import utils

// constants
const token_path = "$configuration.base_path/tokens.json"

// structs
struct MessageInfo {
    pub:
        sender string
        timestamp string
        contents string
}
```

Http Pages

Http endpoints

Both of these endpoints reference http pages to be hosted by the api server, I have decided not to include the html files because they are both quite a lot of lines due to housing html, javascript and css code all within the same file, but the files can be found [here](#).

```
['/dashboard'; get]
pub fn (mut app App) dashboard_page() vweb.Result {
    if !is_logged_in(app) {
        return app.redirect('/login')
    }

    raw_message_objs := app.db.get_latest_messages(0, 25)
```

```

mut message_objs := []MessageInfo{}

for msg in raw_message_objs {
    mut sender := ""

    data := json.decode([]u8, msg.sender) or {
        // eprintln("Error decoding JSON of sender: $err")
        sender = msg.sender
        []u8{}
    }

    for item in data {
        // Big O notation of O(n^2) so not great, but should be a s
        sender += item.hex()
    }

    message_objs << MessageInfo{
        sender: sender
        timestamp: msg.timestamp
        contents: msg.contents
    }
}

return $vweb.html()
}

['/login'; get]
pub fn (mut app App) login_page() vweb.Result {
    if is_logged_in(app) {
        return app.redirect('/dashboard')
    }
    return $vweb.html()
}

```

Api Endpoints

Api endpoints

All of the endpoints in this section are accessed by html requests within code and are not meant to be accessed directly by users, and handle the token logic to provide a somewhat secure method of allowing a node's owner to talk to the node over the internet. This is not the most secure way this could've been done but should be fine by now.

```

['/dashboard/gentoken'; get]
pub fn (mut app App) gentoken_route() vweb.Result {
    config := configuration.get_config()
    // Generate a new token

```

```

token := (rand.int_u64(4294967295) or { return app.server_error(500) }).str
cur_tokens := utils.read_file(token_path, true)

if !cur_tokens.loaded {
    if !utils.save_file(token_path, json.encode([token]), 0) {
        eprintln("[Server] Failed to save token to file.")
    }
} else {
    mut prev_tokens := json.decode([]string, cur_tokens.data) or {
        eprintln("[Server] Failed to load previous tokens, overwriting")
        []string{}
    }
    prev_tokens << token

    if !utils.save_file(token_path, json.encode(prev_tokens), 0) {
        eprintln("[Server] Failed to save token to file.")
    }
}

println("\n\n[Server] Generated new token: $token\n[Server] Enter it at http://$host:$port/api/login")

// Confirm the token has been sent
return app.ok("ok")
}

['/dashboard/login'; post]
pub fn (mut app App) login_route() vweb.Result {
    data := app.Context.req.data

    prev_tokens_raw := utils.read_file(token_path, true)

    if !prev_tokens_raw.loaded {
        eprintln("[Server] Failed to load token storage.")
        return app.server_error(401)
    }

    prev_tokens := json.decode([]string, prev_tokens_raw.data) or {
        eprintln("[Server] Failed to decode token data.")
        return app.server_error(401)
    }

    if data in prev_tokens {
        println("[Server] Token supplied valid.")
        cookie := http.Cookie{
            name: "token"
            value: data
            max_age: 3600
            secure: false
        }

        app.set_cookie(cookie)
        return app.ok("ok")
    }
}

```

```

eprintln("[Server] Token supplied not valid.")
println("[Server] Supplied data $data, tokens available: $prev_tokens")
return app.server_error(401)
}

['/dashboard/send_message'; post]
pub fn (mut app App) send_message_route() vweb.Result {
    if !is_logged_in(app) {
        return app.server_error(401)
    }

    data := app.Context.req.data
    send_message(app.db, data, []u8{})
    return app.ok("ok")
}

```

Internal Functions

Internal functionality

The only function in this section just allows the various api endpoints to check if a user has a valid login token and responds with a boolean value to tell the requesting function whether or not they are logged in or not.

```

fn is_logged_in(app App) bool {
    cookie := app.get_cookie('token') or {
        eprintln("[Server] Failed to get cookie. Error: $err")
        return false
    }

    if cookie == '' {
        return false
    }

    prev_tokens_raw := utils.read_file(token_path, true)

    if !prev_tokens_raw.loaded {
        eprintln("[Server] Failed to load token storage.")
        return false
    }

    prev_tokens := json.decode([]string, prev_tokens_raw.data) or {
        eprintln("[Server] Failed to decode token data.")
        return false
    }

    if cookie in prev_tokens {
        println("[Server] Token supplied valid.")
    }
}

```

```
// now get the node's data and do stuff with it
return true
}

eprintln("[Server] Token supplied not valid.")
println("[Server] Supplied data $cookie, tokens available: $prev_tokens")
return false
}
```

Challenges

One of the key challenges this cycle was to figure out how to give access to the dashboard to the node's owner and not to other users without having to make them send any private data over a network as it is fairly probable that the software will be running without https as it isn't required for any other part of the software.

The method I created to handle this is very simple but should be secure enough for now and simply relies upon the user trying to login having access to the terminal output of the software at login time.

This is because the security method is based upon a basic cookie and token system where the user clicks "get a token" to generate a new token that is then outputted in the software's terminal output, they can then copy this token into the input field to login where it is submitted as a form of password. Assuming this token is correct, the user's browser is then sent a cookie holding this token so that any other request sent by the user contains this token and verifies that they have control of this node.

Testing

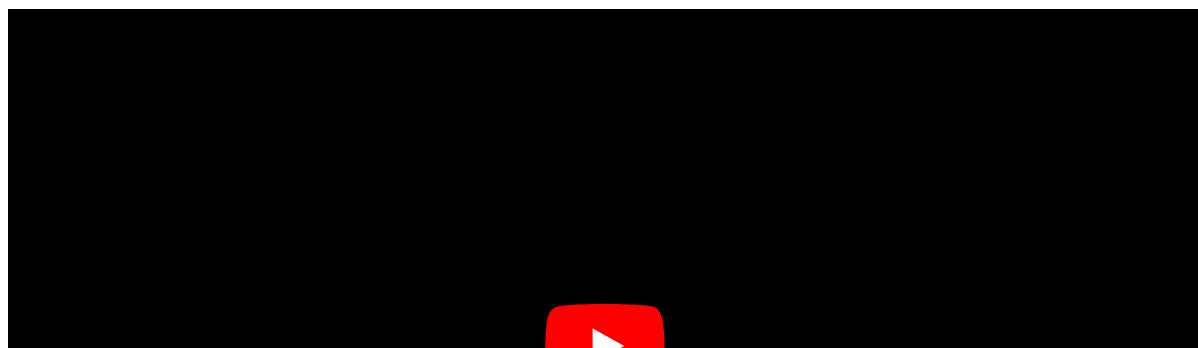
Test Table

Tests

Test	Instructions	What I expect
1	Navigate to 'https://localhost:8000/dashboard'	The dashboard page should show the login page with an option to input a token or generate one.
2	Click the 'generate token' button	A token to be output in the terminal output.
3	Enter an invalid token into the input field and click 'submit'	The dashboard should not grant user access.
4	Enter an valid token into the input field and click 'submit'	The dashboard should give access to itself.
5	Type a message and click send.	The node should receive the message from the dashboard and attempt to send it across the network.
6	Connect to and then send a message to another node using the dashboard.	The node should connect to an existing node and then send a message to it after a message is received from the dashboard.
7	Send a message to the node from another node.	The node on the recipient node should receive the same message as was sent from the other node.

Evidence

Evidence





2.2.13 Cycle 13 - Introducing a sql database

Design

Objectives

Now that you can send messages between nodes, it's time to introduce a way of storing these messages and the best way in which I can think of that to be done is by creating a local database such that the node can store and load data onto that database and use it as its memory. The benefit of introducing a database for this reason also means that the references that were introduced in [Cycle 11 - Remembering Nodes](#) can be moved to the database for faster and better storage.

The way I'm going to go to do this is through the use of docker, which is a service for generating containerised "virtual machines" that *should* run identically no matter the hardware. This means all the program has to do is tell docker to generate a PostgreSQL database image and as long as the user has docker installed, it should then be able to be connect and use this just like it would any other database.

PostgreSQL was chosen because V - the language the node is written in - supports it and I have used it in other projects before so should be able to work with it easily.

- Generate a PostgreSQL docker container and connect to it upon startup
- Build the PostgreSQL table with required fields.
- Refactor references to use the database rather than the json file used at the moment.
- Save messages to the database and only forward them through the network if the node hasn't seen it before.

Usability Features

- Docker container and database should be controlled automatically by the node software by default to minimise the amount of interactions users have to have with the software.
- Contain the ability to change database settings to use a dedicated, custom, database server to make grouping nodes together easier for the user.

Key Variables

Variable Name	Use
---------------	-----

init_psql_docker	This is a constant that defines the shell command that is run by the software and creates a docker container that runs the PostgreSQL database.
start_psql_docker	This is a constant that runs a shell command that runs to start a pre-made docker container.
stop_psql_docker	This is a constant that runs a shell command to stop a currently running docker container

Pseudocode

The two primary files to be created in this cycle are for the generation of the docker container which runs the database and the initialisation of the database tables. The other additions that need to be made are logically the same as they were with the JSON file storage method so I will not include psuedocode for them but will show the code after they are completed in the outcome section of this cycle.

Docker Container Handling

This code will allow the node software to send direct bash commands to the Operating System and through that create, start and stop a docker container which houses the PostgreSQL database used by the software.

```
// external imports
```

```

IMPORT operating_system as // os stands for operating system
IMPORT time

// config constants
image_name = "monochain-postgresql"
db_name = "postgres"
db_password = "password"

// shell commands
init_sql_docker = "docker run --name $image_name -p 5432:5432 -e POSTGRES_PASSWORD=$db_password"
start_sql_docker = "docker start $image_name"
stop_sql_docker = "docker stop $image_name"

// returns whether or not the command executed properly
FUNCTION sh(cmd):
    //sh stands for shell as in a shell command
    OUTPUT "[shell] > $cmd"
    executed = os.execute(cmd)
    OUTPUT "[shell] - $executed.output"

    // 0 means the command executed properly
    RETURN (executed.exit_code == 0)
END FUNCTION

FUNCTION launch():
    OUTPUT "[Database] Launching database..."
    // try to start the container
    started = sh(start_sql_docker)

    IF (!started):
        OUTPUT "[Database] No database found, or other error, trying to initialise a new one."
        initialised = sh(init_sql_docker)

        IF (!initialised):
            OUTPUT "\n\n[Database] Could not start or initialise database container."
            EXIT
        END IF

        OUTPUT "[Database] Database initialised successfully."
    END IF

    time.sleep(3 SECONDS)
    OUTPUT "[Database] Database running."
END FUNCTION

FUNCTION stop(){
    OUTPUT "[Database] Stopping database..."
    stopped = sh(stop_sql_docker)

    IF (!stopped):
        OUTPUT "[Database] Failed to stop database, may be due to container not exist."
    ELSE
        OUTPUT "[Database] Stopped database."
}

```

```
    END IF  
END FUNCTION
```

Database initialisation

All that needs to be done in this section is defining what the tables will look like in the database and creating functions to create those tables.

```
// external imports - PostgreSQL is the database type
```

```

IMPORT postgres as pg

// config constants
host = "localhost"
port = 5432

config = pg.Config{
    host: host
    port: port
    user: "postgres"
    password: db_password
    dbname: db_name
}

OBJECT Http_Reference:
    id          [primary; sql: serial; sql_type: 'SERIAL']      // just for the db
    domain      [default: '')]    // domain of node
    key         // key that is attached to node
    last_connected [default: 'CURRENT_TIMESTAMP'; sql_type: 'TIMESTAMP'] // when the re
END OBJECT

OBJECT Message_Table:
    id          [primary; sql: serial; sql_type: 'SERIAL']      // just for the db
    timestamp   [default: 'CURRENT_TIMESTAMP'; sql_type: 'TIMESTAMP'] // when the me
    sender      // key of the sender
    receiver    // key of the receiver
    contents    // contents of the message
    signature   // the signature of the message
END OBJECT

OBJECT DatabaseConnection:
    connection

    FUNCTION init_tables(this):
        OUTPUT "[Database] Creating http reference table..."

        SQL this.connection:
            CREATE table Http_Reference
        END SQL

        OUTPUT "[Database] Creating message table..."

        SQL this.connection {
            CREATE table Message_Table
        END SQL

        OUTPUT "[Database] Tables created.\n"
    END FUNCTION
END OBJECT

// interfacing with the tables using the pg module
FUNCTION connect(over_ride_config, supplied_config):

```

```

config_to_use = config

IF (over_ride_config):
    config_to_use = supplied_config
END IF

connection = {}

TRY:
    connection = pg.connect(config_to_use)
CATCH:
    OUTPUT "[Database] Could not connect to database, docker container probably not running"
    EXIT
END TRY

db = DatabaseConnection{connection: connection}

OUTPUT "[Database] Connected to database. ($host:$port)"

OUTPUT "[Database] Creating tables..."
db.init_tables()
RETURN db
END FUNCTION

```

Development

Although all the code shown in this cycle does not mention web-sockets, it is worth mentioning that whilst during the development of this cycle I was primarily working on switching from using json files to a database, I was also starting to work on the web-socket functionality that will be shown in the next cycle, hence the code for this cycle is available [here](#) but exploring outside the database module will show evidence of code that is referenced in the next cycle.

On that note, all of the following code is available in separate files within the 'database' file, stored at `/packages/node/src/modules/database/` with the specific file names commented at the top of the code below.

Outcome

Creating and handling the docker container

This code allows the node software to send direct bash commands to the Operating System and through that create, start and stop a docker container which houses the postgreSQL database used by the software.

```
// found at /database/containerHandling.v
module database

// external imports
import os
import time

// config constants
pub const image_name = "monochain-postgresql"
pub const db_name = "postgres"
pub const db_password = "password"

// shell commands
const init_psql_docker = "docker run --name $image_name -p 5432:5432 -e POSTGRES_PA
const start_psql_docker = "docker start $image_name"
const stop_psql_docker = "docker stop $image_name"

// returns whether or not the command executed properly
fn sh(cmd string) bool {
    println("[shell] > $cmd")
    executed := os.execute(cmd)
    print("[shell] - $executed.output")

    // 0 means the command executed properly
    return executed.exit_code == 0
}

pub fn launch(){
    println("[Database] Launching database...")
    // try to start the container
    started := sh(start_psql_docker)

    if !started {
        println("[Database] No database found, or other error, trying to in
initialised := sh(init_psql_docker)

        if !initialised {
            eprintln("\n\n[Database] Could not start or initialise data
exit(300)
    }

    println("[Database] Database initialised successfully.")
}
```

```

        time.sleep(3 * time.second)
        println("[Database] Database running.")
    }

pub fn stop(){
    println("[Database] Stopping database...")
    stopped := sh(stop_psql_docker)

    if !stopped {
        eprintln("[Database] Failed to stop database, may be due to contain
    } else {
        println("[Database] Stopped database.")
    }
}

```

Database Interactions

Interfacing with the database and creating relevant tables

The way in which the PostgreSQL module works in vlang is to create a struct that represents a table and then to use it in the table's generation, hence a lot of this section of code is just constants and structs to handle these tables and connect to the database in the first place.

The second half of the code is then just to connect to the database and create the tables based upon those references.

```

// found at /database/postgresInterface.v
module database

// external imports
import pg

// config constants
const host = "localhost"
const port = 5432

const config = pg.Config{
    host: host
    port: port
    user: "postgres"
    password: db_password
    dbname: db_name
}

pub struct Http_Reference {
    pub:
        id          int      [primary; sql: serial; sql_type: 'SERIAL']
        domain     string   [default: '' ] // domain of node
}

```

```

        key          string // key that is attached to node
        last_connected string [default: 'CURRENT_TIMESTAMP'; sql_
    }

pub struct Message_Table {
    pub:
        id          int    [primary; sql: serial; sql_type: 'SERIAL']
        timestamp   string [default: 'CURRENT_TIMESTAMP'; sql_type: 'T
        sender      string // key of the sender
        receiver    string // key of the receiver
        contents    string // contents of the message
        signature   string // the signature of the message
    }

pub struct DatabaseConnection {
    pub mut:
        connection pg.DB
}

// interfacing with the tables using the pg module
pub fn connect(over_ride_config bool, supplied_config pg.Config) DatabaseConnection
    mut config_to_use := config

    if over_ride_config {
        config_to_use = supplied_config
    }

    connection := pg.connect(config_to_use) or {
        eprintln("[Database] Could not connect to database, docker containe
        exit(310)
    }

    db := DatabaseConnection{connection: connection}

    println("[Database] Connected to database. ($host:$port)")

    println("[Database] Creating tables...")
    db.init_tables()
    return db
}

pub fn (db DatabaseConnection) init_tables() {
    println("[Database] Creating http reference table...")
    sql db.connection {
        create table Http_Reference
    }
    println("[Database] Creating message table...")
    sql db.connection {
        create table Message_Table
    }
    println("[Database] Tables created.\n")
}

```

References refactor file

This is very similar to the original references file, but is simply adjusted to using databases rather than the old json method, so I won't go into too much detail on it but I've added it below as it will be tested in the tests at the end of this cycle.

```
// found at /database/references.v
module database

import time

pub fn (db DatabaseConnection) aware_of(input_domain string) bool {
    println("[Database] Checking if domain $input_domain is in database")
    http_matches := sql db.connection {
        select from Http_Reference where domain == input_domain order by la
    }

    if http_matches.len > 0 {
        println("[Database] Found a reference to $input_domain")
        return true
    }

    println("[Database] No reference to $input_domain found.")
    return false
}

pub fn (db DatabaseConnection) get_key_http(input_domain string) []u8 {
    ref := sql db.connection {
        select from Http_Reference where domain == input_domain order by la
    }

    return ref.key.bytes()
}

pub fn (db DatabaseConnection) create_ref(input_domain string, pubkey []u8) bool {
    println("[Database] Checking if a reference to $input_domain already exists")
    if db.aware_of(input_domain) {
        println("[Database] Reference to $input_domain already exists, upda
        db.submit_update(input_domain, pubkey)
        return true
    }

    println("[Database] Creating reference for $input_domain")

    db.submit_create(input_domain, pubkey)
    println("[Database] Reference to $input_domain created successfully")
    return true
}
```

```

    }

fn (db DatabaseConnection) submit_create(input_domain string, pubkey []u8) {
    key_str := pubkey.str()
    println("[Database] Creating http reference for $input_domain")
    ref := Http_Reference{
        domain: input_domain
        key: key_str
    }
    sql db.connection {
        insert ref into Http_Reference
    }

}

fn (db DatabaseConnection) submit_update(input_domain string, pubkey []u8) {
    cur_time := time.utc()
    key_str := pubkey.str()

    println("[Database] Updating http reference for $input_domain")
    sql db.connection {
        update Http_Reference set key = key_str, last_connected = cur_time
    }
}

pub fn (db DatabaseConnection) get_refs() []Http_Reference {
    http_refs := sql db.connection {
        select from Http_Reference
    }

    return http_refs
}

```

Additional Functions

Some basic wrappers used for key functionality

These wrapper functions exist so that if a user wanted to change the node software to use a different database type or module, they would only have to change the code in the database module and wouldn't have to change sql requests in external modules.

```

// found at /database/queryCreator.v
module database

pub fn (db DatabaseConnection) get_message(parsed_signature string, parsed_sender s
existing_message := sql db.connection {
    select from Message_Table where signature == parsed_signature && se

```

```

    }

    return existing_message
}

pub fn (db DatabaseConnection) get_latest_messages(offset_quantity int, num int) [] {
    println("[Database] Getting latest messages")
    existing_message := sql db.connection {
        select from Message_Table where id > 0 order by timestamp desc limit
    }
    return existing_message
}

pub fn (db DatabaseConnection) save_message(message Message_Table) {
    sql db.connection {
        insert message into Message_Table
    }
}

```

Challenges

The main challenge this cycle was to figure out the best way of setting up a database whilst balancing how many additional things the user would have to download with how much setup and usage they would need to do.

After various testing and trialing of different methods that resulted in the use of a docker container to host the database as the setup and management of docker can be handled by the node software and simply requires the user to download docker, although I later realised that the user would also need to install PostgreSQL and libpq in order to have all the header files required to run the node. (This can be avoided by compiling directly to binary with all the required files but in most cases this isn't a great idea as most users will need to recompile the software to fit their systems)

This then means that three new requirements are needed to run the software:

- Docker
- PostgreSQL
- libpq (or libpq-dev for linux)

However a benefit to moving to docker is that in the future the entire software could be moved into a docker container, allowing the software to simply run within two docker containers, one for the software and the other for the database, which would prevent users from having to download anything except docker directly to their computer and instead all other downloads would be handled within docker without any user interference.

Testing

All of the tests for this cycle were automated using a basic script I wrote in order to be able to test the whole

module quickly and easily.

Test Table

Tests

Test	Instructions	What I expect
1	Use the Shell function to echo a message	The message to be echo'ed from the terminal
2	Launch the database.	The database to launch and no crashes to occur.
3	Stop the database.	The database to stop and no crashes to occur.
4	Launch and then connect to the database.	The database to launch then accept the program's connection without any crashes.
5	Get the last 5 messages from the database.	An array of messages to be returned in the expected type.
6	Create a node reference in the database.	The reference to be created in the database.
7	Create a node reference and then check if the software is aware of that node.	The "aware_of" check should return true for the reference test that already exists.
8	Collect all references from the database.	Should return an array of node references in the expected type.

Testing Code

All the functions in this script that start with "test_" are testing functions that can be run using the vlang test functionality which allows the use of a shell command that looks something like the following: `v test ./file_test.v`

```

import database
import pg

// The actual tests:

fn test_shell(cmd) {
    echo "hello world"
    assert database.ch(cmd)
}

```

```
        assert database.smithy()

    }

fn test_db_launch() {
    database.launch()
    // this test will crash if it fails
    assert true
}

fn test_db_stop() {
    database.launch()
    database.stop()
    // this test will crash if it fails
    assert true
}

fn test_db_connection() {
    database.launch()
    db := database.connect(false, pg.Config{})
    database.stop()
    // this test will crash if it fails
    assert true
}

fn test_messages() {
    eprintln("WARNING - THIS TEST RELIES ON THE DATABASE HAVING ATLEAST 1 MESSAGE")
    database.launch()
    db := database.connect(false, pg.Config{})
    messages := db.get_latest_messages(0, 5)
    database.stop()
    // this test will crash if it fails
    assert messages.len > 0
}

fn test_create_ref() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)
    database.stop()
    // this test will crash if it fails
    assert true
}

fn test_aware_ref() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)
    aware := db.aware_of('test')
    database.stop()
    assert aware
}

fn test_refs() {
```

```

... code ...

    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)      // create a ref if the test one doesn't already exist
    refs := db.get_refs()
    database.stop()
    assert refs.len > 0
}

// This is a function used just to make the other tests easier
fn create_test_ref(db DatabaseConnection) {
    mut aware := db.aware_of('test')

    if !aware {
        // just in case the test ref doesn't exist
        db.create_ref('test', 'test'.bytes())
        aware = db.aware_of('test')
    }
}

```

Evidence

Test 1

```

Running tests in /Users/derrick/Workshop/Projects/A-Level Project
[shell] > echo "hello world"
[shell] - hello world
OK [1/8] 2.909 ms 1 assert | main.test_shell()

```

Test 2

```

[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
OK [2/8] 3467.944 ms 1 assert | main.test_db_launch()

```

Test 3

```

[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [3/8] 3410.560 ms 1 assert | main.test_db_stop()

```

Test 4

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [4/8] 3688.393 ms 1 assert | main.test_db_connection()
```

Test 5

```
WARNING - THIS TEST RELIES ON THE DATABASE HAVING ATLEAST 1 MESSAGE, IF IT DOES NOT, IT WILL FAIL
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Getting latest messages
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [5/8] 3485.572 ms 1 assert | main.test_messages()
```

Test 6

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
```

```
[Database] Stopped database.  
OK [6/8] 3520.513 ms 1 assert | main.test_create_ref()
```

Test 7

```
[Database] Launching database...  
[shell] > docker start monochain-postgresql  
[shell] - monochain-postgresql  
[Database] Database running.  
[Database] Connected to database. (localhost:5432)  
[Database] Creating tables...  
[Database] Creating http reference table...  
NOTICE: relation "Http_Reference" already exists, skipping  
[Database] Creating message table...  
NOTICE: relation "Message_Table" already exists, skipping  
[Database] Tables created.  
  
[Database] Checking if domain test is in database  
[Database] Found a reference to test  
[Database] Checking if domain test is in database  
[Database] Found a reference to test  
[Database] Stopping database...  
[shell] > docker stop monochain-postgresql  
[shell] - monochain-postgresql  
[Database] Stopped database.  
OK [7/8] 3496.402 ms 1 assert | main.test_aware_ref()
```

Test 8

```
[Database] Launching database...  
[shell] > docker start monochain-postgresql  
[shell] - monochain-postgresql  
[Database] Database running.  
[Database] Connected to database. (localhost:5432)  
[Database] Creating tables...  
[Database] Creating http reference table...  
NOTICE: relation "Http_Reference" already exists, skipping  
[Database] Creating message table...  
NOTICE: relation "Message_Table" already exists, skipping  
[Database] Tables created.  
  
[Database] Checking if domain test is in database  
[Database] Found a reference to test  
[Database] Stopping database...  
[shell] > docker stop monochain-postgresql  
[shell] - monochain-postgresql  
[Database] Stopped database.  
OK [8/8] 3554.807 ms 1 assert | main.test_refs()
```

2.2.14 Cycle 14 - Adding Web sockets for dynamic Nodes.

Design

Objectives

Currently the only way a node can contribute to a network is if it either has a static public DNS/IP address that it can be reached at or if it has a direct local connection to a node with such an address. This is a problem because it means that users have to setup some kind of port forwarding, networking tunnel or some other somewhat complicated method just to connect to the network which increases the barrier to entry and is obviously not good.

The solution to this is to introduce web sockets so that "private" nodes without a public address can open a communication tunnel to a public node that can then communicate with it in either direction and therefore continue to use it for the network without having such a high barrier to entry.

- Convert the main section of the handshake method into a function so that it can be used with either web sockets or http.
- Add a web sockets object to the references created in the last cycle so that nodes don't get http addresses and web socket ids mixed up.
- Successfully ensure that the handshake method works for both http and web sockets.

Usability Features

- Internal functionality - The actual logic of message parsing, validation and what to do with that data should be separated out such that it can be reused by any method of communication (http, websocket, etc)
- Bidirectional communication - Allows worker nodes to communicate with and get information from the network in realtime without having to publicly share networking data.

Key Variables

Variable Name	Use
Websocket_Server	This object holds all the methods/functions for the web-socket server and it what allows web-socket connections to be generated and used.
WS_Object	This is the object which is sent and received through web-socket connections and allows messages to be standardised.

Pseudocode

The amount of code that this cycle will require is likely to be quite significant so I will be making the pseudocode for this cycle much higher level so it is easier to understand what is going on.

However the below pseudocode still contains all the key functionality of the websockets server.

The Websocket Server

```

OBJECT Websocket_Server:
    connections

    FUNCTION send_to_all(this, msg) {
        OUTPUT "[Websockets] Sending message to all clients..."

        FOR (client IN this.connections):
            TRY:
                client.send(msg)
            CATCH:
                OUTPUT "[Websockets] Failed to send a message"
                RETURN false
            END TRY
        END FOR

        // nothing went wrong so return true
        RETURN true
    END FUNCTION

    FUNCTION connect(this, ref):
        OUTPUT "[Websockets] Connecting to server $ref"

        TRY:
            new_connection = CONNECT TO ref
            this.connections.append(new_connection)
            OUTPUT "[Websockets] Connected to $ref successfully"
        CATCH:
            OUTPUT "[Websockets] Failed to connect to $ref"
            RETURN false
        END TRY

        // nothing went wrong so return true
        RETURN true
    END FUNCTION

    FUNCTION listen(this):
        FOR (client in this.connections):
            TRY:
                client.listen()
                OUTPUT "[Websockets] Now listening to $client"
            CATCH:
                OUTPUT "[Websockets] Failed to listen to $client"
            END TRY
        END FOR
    END FUNCTION

    // the function that is run anytime a connection receives a message
    FUNCTION on_message(this, msg, sender):
        OUTPUT "[Websockets] Received message: $msg"

        // first check the message's encoding method, this software only uses
        // text so the opcode must be that of text to say that the message

```

```

// was encoded in plain text.

SWITCH msg.opcode
    CASE text:
        // now the message is decoded into a json object.
        parsed_msg = {}

    TRY:
        parsed_msg = json.decode(WS_Object, msg.payload.str())
    CATCH:
        OUTPUT '[Websockets] Could not parse message: $err'
        RETURN FALSE
    END TRY

    // if this object is a Broadcast request then handle it.
    IF parsed_msg IS A BROADCAST MESSAGE:
        OUTPUT '[Websockets] received broadcast message: $parsed_msg'
        // check if the request is valid
        valid = CHECK IF parsed_msg IS VALID

        IF (valid):
            // request is valid, so send a success response
            OUTPUT "[Websockets] Broadcast message was valid"
            sender.send(mut ws, json.encode(SUCCESS{"Broadcast": true}))
        ELSE:
            // request was invalid, send an error response
            println('[Websockets] Broadcast message was invalid')
            sender.send(mut ws, json.encode(ERROR{"Broadcast": false}))
        END IF

        ELSE IF parsed_msg IS A SUCCESS MESSAGE:
            // received a success response to a message.
            OUTPUT "[Websockets] Received success message: $parsed_msg"

        ELSE IF parsed_msg IS AN ERROR MESSAGE:
            // received an error response to a message.
            OUTPUT "[Websockets] Received error message: $parsed_msg"

        ELSE:
            // object is of an unknown type
            OUTPUT "[Websockets] Received unknown message: $parsed_msg"

    END IF
END CASE

OTHERWISE:
    // if this is called then the message was encoded incorrectly
    OUTPUT "[Websockets] received unknown message: $msg"
END OTHERWISE
END SWITCH

END FUNCTION

```

```
END OBJECT
```

Development

Since the web sockets have two types of connections, servers and clients, some kind of wrapper function will be required to prevent all parts of the code from having to have two methods for both objects and instead simplifying it to being called on the wrapper object and then that object handling how to deal with the client and server respectively.

The other key section of code that needs to be developed is how the web-socket object is then passed around the code to various api endpoint functions, as these api endpoints will need to be able to send their own web-socket messages for functionality such as message forwarding. To do this, I plan to simply pass the object as a "shared" parameter into the api generator so that the endpoints can use it as required, however since I am not completely sure on how V will handle this I will first write a test program which will be shown below in the outcome section.

The reason that I didn't use the shared parameter when dealing with the configuration object earlier on is because when I last looked into this type of parameter the Vweb module didn't fully support it, thus it caused some weird side effects, but Vweb should now support shared parameters to their full extent so it should all work as expected.

Outcome

The test code

This was the basic test code I wrote in order to confirm that shared parameters do in fact work as they are described to and that I can therefore use them to pass the web-socket connections around to the various api endpoints as required.

This code will be ran and demonstrated in test 1 of the testing section further along in this cycle.

```

struct Info {
    test int
}

struct App {
    vweb.Context
    info shared Info
}

pub fn start(config configuration.UserConfig) {
    info := Info{test: 0}
    app := App{info: info}
    api := go vweb.run(app, config.port) // start server on a new thread

    // there's some other code here but it isn't important in this cycle

    api.wait()      // bring server process back to main thread
}

["/test"]
pub fn (mut app App) test() vweb.Result {
    println(app)
    mut result := ""
    lock app.info {
        cur := app.info.test
        result = json.encode(cur)
        app.info = Info{test: cur + 1}
    }
    return app.text(result)
}

```

The web socket client file

This code handles the ability for multiple client web-socket connections and will be used by private nodes that do not intend to host servers for other clients to connect to and hence need the ability to create and store multiple client connections whilst still being able to create connections easily.

```

// Found at /packages/node/src/modules/server/ws_client.v
module server

// internal modules
import database
import configuration

// external
import net.websocket
import log

// this represents the "client" object that can hold multiple connections
struct Client {
    db database.DatabaseConnection [required]
    config configuration.UserConfig [required]
    mut:
        connections []websocket.Client
}

// this initiates the original client object without any connections
pub fn start_client(db database.DatabaseConnection, config configuration.UserConfig) Client {
    c := Client{
        db: db
        config: config
        connections: []websocket.Client{}
    }

    println("[Websockets] Created client.")
    return c
}

// this is ran on the client object and connects to a server
pub fn (mut c Client) connect(ref string) bool {
    // create a new client connection object
    mut ws := websocket.new_client(ref, websocket.ClientOpt{logger: &log.Logger(&log.Log{
        level: .info
    })) or {
        eprintln("[Websockets] Failed to connect to $ref\n[Websockets] Error: $err")
        return false
    }

    // setup logging functions
    ws.on_open(socket_opened)
    ws.on_close(socket_closed)
    ws.on_error(socket_error)

    // setup messaging function
    println('[Websockets] Setup Client, initialising handlers... ')
    ws.on_message_ref(on_message, &c)

    // actually connect to the server
    ws.connect() or {

```

```

        eprintln("[Websockets] Failed to connect to $ref\n[Websockets] Error: $err")
    }
    // start listening to the connection on a new thread
    go ws.listen()

    // add the connection to the client's connection array.
    c.connections << ws
    println('[Websockets] Connected to $ref')
    return true
}

pub fn (mut c Client) send_to_all(data string) bool {
    // this loops through all connections and sends a message to each one
    // then waits for the threads those messages were initiated on to return.

    println("[Websockets] Sending a message to all ${c.connections.len} clients")
    mut threads := []thread bool{}
    for mut connection in c.connections {
        println("[Websockets] Starting a new thread to send a message to $connection.")
        threads << go send_ws(mut connection, data)
    }

    println("[Websockets] Waiting for all threads to finish")
    threads.wait()
    println("[Websockets] All threads finished, message sent.")
    return true
}

fn socket_opened(mut c websocket.Client) ? {
    // this runs everytime a new socket connection is opened.
    println("[Websockets] Socket opened")
}

fn socket_closed(mut c websocket.Client, code int, reason string) ? {
    // this runs everytime a socket connection is closed
    println("[Websockets] Socket closed, code: $code, reason: $reason")
}

fn socket_error(mut c websocket.Client, err string) ? {
    // this runs any time an error occurs with a socket connection.
    println("[Websockets] Socket error: $err")
}

```

The web socket server file

This file contains all the logic and code required to assemble a web-socket server for use in public nodes, although it may seem as if both the client and server files do very similar things, they are fundamentally different in the fact that the web-socket module built into V does not support creating connections as a server object or receiving them as a client.

This means that public nodes that wish to accept incoming connections must use the server object type whilst private nodes that wish to send out outgoing connections must use the client object type, even if the logic in each is almost identical.

```

// Found at /packages/node/src/modules/server/ws_server.v
module server

// internal modules
import database
import configuration

// external modules
import net.websocket
import log

// the server object
struct Server {
    db database.DatabaseConnection [required]
    config configuration.UserConfig [required]
    mut:
        sv websocket.Server [required]
}

pub fn start_server(db database.DatabaseConnection, config configuration.UserConfig) Server {
    // create and setup the server object to accept incoming connections
    mut sv := websocket.new_server(.ip, config.ports.ws, "", websocket.ServerOpt{
        logger: &log.Logger(&log.Log{
            level: .info
        })
    })
    mut s := Server{db, config, sv}

    println("[Websockets] Server initialised on port $config.ports.ws, setting up handlers")
    sv.on_message_ref(on_message, &s)

    println("[Websockets] Server setup on port $config.ports.ws, ready to launch.")
    // this does not start listening to the server, need to do that later
    return s
}

pub fn (mut s Server) listen() {
    // this allows the server to actually start listening for connections.
    S.sv.listen() or {
        eprintln("[Websockets] ERROR - Error listening on server: $err")
        exit(1)
    }
}

pub fn (mut s Server) send_to(id string, data string) bool {
    // this sends a message to a specific client connection based upon id.

    println("[Websockets] Sending a message to $id")

    mut cl := S.sv.clients[id] or {
        eprintln("[Websockets] Client $id not found")
        return false
    }
}

```

```

    }

    cl.client.write_string(data) or {
        eprintln("[Websockets] Failed to send data to client $id")
    }

    println("[Websockets] Message sent to $id")
    return true
}

pub fn (mut S Server) send_to_all(data string) bool {
    // this loops through all connections and sends a message to each one.

    len := S.sv.clients.keys().len
    println("[Websockets] Sending a message to all $len clients")

    mut threads := []thread bool{}
    for id in S.sv.clients.keys() {
        println("[Websockets] Sending message to socket with $id")
        threads << go S.send_to(id, data)
    }

    println("[Websockets] Waiting for all threads to finish")
    threads.wait()
    println("[Websockets] Message sent to all clients")
    return true
}

```

The generic web socket file

This file includes two main sections: one for wrapping functionality over both the client and server objects so the rest of the code doesn't have to worry about handling both object types and can just call any methods it needs to call; and the other for receiving messages then decoded and using them as they would be used by the http side of the project.

The wrapper object

This is the first half of the code and simply helps make it easier for other parts of the code to deal with the web-socket connections as described in the summary of this file.

```

// Found at /packages/node/src/modules/server/ws_generic.v

module server

// internal
import database
import configuration

// external
import net.websocket
import json

// websocket server

struct Websocket_Server {
    is_client bool [required]
    db database.DatabaseConnection [required]
    mut:
        c Client
        s Server
}

pub fn (mut ws Websocket_Server) send_to_all(msg string) bool {
    // this selects between the client and server object and sends a request
    // to the relevant object to send a message to all of it's connections.

    println("[Websockets] Sending message to all clients...")
    if ws.is_client {
        println("[Websockets] Sending messages as a client...")
        return ws.c.send_to_all(msg)
    } else {
        println("[Websockets] Sending messages as a server...")
        return ws.s.send_to_all(msg)
    }
    return false
}

pub fn (mut ws Websocket_Server) connect(ref string) bool {
    // if this is called by a client object then a connection will be established
    // with the reference supplied, but if it was a server then an error message
    // will be returned as that is not possible.

    println("[Websockets] Connecting to server $ref")
    if ws.is_client {
        println("[Websockets] Connecting as a client...")
        return ws.c.connect(ref)
    } else {
        println("[Websockets] Cannot connect as a server, only clients can connect to")
        return false
    }

    return false
}

```

```
}
```

```
pub fn (mut ws Websocket_Server) listen() {
    // servers are required to listen for incoming connections hence
    // this starts that process when called and does nothing for a client as they
    // don't need to.

    if !ws.is_client {
        println("[Websockets] Listening for connections...")
        ws.s.listen()
    } else {
        println("[Websockets] Cannot listen as a client, only servers can listen for connections")
        return
    }
}
```

The generic functions

These functions are for use by both the client and server web-socket objects and are also useful in other parts of the program hence are stored as "generic functions" as they can be used generically.

```

pub fn gen_ws_server(db database.DatabaseConnection, config configuration.UserConfig) Websocket {
    // this simply attempts to create a websocket server
    // based upon the node's config settings.

    if config.self.public {
        println("[Websockets] Node is public, starting server...")
        return Websocket_Server{
            is_client: false,
            db: db,
            s: start_server(db, config)
        }
    } else {
        println("[Websockets] Node is private, starting client...")
        return Websocket_Server{
            is_client: true,
            db: db,
            c: start_client(db, config)
        }
    }

    eprintln("[Websockets] Error starting websocket server.")
    exit(1)
}

// structs and types used for decoding and encoding messages.

struct WS_Error {
    code int
    info string
}

struct WS_Success {
    info string
}

type WS_Object = Broadcast_Message | WS_Error | WS_Success

// the function that is run anytime a connection receives a message
fn on_message(mut ws websocket.Client, msg &websocket.Message, mut obj &Websocket_Server) ? {
    println('[Websockets] Received message: $msg')

    // first check the message's encoding method, this software only uses
    // text so .text_frame is the only option here.
    match msg.opcode {
        .text_frame {
            // now the message is decoded into a json object.
            parsed_msg := json.decode(WS_Object, msg.payload.str()) or {
                eprintln('[Websockets] Could not parse message: $err')
                return
            }

            // if this object is a Broadcast request then handle it.
            if parsed_msg is Broadcast_Message {

```

```

    println('[Websockets] received broadcast message: $parsed_msg')
    // check if the request is valid
    valid := broadcast_receiver(obj.db, mut obj, parsed_msg)

    if valid == .ok {
        // request is valid, so send a success response
        println('[Websockets] Broadcast message was valid')
        send_ws(mut ws, json.encode(WS_Success{"Broadcast messag
    } else {
        // request was invalid, send an error response
        println('[Websockets] Broadcast message was invalid')
        send_ws(mut ws, json.encode(WS_Error{1, "Broadcast me
    }

} else if parsed_msg is WS_Success {
    // received a success response to a message.
    println("[Websockets] Received success message: $parsed_msg.i
} else if parsed_msg is WS_Error {
    // received an error response to a message.
    eprintln('[Websockets] Received error message: $parsed_msg.in
} else {
    // object is of an unknown type
    eprintln('[Websockets] Received unknown message: $parsed_msg'
}

}

else {
    // if this is called then the message was encoded incorrectly.
    println('[Websockets] received unknown message: $msg')
}

}

// this wraps the "write_string" method for more graceful error handling.
fn send_ws(mut ws websocket.Client, msg string) bool {
    ws.write_string(msg) or {
        eprintln('[Websockets] Could not send message: $err')
        return false
    }
    return true
}

```

The commit for this code is available [here](#).

Challenges

The main challenge faced through the development of this cycle was not one of my own code, but instead of the language that I am using's standard library. In particular, the `log` module that is included within it.

The Problem

The error in particular was due to the generation of a function to generate a 'logger' (an object that allows for the logging of the program in a log file or terminal output) which upon being fed the value 'nil' (representing nothing) should generate a default logger using the Log object, however in certain circumstances it was instead attempting to generate the logger from a `voidptr` object - which is just an empty object in C.

This happens because Vlang is built on top of the low level language C, and therefore when it is compiled, the code gets compiled from V to C and then from C to an executable for whatever platform is being targeted.

This is the C code being generated by Vlang.

```
.logger = HEAP(log(Logger, /*&log.Logger*/I_voidptr_to_Interface_log_LOGGER(HEAP(voidptr, (
```

This is the C code that should've been generated.

```
.logger = HEAP(log(Logger, /*&log.Logger*/I_log_Log_to_Interface_log_LOGGER(((log_Log*)mem
```

The fix

Although completing the actual fix was fairly quick and easy, finding what was broken in the first place was a lot more complicated. This involved writing testing files to ensure it was actually a compiler issue and not just my code, then careful reading of the semi-compiled C files generated by Vlang to find the compiler error, then changing pieces of code in the compiler one line at a time to figure out how to prevent it from happening until eventually - after about 3 days of repetitive bug hunting - I stumbled upon what I needed.

The quick fix for this was relatively simple and only required changing the default value for the logger inside the `net.websocket` module (which is where the logger module was incorrectly generating code) from `nil` to a `Log` object although this didn't actually fix the root problem, it did solve it enough for me to carry on working on my project and fixing the root problem would've been far outside the scope of this project.

After creating this fix and validating that it didn't break anything else, I realised that other developers with less experience in this kind of thing might be having this issue and be getting completely stuck so I decided to share my edited version of the compiler by submitting a 'pull request' to the Github repository. This allows the team that make the Vlang compiler to approve my change and make it part of the official Vlang compiler.

One of the members of the Vlang team then confirmed that the bug I had discovered did in fact exist, validated that my version didn't break anything else, created a test to stop it happening in the future and then merged my changes into the compiler. This means that as a byproduct of this A Level project I have contributed to the compiler of a language used by hundreds of thousands of people!

The code I changed

Here are the two fixes I made, the top image looks like I changed a lot more than I actually did but that's just due to me changing the code comments slightly.

```

v 13 vlib/net/websocket/websocket_client.v
@@ -36,11 +36,14 @@ pub mut:
36   36     header      http.Header // headers that will be passed when connecting
37   37     conn        &net.TcpConn = unsafe{ nil } // underlying TCP socket connection
38   38     nonce_size    int = 16 // size of nounce used for masking
39 -   panic_on_callback bool        // set to true if callbacks can panic
40 -   state        State        // current state of connection
41 -   logger       &log.Logger = unsafe{ nil } // logger used to log messages
42 -   resource_name string      // name of current resource
43 -   last_pong_ut   i64         // last time in unix time we got a pong message
44 +   panic_on_callback bool        // set to true if callbacks can panic
45 +   state        State        // current state of connection
46 +   logger       &log.Logger = &log.Logger(&log.Log{
47 +     level: .info
48 +   })
49   +   resource_name string      // name of current resource
46 +   last_pong_ut   164        // last time in unix time we got a pong message
44  47 }
45  48
46  49 // Flag represents different types of headers in websocket handshake

```

The code I changed (green is my code and red is what it replaced)

```

v 4 vlib/net/websocket/websocket_server.v
@@ -9,7 +9,9 @@ import rand
 9   9   // Server represents a websocket server connection
10  10  pub struct Server {
11  11    mut:
12 -   logger      &log.Logger = unsafe{ nil } // logger used to log
12 +   logger      &log.Logger = &log.Logger(&log.Log{
13 +     level: .info
14 +   })
13  15  ls          &net.TcpListener = unsafe{ nil } // listener used to get incoming connection to socket
14  16  accept_client_callbacks []AcceptClientFn // accept client callback functions
15  17  message_callbacks      []MessageEventHandler // new message callback functions

```

The code I changed (green is my code and red is what it replaced)

 master ▾ [v / vlib / net / websocket / websocket_client.v](#)



AlfieRan [net.websocket: swap unsafe use of nil for a safe default value \(#15836\)](#) ✓

 8 contributors

499 lines (473 sloc) | 13.7 KB

```

1 // websocket module implements websocket client and a websocket server
2 // attribution: @thecoderr the author of original websocket client
3 [manualfree]

```

A screenshot of my commit in the official V git repository.

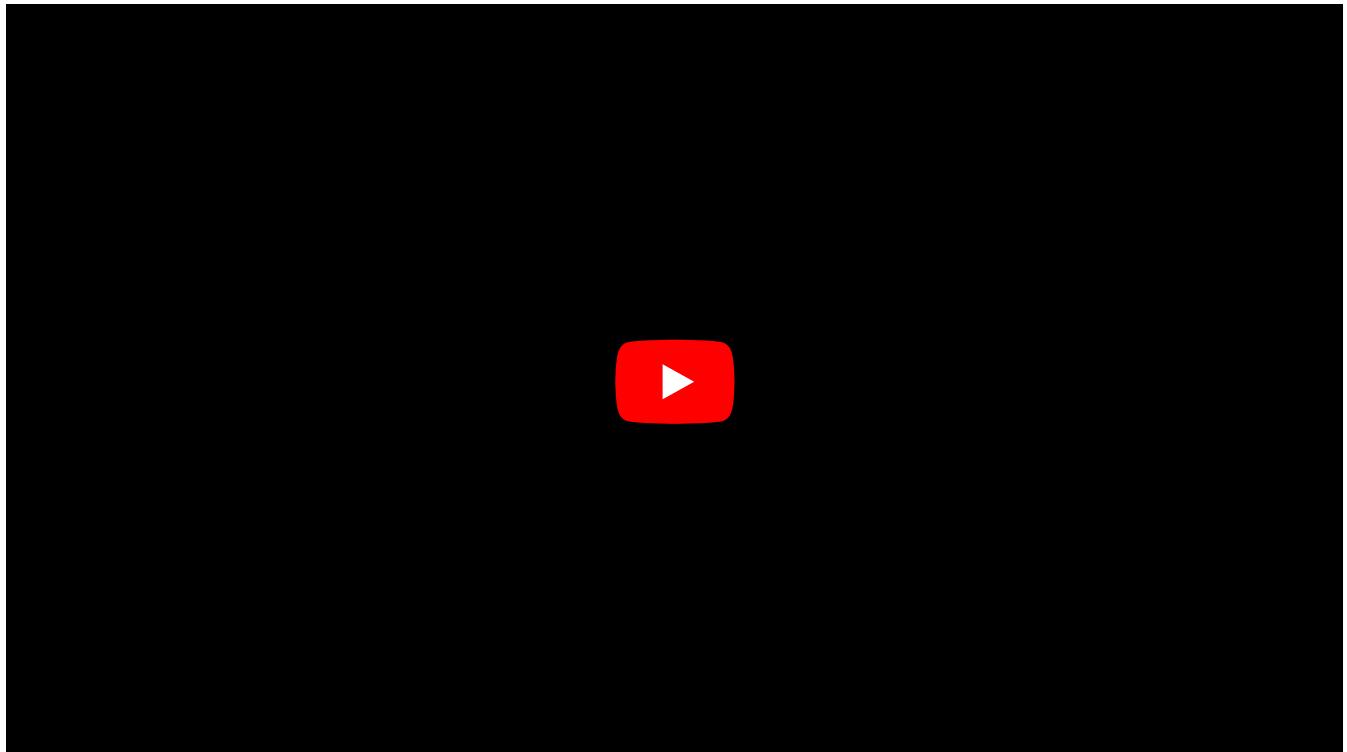
Testing

Tests

Test	Instructions	What I expect
1	Run the test code, navigate to "http://localhost:8000/test"	A number to be displayed which increases by 1 for each page refresh.
2	Send a message using web sockets	The message sent to be transmitted and received successfully.
3	Send multiple messages for a prolonged period of time (once every 0.5 seconds for 30 seconds).	The messages to continue to be transmitted and validated consistently during the entire time.

Evidence

Test 1



Test 2

```
[Broadcaster] Assembling message with data: hi from linux :)
[utils] Reading file: ./monochain/config.json

[config] Config Loaded...
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[Database] Saving message to database.
[Database] Message saved.
[Broadcaster] Message assembled, broadcasting to refs...
[Broadcaster] Sending message to all known nodes.
[Broadcaster] Sending message to public nodes.
[Broadcaster] Attempting to send message to https://nano.monochain.network
[Broadcaster] Attempting to send message to http://192.168.1.20:8000
[Broadcaster] Sending message to websocket nodes.
[Broadcaster] Attempting to send message to http://192.168.1.20:8002
[Broadcaster] Created threads to send message to all known nodes.
[Broadcaster] Waiting for threads to return.
/tmp/v_1000/node.4283625912010421186.tmp.c:13185: at server__Websocket_Server_send_to_all_thread_wrapper: RUNTIME ERROR: invalid memory access
00755a28 : by ???
[Broadcaster] Failed to send a message to http://192.168.1.20:8000, Node is probably offline. Error: dial_tcp failed for address 192.168.1.20:8000
tried addrs:
    192.168.1.20:8000: net: socket error: 111; code: 111

0074a770 : by ???
7f9f272088 : by ???
7f9f3670cc : by ???
```

Websocket message attempted to be sent from a Linux machine

```
[Broadcaster] Assembling message with data: hi from macbook to macbook
[utils] Reading file: ./monochain/config.json
```

```
[config] Config Loaded...
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[Database] Saving message to database.
[Database] Message saved.
[Broadcaster] Message assembled, broadcasting to refs...
[Broadcaster] Sending message to all known nodes.
[Broadcaster] Sending message to public nodes.
[Broadcaster] Attempting to send message to http://192.168.1.2:8000
[Broadcaster] Attempting to send message to http://192.168.1.4:8000
[Broadcaster] Attempting to send message to http://nano:8000
[Broadcaster] Attempting to send message to test
[Broadcaster] Sending message to websocket nodes.
[Websockets] Sending message to all clients...
[Websockets] Sending a message to all 1 clients
[Websockets] Sending message to socket with 8d753e7f933f9b84e38f812e6d6ae61a
[Websockets] Message sent to 8d753e7f933f9b84e38f812e6d6ae61a
[Websockets] Message sent to all clients
```

Websocket message successfully sent between two macOS devices.

Test 3

Sending messages once every 0.5 seconds for a prolonged period of time.

```
[Broadcaster] Received message:
[Broadcaster] Sender: [75, 226, 169, 219, 199, 15, 152, 247, 10, 70, 102, 135, 149, 89, 193, 167, 70, 81, 84, 215, 5, 39, 178, 0, 169, 3, 165, 45, 124, 178, 107, 202]
[Broadcaster] Sent at: 2022-09-22 12:03:11
[Broadcaster] Message: jumagi1663848191170

[Broadcaster] Sending message to all known nodes.
[Broadcaster] Sending message to public nodes.
[Broadcaster] Have not seen message before.
[Broadcaster] Saving message to database.
V panic: array.get: index out of range (i == 0, a.len == -1)
v hash: bbff1ba4
0  src          0x00000001006ccf8 array_get + 224
1  src          0x00000001007d6174 database__DatabaseConnection_get_refs + 968
2  src          0x00000001007d68d8 server__forward_to_all + 172
3  src          0x00000001007d6768 server__broadcast_receiver + 1120
4  src          0x00000001007beaac server__App_broadcast_route + 364
5  src          0x00000001007ba39c vweb__handle_conn_T_server__App + 6580
6  src          0x00000001006c5fb0 vweb__handle_conn_T_server__App_thread_wrapper + 96
7  src          0x00000001007fd25c GC_start_routine + 104
8  libsystem_pthread.dylib 0x00000001906b826c __pthread_start + 148
9  libsystem_pthread.dylib 0x00000001906b308c thread_start + 8
error: Command failed with exit code 1.
info: Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
alfier@Alfies-MacBook-Pro node %
```

3 Testing

3.1 Testing for Function and Robustness

Criteria To Assess

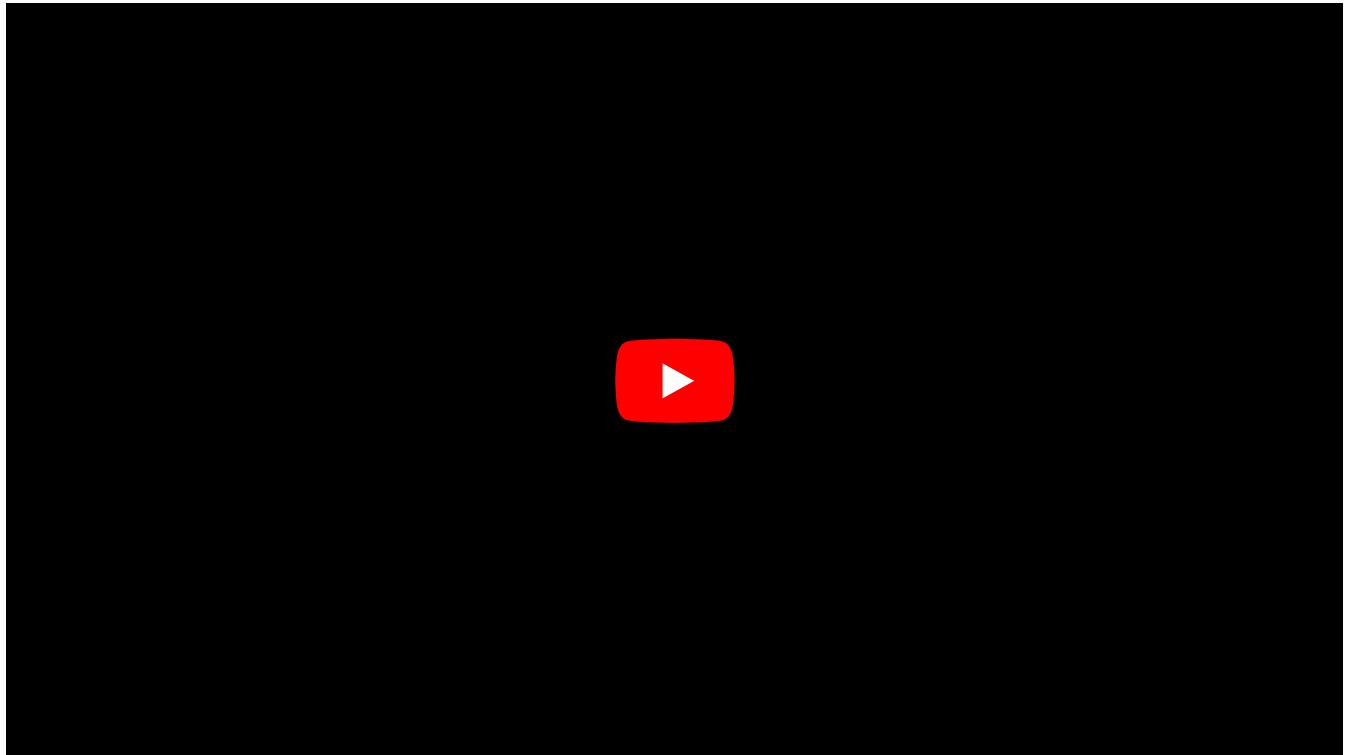
In my success criteria, I had 4 criteria which were related to the function and robustness of the project, three of which were in relation to the node software and one the webportal (website). Since development of the project has now halted, it is time to see how the project fits or doesn't fit into these criteria.

Criterion	Description
14	The node software should not crash.
15	The software should be capable of receiving multiple messages per second.
16	The software should attempt to retry any failed processes (such as loading a file) and should only exit after multiple failed attempts.
24	The web portal should run without crashing under any circumstance

Criterion 14

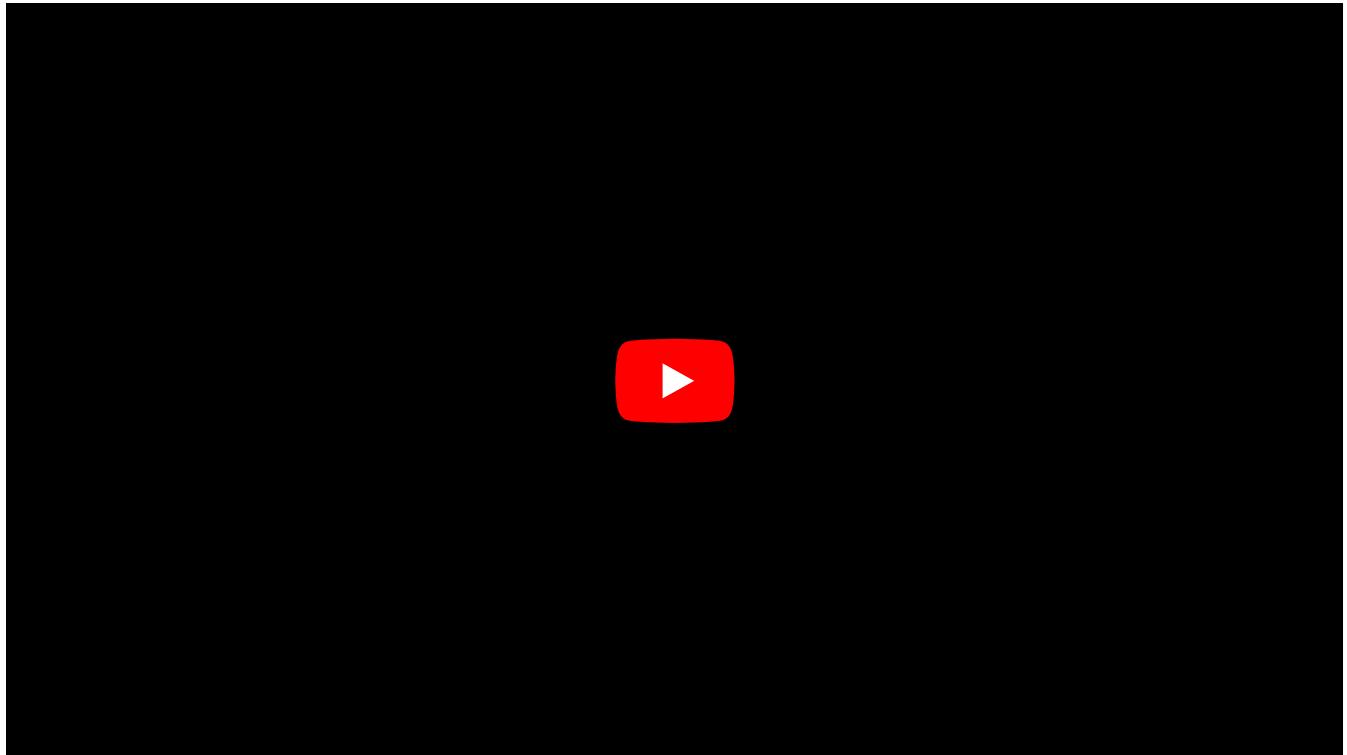
Although the software does not crash in most circumstances, under a large load it will still crash and since part of the testing involved putting the software under considerable load for a long period of time and it routinely crashed, this criterion has sadly not been met.

This is due to the software running directly on the cpu it is operating on and VLang not having an overhead, therefore if the software crashes it cannot be recovered so although most errors are handled internally through my code, some errors are outside of the scope of the program and cannot be saved once they occur. This effectively means that the only way to stop the program crashing with these kind of errors would be to prevent them from happening, instead of the traditional method to "wrap" the parts of the program that have these crashes with a error handling function since that doesn't exist in VLang.



General usage demo shown running without problems

More evidence for these crashes being a combined cause of both my code and VLang itself is how it behaves differently depending on the operating system it runs on. When using macOS the software requires a pretty significant load of multiple messages per second for 10-20 seconds, generating processes with ~4 threads running at any given time (although these are "green" threads that are based upon software and don't run in a separate hardware thread each). However, when using Linux the software crashes every time that it tries to send a web-socket message which shows that VLang is not as consistent as it would ideally be. Obviously some variations would be expected and, once again, optimising/refactoring my code and how it handles multiple threads would likely increase the stability of the software, but the fact that the exact same code compiled natively on different systems can run so fundamentally differently, without any form of compiler errors or warnings goes to show how VLang still has a long way to mature. Evidence for these claims can be found in the video below.



A demo showcasing how Vlang performs differently depending on different systems

Criterion 15

The testing for this criterion involved setting up a basic js script that simulated the api requests being made by the dashboard to the broadcast route so as to produce messages, this sent the message "jumagi" paired with the current epoch timestamp to send messages once to twice per second for a prolonged period of time.

This was completed using two separate nodes setup on a local network such that no external issues could be the cause of any issues. This resulted in the nodes successfully communicating for approximately 10-20 seconds before the one that sent the messages (no matter which node was used) crashing with a panic from within the built in "database" module. This means that the issue was probably caused by the Vlang database module not expecting to having to get and send messages multiple times per second and panicking because of that.

The below images show the dashboard during the test and the error message.

Monochain Dashboard

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:11
Message: jumagi1663848191172

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:10
Message: jumagi1663848190172

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:10
Message: jumagi1663848190670

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:09
Message: jumagi1663848189170

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:09
Message: jumagi1663848189670

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:08
Message: iumarai1663848188172

Sender: 4be2a9dbc70f98f70a4666879559c1a7465154d70527b200a903a52d7cb26bca
Timestamp: 2022-09-22 12:03:08
Message: iumarai1663848188670

Send

To get collect new messages ->

```
[Broadcaster] Received message:  
[Broadcaster] Sender: [75, 226, 169, 219, 199, 15, 152, 247, 10, 70, 102, 135, 149, 89, 193, 167, 70, 81, 84, 215, 5, 39, 178, 0, 169, 3, 165, 45, 124, 178, 107, 202]  
[Broadcaster] Sent at: 2022-09-22 12:03:11  
[Broadcaster] Message: jumagi1663848191170  
  
[Broadcaster] Sending message to all known nodes.  
[Broadcaster] Sending message to public nodes.  
[Broadcaster] Have not seen message before.  
[Broadcaster] Saving message to database.  
V panic: array.get: index out of range (i == 0, a.len == -1)  
v hash: bbff1ba4  
0 src 0x00000001006ccccf8 array_get + 224  
1 src 0x00000001007d6174 database_DatabaseConnection_get_refs + 968  
2 src 0x00000001007d68d8 server_forward_to_all + 172  
3 src 0x00000001007d6768 server_broadcast_receiver + 1120  
4 src 0x00000001007bbeac server_App_broadcast_route + 364  
5 src 0x00000001007ba39c vweb_handle_conn_T_server_App + 6580  
6 src 0x00000001006c5fb0 vweb_handle_conn_T_server_App_thread_wrapper + 96  
7 src 0x00000001007fd25c GC_start_routine + 104  
8 libsystem_pthread.dylib 0x00000001906b826c _pthread_start + 148  
9 libsystem_pthread.dylib 0x00000001906b308c thread_start + 8  
error Command failed with exit code 1.  
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.  
alfier@Alfies-MacBook-Pro node %
```

The evidence from a prolonged test in cycle 14

This shows that this criterion was not met, and it is once again, partially due to Vlang's limitations as a language and how my code was written.

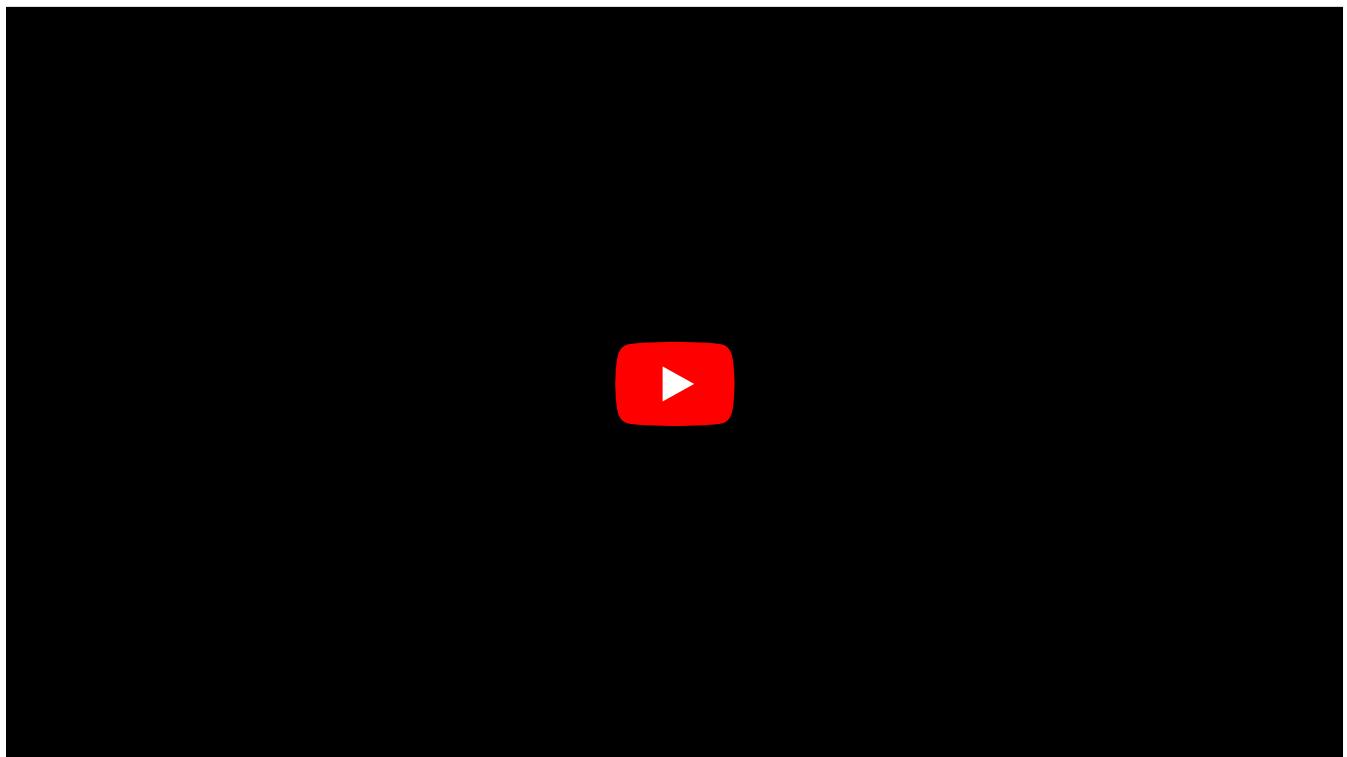
Criterion 16

In the demo video below the configuration file is renamed to 'hide' it from the program such that the software wishes to regenerate this file upon startup. The software is then run so that it asks the user if they want to generate a new configuration file, at which point the user is prompted with an input field.

The video then proceeds to show two scenarios:

1. [0:12] The first where the user inputs an incorrect response every time they are prompted - 6 attempts - until the system exits since it can be determined that either something is wrong with the input field or the user may wish to not answer the prompt;
2. [0:32] The second where the user inputs an incorrect piece of data once and then inputs a correct piece of data, showing failing a process and then passing it on an alternative attempt does not cause any issues and allows the program to continue as if nothing happened. Following this up, the video then shows the file generated in response to this and how it is valid.

The second half of the video simply shows the deletion of this newly generated file and replacement of it with the old config file through renaming the original file back to `config.json` to show that this also allows for config files to be loaded in and out easily without causing any issues. This is not necessarily relevant for this criterion but is still a useful feature.



This clearly shows an example of the program reattempting a failed process multiple times until it has either succeeded or failed enough times for the program to expect it not to succeed without user intervention. This demo clearly passes this criterion and since most other parts of the program that can fail in such a way in which the program can handle them itself are also given this treatment then the criterion is met.

- (!) "Failures that the program cannot handle itself" is in reference to errors such as memory issues, the system the program runs on losing power or anything that is outside of what is controllable through VLang.

Criterion 24

Since the initialisation of the website on 'Vercel' (a website hosting service) in cycle 1 of the development, the web portal has never crashed publicly. This is due to using a 'linting' service 'Eslint' which scans all the code for the site before it is pushed to the public server and does not allow the website to update unless it is all valid and passes all required tests.

Status 3/5				
Production	monochain-webportal-8qafzh0cr-alfieran.ver...	Error 44s	updated pub/priv ↳ master	116d ago by AlfieRan
Production	monochain-webportal-kjl64x6yp-alfieran.ver...	Error 52s	remove private: false from package.json ↳ master	207d ago by AlfieRan
Production	monochain-webportal-bfrukc9k0-alfieran.ver...	Error 6s	changes readme ↳ master	207d ago by AlfieRan

Failed build logs on Vercel

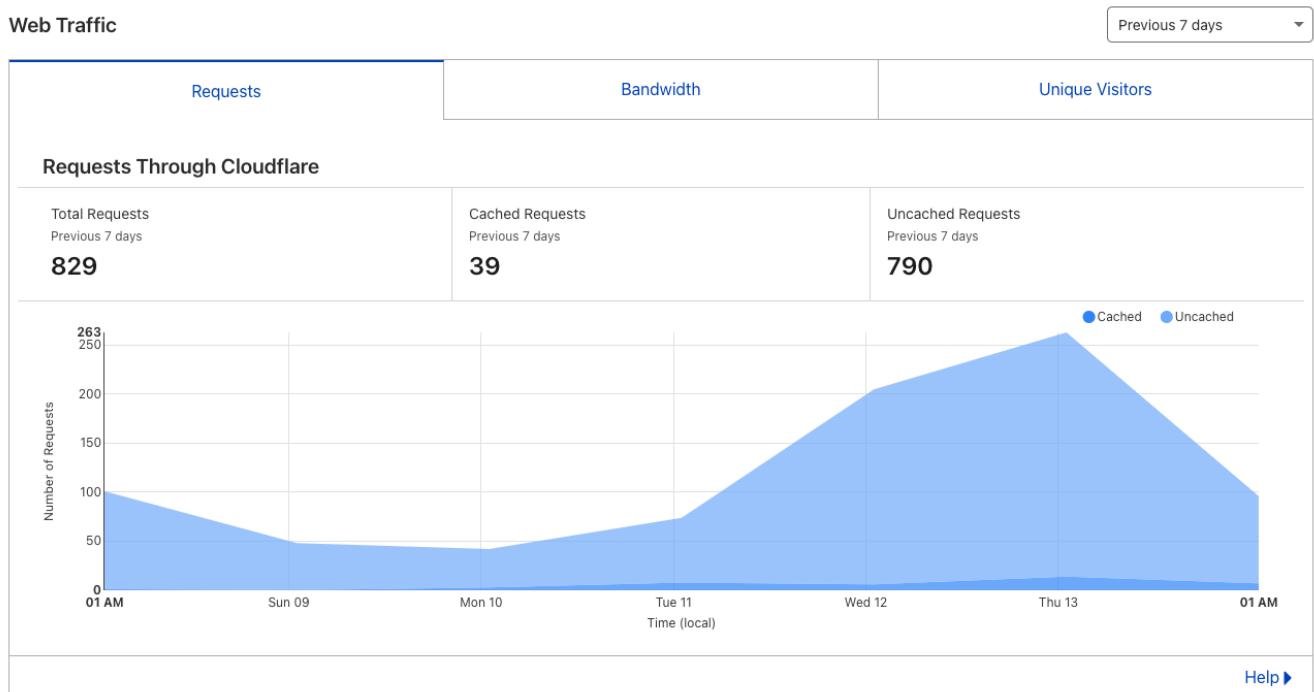
This can be seen above through the shown 'failed' deployments which were caught by this 'linting' service and prevents them from being pushed to the public website server. For reference I have also included the past 7 build deployments, all of which have passed and were then pushed to the public server.

Status 4/5				
Production (Current)	monochain-webportal-j2wp8fn2w-alfieran.ver...	Ready 37s	added more tests ↳ master	19d ago by AlfieRan
Production	monochain-webportal-ilb2qdi1m-alfieran.ver...	Ready 36s	program ↳ master	20d ago by AlfieRan
Production	monochain-webportal-a6mswie0w-alfieran.v...	Ready 30s	added some unit tests ↳ master	21d ago by AlfieRan
Production	monochain-webportal-23ystf21i-alfieran.ver...	Ready 31s	removed http handshake from private nodes ↳ master	25d ago by AlfieRan
Production	monochain-webportal-6k8mkql7w-alfieran.v...	Ready 31s	new node version ↳ master	25d ago by AlfieRan
Production	monochain-webportal-abhfthava-alfieran.ver...	Ready 30s	docker W ↳ master	25d ago by AlfieRan
Production	monochain-webportal-btun8angi-alfieran.ver...	Ready 31s	more logging ↳ master	26d ago by AlfieRan

To prove this I have included a video below of me attempting to crash the webportal by interacting with it, which obviously doesn't phase the site.



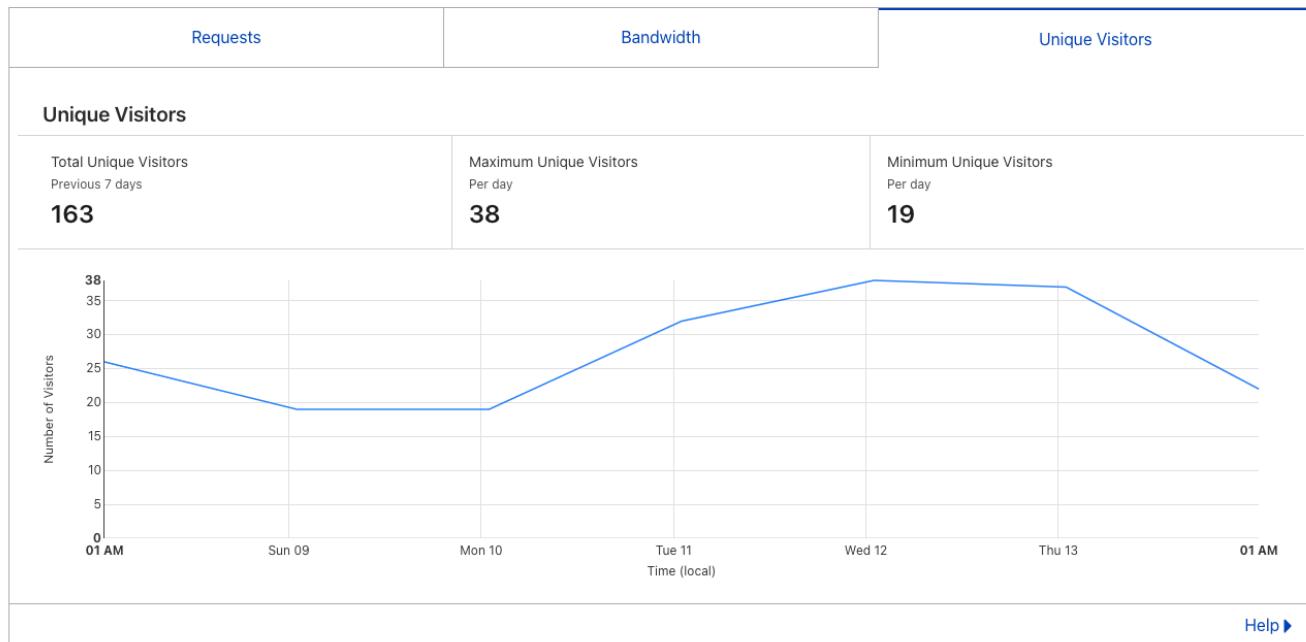
The video below contains flashing images.



A graph of the requests the webportal has received in the past 7 days.

Web Traffic

Previous 7 days ▾



A graph of the unique visitors the webportal has received in the past 7 days

To prove that the webportal has never crashed I have also included some of the traffic data over the last 7 days to show that it has a steady stream of users visiting the site and interacting with it.

All of this evidence shows that the webportal has never crashed past an extent that couldn't be solved simply by refreshing the page and hence it has passed this criterion.

3.2 Usability Testing

Criteria To Assess

To assess the usability of the project, I have put together a survey to get information on the user's experience whilst using the project. This survey is being used to assess the 5 key points:

Test	Test Details
1 - Effective	<p>Do you understand the idea of the project?</p> <p>Do you understand the different parts of the project?</p> <p>Do you know how to setup the node software?</p> <p>Did you know how to use the website?</p>
2 - Efficient	<p>Is the website quick and easy to navigate through</p> <p>Does the configuration file contain all needed settings?</p>
3 - Engaging	<p>Did you like using the project?</p> <p>Did you like the style of the website?</p> <p>Was the messaging demo engaging?</p>
4 - Error Tolerant	<p>Did any parts of the project crash?</p> <p>Have you found any errors/bugs in the node software?</p> <p>If so, what are they?</p>
5 - Easy to Learn	<p>Did you know how to use the messaging demo without any help?</p> <p>Did you know how to setup a node?</p> <p>Did you know how to use tokens to get access to the dashboard?</p>

Listed below are the results to the questions and why these questions were important, then what the responses to these questions actually means.

(i) The survey can be found here:

[Google Form Link](#)

User Feedback

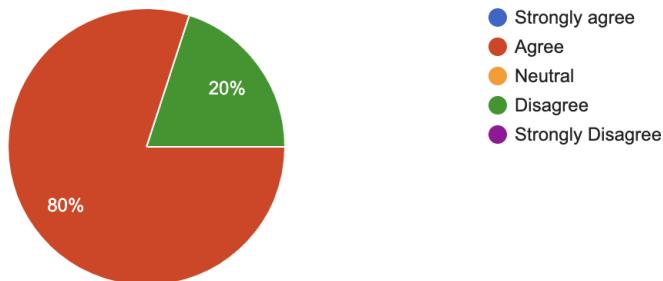
Effective

Do you understand the idea of the project?

Do you understand the idea of the project?

 Copy

5 responses



This aim of this question was to ensure that the website/webportal explained the project quickly and easily enough that the users that looked at it understood the idea of project without any further explanation.

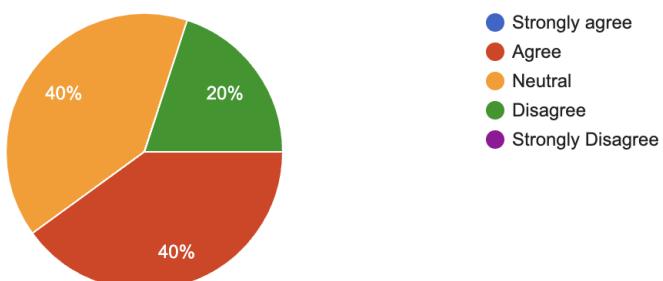
As shown it appears that most users (80%) understood the point of the project in some reasonable amount, however, none of the users seemed to understand the project in great detail and one user responded with a disagreement to say that they did not understand the project. This means that the idea of the project was conveyed to some positive extent using the webportal but was not clear enough and needs some more work.

Do you understand the different parts of the project?

Do you understand the different parts of the project?

 Copy

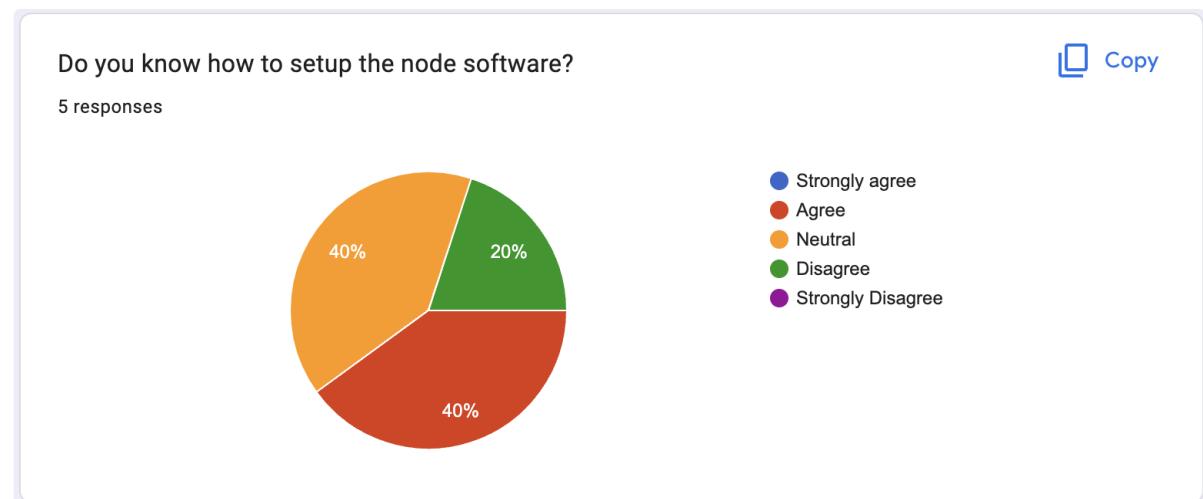
5 responses



Following on from the previous question, this shows that although 80% of the users understand the idea of the project on the whole, once asked about if they understand all the different parts of

the project, that number drops significantly. This shows that the separate sections of the project are not explained in enough detail and this needs to be expanded so that all users understand the very basic information about all parts of the project and understand which parts are aimed at them and which aren't.

Do you know how to setup the node software?

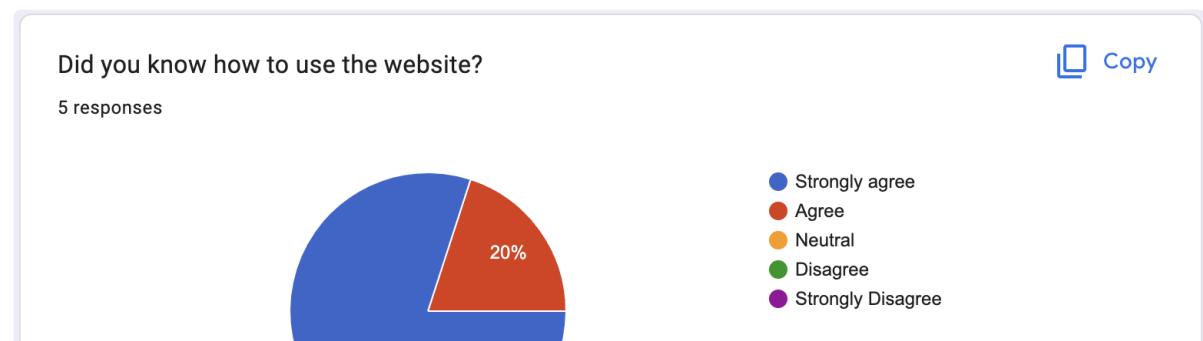


This question was important because the node software is the more technical side of the project (as shown in the [design frame](#)) and hence it is important that technical users understand how to setup and use the node software where as non-technical users do not need to know how to do this. Having said that, the lower the entry requirements to all parts of the project, the better, hence it is still better to have as many users as possible understand how to setup and use a node.

As shown in the data 40% of the users agreed to knowing how to setup the node software, with 40% being "neutral" which likely means they figured it out but with the help of other users or myself, then the final 20% of users disagreeing - showing that they didn't know how to do it. This roughly equates to the estimation of half of the users being more technical and half being more casual.

Interestingly the results for this question are also identical to that of the previous question, showing the suggestion of a correlation between users that understand that separate sections of the project and those that understand how to setup and use the node software.

Did you know how to use the website





80%

This question shows that all of the users either agreed or strongly agreed that they knew how to use the website. This is very important since this part of the project is the key part that all of the users needed to know how to use and understand since it acts as the entry point to every other part of the project.

Hence this strong of a response shows that the website was very effective and easy to use and as such was successful in its goal to be an effective entry point to the project.

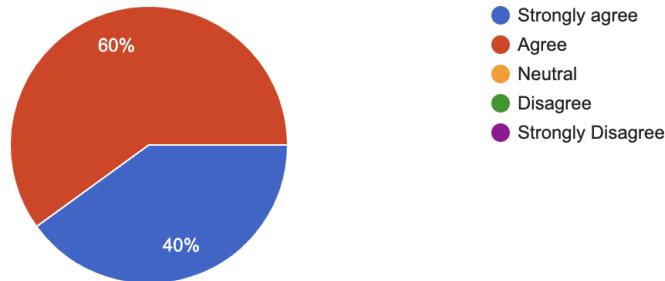
Efficient

Is the website quick and easy to navigate through?

Is the website quick and easy to navigate through?

 Copy

5 responses



Since the website is important for all uses of the project, it is important that the website is quick and easy to navigate through so that users can get to any part of the project or information that they want to get to as efficiently as possible.

As shown in the results this was a success, since 40% of the users strongly agreed and 60% of the users agreed meaning that all of the users agreed to some extent that the website is quick and easy to navigate through.

Does the configuration file contain all needed settings?

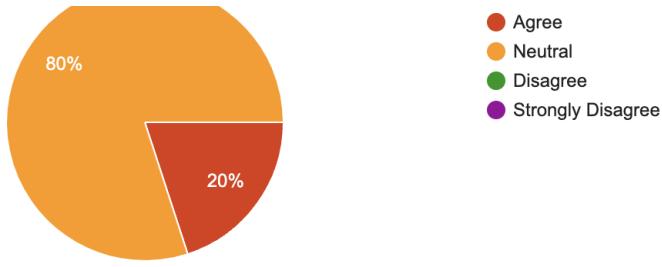
Does the configuration file contain all needed settings?

 Copy

5 responses



Strongly agree



This is a very specific question aimed at more technical users, since it is not only about the node software, which is used by technical users but also about custom configurations related to the software. This is shown clearly in the results as one of the users agreed the configuration file contained all settings they needed, yet the other 4 responded with a "neutral" meaning that they likely either didn't know how to use the configuration file or didn't need to use it.

This is both a success and a failure, since the configuration file should only be needed by very technical users and such most users shouldn't need to use the file and if they do it should contain all the settings they need which is shown by the responses. However, ideally most of the users would know of the configuration file and know how to use it if they need to because as with the rest of the project, the lower the requirements to use any part of the project, the better.

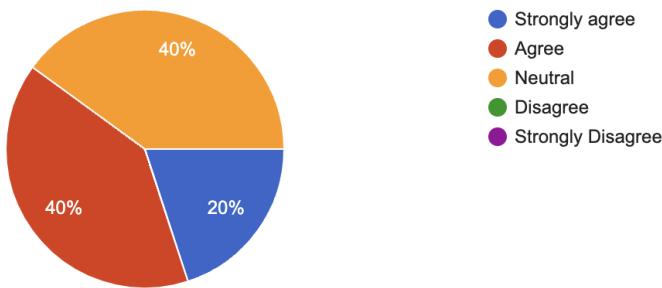
Engaging

Did you like using the project?

Did you like using the project?

Copy

5 responses

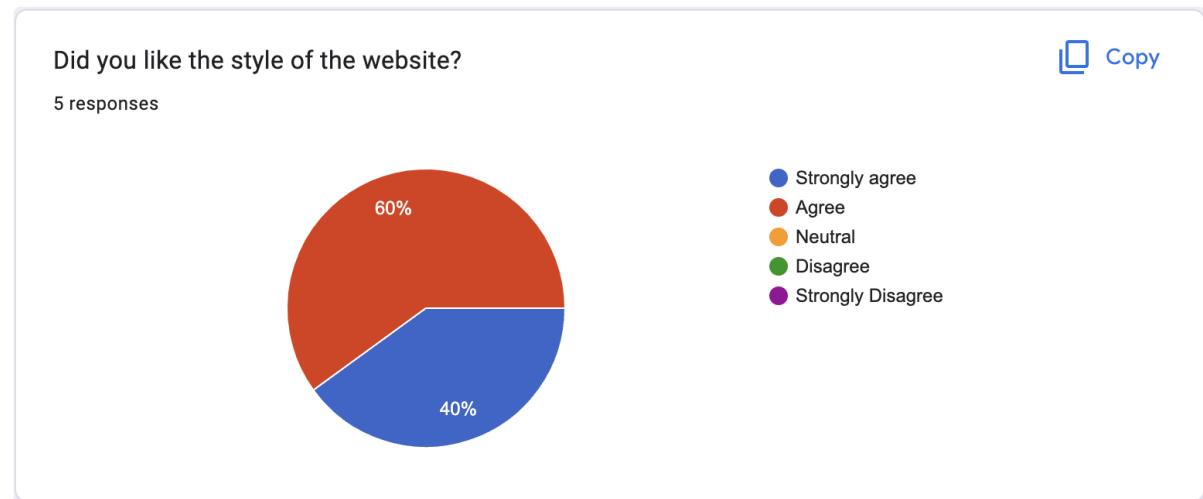


It is important that users like using the project, as with a project based upon a decentralised network such as this one, the project gets faster and more secure the more users it has. This makes it more enticing to new users so that they then join the project and it continues to scale well.

The responses to this question somewhat positively reflect this, with 60% of the users agreeing or strongly agreeing to liking using the project - which is good - however 40% of the users responded "neutral" to this question meaning that although they didn't dislike using the project

they didn't like using it either. Ideally all of the users would at least agree to liking using the project with a decent proportion strongly agreeing to liking using the project.

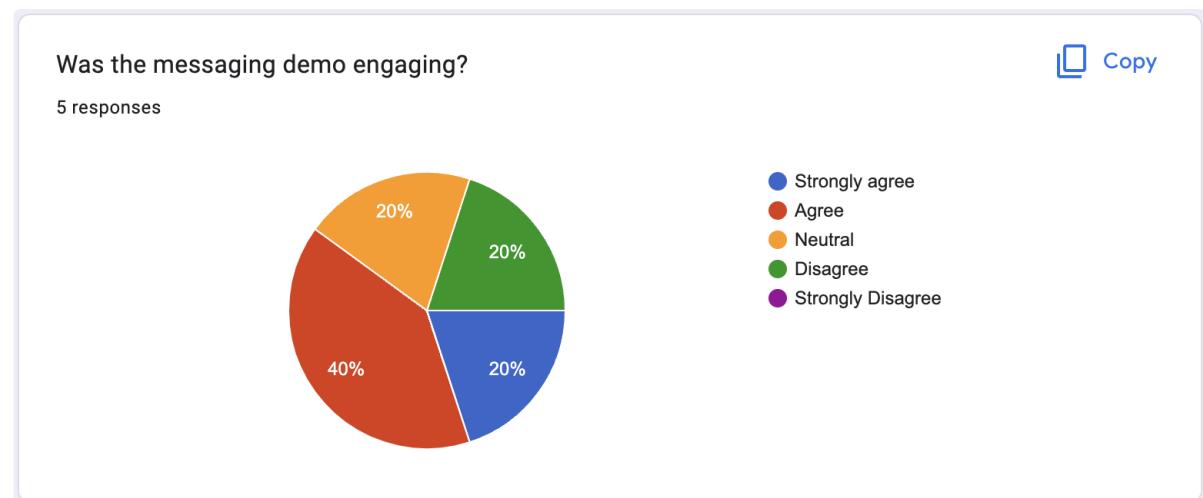
Did you like the style of the website?



The style of the website is important since a well styled website makes users want to look around and learn about the project, whereas a poorly styled website can scare users away and make them assume that the whole project is old, outdated and badly made.

Therefore it is very positive that all of the users agreed to liking the style of the website, with 40% of them strongly agreeing. Ideally all of the users would strongly agree to liking the style of the website because then they would be more likely to look around and visit the website just for the style and learn about the project in the progress, however all of the users agreeing to liking the style of the website is a positive since it means that users are unlikely to leave or associate the project negatively purely due to the website's look and appearance.

Was the messaging demo engaging?



The messaging demo was the demo created during cycle 12 to show the networking capabilities of the project and was the most up to date example of what the node software could do whilst

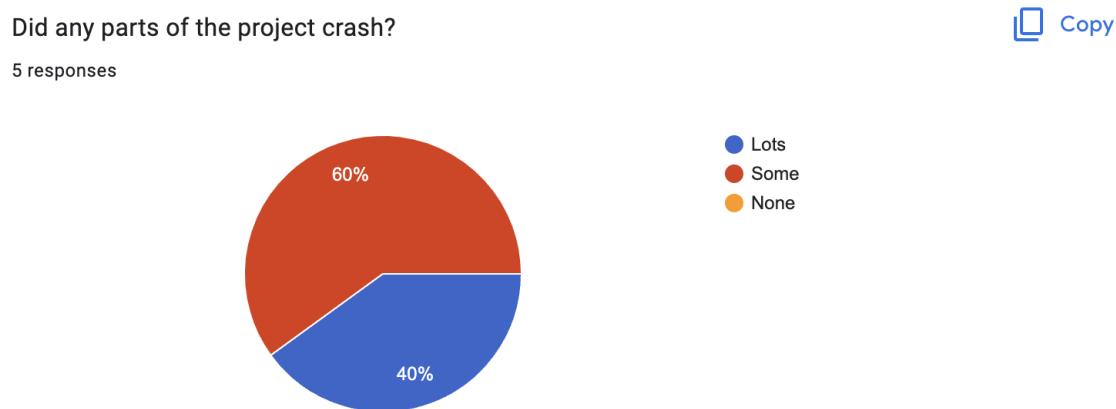
having low requirements for the users. It is important that this demo was engaging since it represents the kind of program that can be made upon the top of the decentralised networking technology and if the programs built on top of this project aren't engaging then the project itself is not engaging, which is obviously not good.

Bearing this in mind the responses to this question are somewhat positive, since 60% of the users agreed to the messaging demo being engaging, meaning that the majority of the users did indeed find it engaging. However, 40% of the users either "disagreed" or responded "neutral" which means that they either barely found it engaging or didn't find it engaging; ideally all of the users would find the demo engaging so these results are not amazing but the majority still did which is good.

It is also worth remembering here that this demo did not represent all the functionality that the project originally intended to include and hence has a long way to progress in the future.

Error Tolerant

Did any parts of the project crash?



The project should not crash and hence the results for this question should be as many "None" responses, however, as shown in the responses this was not the case - 60% of the users responded with "some" parts of the project crashed and 40% of users responded with "Lots". This is not of the ideal scenario, which would be 100% responses of "None"

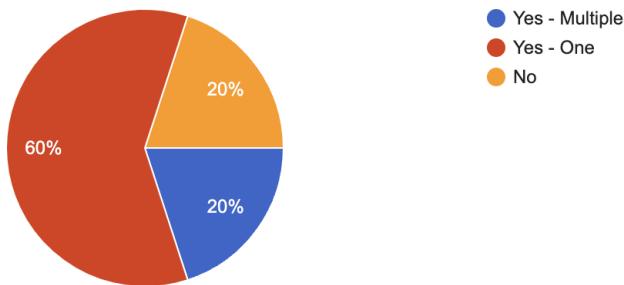
There is a good reason for this however, which will be talked about in the evaluation but essentially comes down to a bad choice of programming language for the node software paired with not enough time management given to fully polish the project to prevent crashes such as the ones the users experienced.

Have you found any errors/bugs in the node software?

Have you found any errors/bugs in the node software?

 Copy

5 responses



This leads on from the previous question asking about if any parts of the project crashed and shows that the majority of the errors and crashes in the project were due to the node software and since the node software was the only part of the project that I encountered crashes, I expect that the one user to respond "No" to finding any errors/bugs likely had the software crash but did not understand the error/bug.

However, since the user that gave the "No" response did not expand their answer in the following question "If so, what are they?", these results shall be taken at face value. These results then show that most users found a bug in the software, with one user stating they found multiple bugs, hence it can be stated that the majority of users found some form of error or bug in the node software and therefore the program was not as error tolerant as it would be ideally.

If so, what are they?

If so, what are they?

4 responses

messaging demo crashed :(

crashes when typing many messages, unclear error message when disconnected

Crashed before I was able to send any messages

The messages did not send if you spammed the messages.

These responses show that the same part of the project crashed for all the users which encountered bugs, this is good because it means that there is a clear section of the program that will need to be tested and fixed before the development of the project continues hence there is a clear direction that can be headed in to fix these issues. The downside to this however, is that this means that this part of the project crashed regularly (80% of the users reported bugs related to this demo) and since this demo relies upon lots of different modules and sections within the project on the whole it is likely to be underlying issues rather than issues just due to how the demo was made.

I have a strong suspicion that these underlying issues due in fact exist and in part due to my language choice for the node software, although better code on my half could likely decrease the quantity and severity of these issues, and I will dissect this in the evaluation of this project.

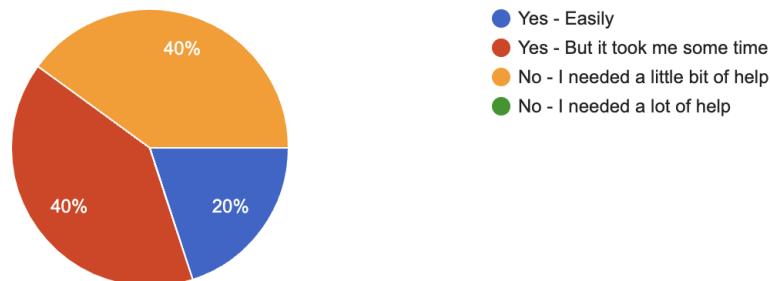
Easy to Learn

Did you know how to use the messaging demo without any help?

Did you know how to use the messaging demo without any help?

 Copy

5 responses



It's important that users know how to use the messaging demo without any help, since this represents the approximate level of complexity of most of the programs which could be built on top of the project and hence if users can understand how to use this demo without any help this is a good sign that they could use other programs built on top of the project later on (such as the wallet, transaction tool and marketplace that were originally planned).

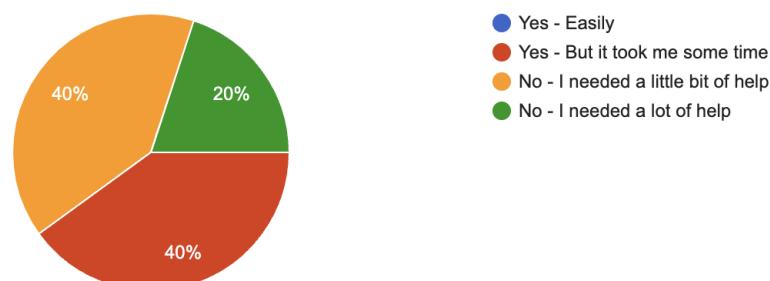
Therefore 60% of the users knowing how to use the demo without help is good, however since 40% of the users still did not figure it out, it would likely be good to create a tutorial/guide on how to use the demo in the future.

Did you know how to setup a node?

Did you know how to setup a node?

 Copy

5 responses



This is an important part of the project and although the node software is aimed at more technical developers than standard users, the more users that know how to setup a node, the better.

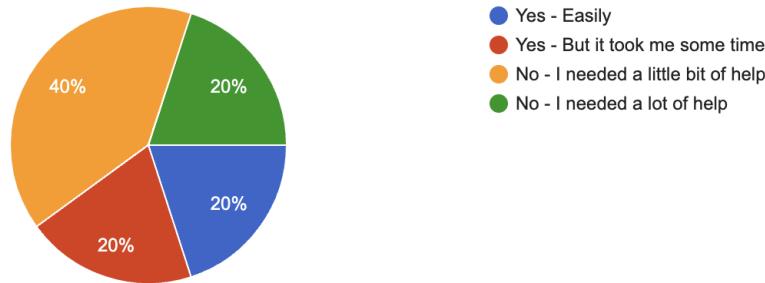
The responses to this question show that all users struggled with this process and either needed additional help or time to figure it out on their own. 40% of the users eventually figured it out without additional help and 60% of the users need some extent of additional help. This shows that this was not a success and some form of solution needs to be introduced to make it easier for users to setup nodes in the future.

Did you know how to use tokens to get access to the dashboard?

Did you know how to use tokens to get access to the dashboard?

 Copy

5 responses



This question was regarding the process of clicking the "get token" button on the dashboard login page, then copy and pasting the token code generated from the node's software's logs into the page and submitting it as proof that the user has control over that node.

I assumed this was fairly simply to do and hence users were not given any support on the page itself explaining how to complete this process, but as shown in the results for this question that was not the case. 40% of users figured out how to complete this on their own and 60% required some form of help from myself. This means that the login page needs some kind of instructions or guide to explain to users how to use the token access and luckily this isn't a very complex addition to the project so could be done fairly easily.

Usability Requirements in Success Criteria

Some of the success criteria requirements fall into the section of usability. These have been covered either in the initial survey or will be covered in the additional survey I created to go over these points in more detail.

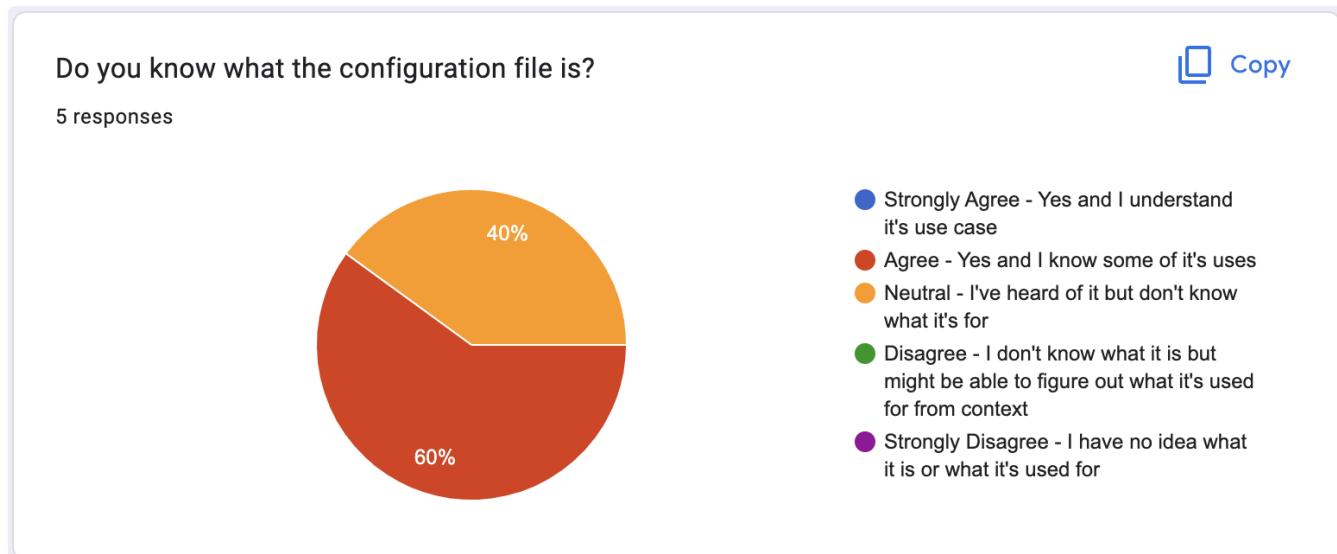
(i) The additional Survey is available here: [Google Form](#).

Criterion	Description
11	The configuration handler should be accessible and useable to technical users without any help, all users should be aware of its existence and able to use it with some external assistance.
13	The node dashboard should be easily accessible and usable by all forms of users.
21	Non-technical users must be able to identify what the point/idea of the project is just from the homepage of the website.
22	Users should be able to get to the majority of what they would want to get to within 3 clicks.
23	The web-portal should be available and working on a variety of device sizes including mobile and desktop.

Criterion 11

(i) The configuration handler should be accessible and useable to technical users without any help, all users should be aware of its existence and able to use it with some external assistance.

In order to assess this criterion I included the following three questions in the survey:

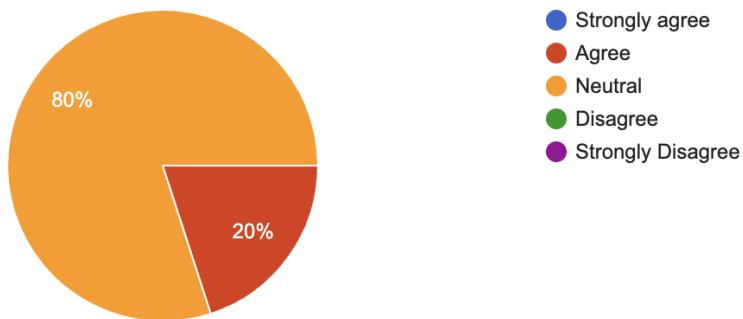


This question assess what proportion of the users are actually aware of the configuration handler/file and as shown in the results above, this was met somewhat successfully with 60% of users being aware of it and knowing some of its uses and the other 40% being aware of it existing but not knowing what it's for. This is evidence that all users are aware of its existence.

Does the configuration file contain all needed settings?

 Copy

5 responses

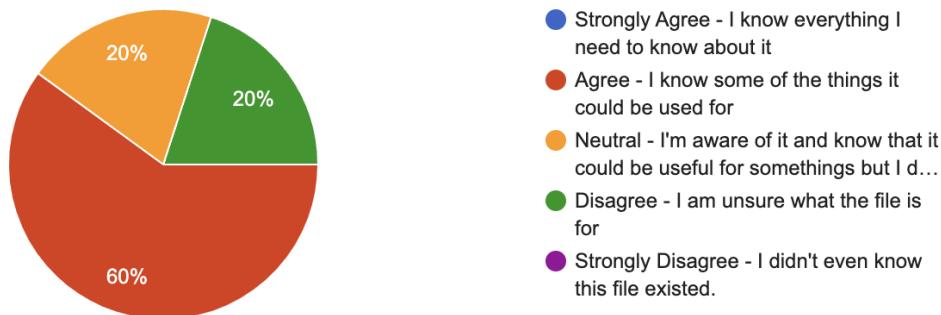


This question shows that the majority of users (80%) likely aren't aware of all the settings within the configuration file as they responded with "neutral" to it containing all needed settings. The reason this assumption can be made is due to users responding "neutral" means that they did not agree or disagree

Do you know what the configuration file is for?

 Copy

5 responses



Knowing what the configuration file is for is another way of ensuring that users are aware of its existence and could use it if they were given external help. This question shows that: the majority, 60%, of users would be capable of doing that; with 20% of users being aware of its existence but they would probably require additional external assistance to explain to them what they can use the file for before in more detail; then the final 20% having no idea what the file can be used for and hence would need a lot of external assistance to do anything using it.

Combining the results from all three questions and it can be concluded that all users had at least heard of the configuration file existing, with the majority knowing what it could be used for and some of the more technical users within that pool knowing how to use it without additional help but the majority of users requiring assistance. This meets the criterion's requirements, although if user testing were to continue past this point it would ideally contain a separate standard and technical user group so as to understand if the different stakeholders were able to do what it was aimed for them to be able to do.

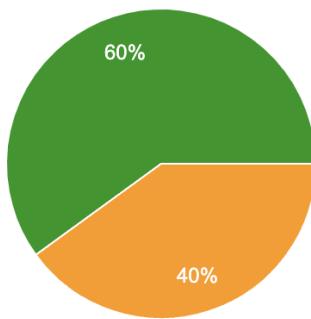
Criterion 13

i The node dashboard should be easily accessible and usable by all forms of users.

Do you know what the node dashboard is?

 Copy

5 responses

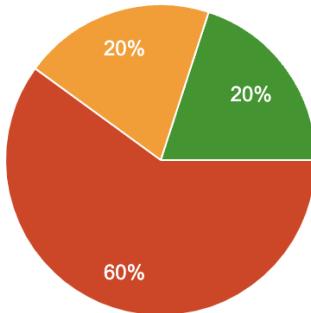


- Strongly Agree - Yes and I understand it's use case
- Agree - Yes and I know some of its uses
- Neutral - I've heard of it but don't know what it's for
- Disagree - I don't know what it is but might be able to figure out what it's used for from context
- Strongly Disagree - I have no idea what it is or what it's used for

Do you know how to access the node dashboard?

 Copy

5 responses



- Strongly Agree - I could navigate to it easily
- Agree - I could navigate to it if someone else started the node for me (like in th...
- Neutral - If shown how to get to the dashboard I could probably get back t...
- Disagree - I'm not sure how to access the node dashboard and would need...
- Strongly Disagree - I would need a lot of help to access it

Interestingly, more users agreed to knowing how to navigate to the node dashboard than the amount that knowing what the dashboard was. This suggests that most users were a bit confused by the questions and thus they needed more clarification, after talking to users after completing the survey it was revealed that the majority of users didn't realise that the messaging demo was also the dashboard and after learning this said they understood what it was much better.

Hence this criterion was given somewhat inaccurate results due to the wording of the questions, although it could also be argued that the messaging demo being the dashboard could've been made more clear on the site itself, which is a valid criticism.

Therefore based upon the fact that most (60%) of users knew how to access the dashboard and after speaking to them after this survey they understood what it was very quickly I deem this criterion partially met, since the results are along the right lines as what is required but they need to be furthered on first.

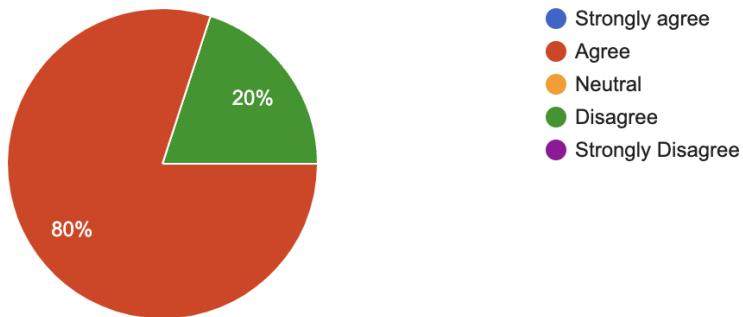
Criterion 21

- i** Non-technical users must be able to identify what the point/idea of the project is just from the homepage of the website.

Do you understand the idea of the project?

 Copy

5 responses



Since each user that completed this survey used the node messaging demo and were given access to the website without any instructions except to "look around and get a feel for the site" before completing the survey, and the node software does not give any explicit explanations for the reason and ideology of the project, it stands that the users will have gotten most of their understandings on the project itself from the website and its homepage.

Therefore as 80% of the users that completed this survey agreed to understanding the idea of the project without any additional help except the website it can be stated that this shows evidence of this criterion being met.

Although with 20% of the users disagreeing and no users strongly agreeing, this likely means that the homepage needs some more work in order to make it even more clear what the idea of the project is, however this can likely be pushed into a slightly more long term fix since it does still pass the criterion.

Criterion 22

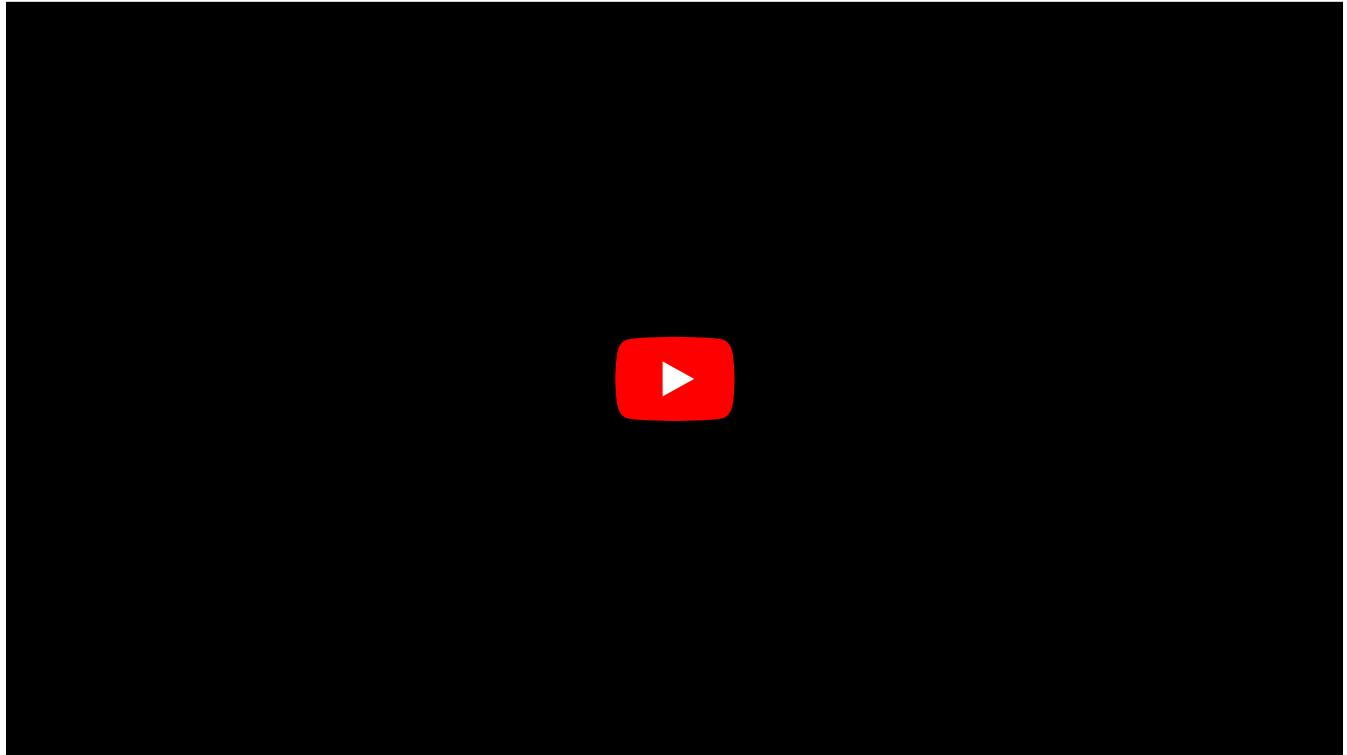
- i** Users should be able to get to the majority of what they would want to get to within 3 clicks.



This shows that All parts of the website can be travelled to within 2 clicks from the home page, hence including the singular click that it takes to get back to the home page, each major page can be accessed from any other major page within 3 clicks. Therefore this shows that this success criterion has been met.

Criterion 23

- (i) The web-portal should be available and working on a variety of device sizes including mobile and desktop.



This device demo shows the website being tested on a large variety of devices, including a standard computer, tablets and phones. Throughout this testing it is shown that the criterion is partially met, as although all parts of the website are functional on all devices and there are no crashes and hence it is available, some parts of the site have visual glitches or bugs that cause the graphics to be partially off the screen or the wrong size. This leads to some text not being readable in specific scenarios which means that in those scenarios I would not classify the website as completely working.

Hence, this success criterion has been partially met and would require some additional tuning on those very specific device sizes to be met, which could be done fairly easily and hence would be good to do in the future.

3.3 Checking Development Tests

The purpose of this section is to go back over all the tests performed in each cycle of development and ensure that the project still passes the tests.

However, there is one new addition since the development tests: the addition of automated tests which allow for some of the tests to be done by the code itself so as to be able to test the majority of the key functions individually without having to call the functions in an isolated state. These tests are written on a module by module basis and do not cover the entire code base - such as the server module which is too complicated to automate tests for without building the module around the tests in the first place - therefore I have added a "type of test" section which specifies whether or not the tests are automated with a test program or done manually by myself.

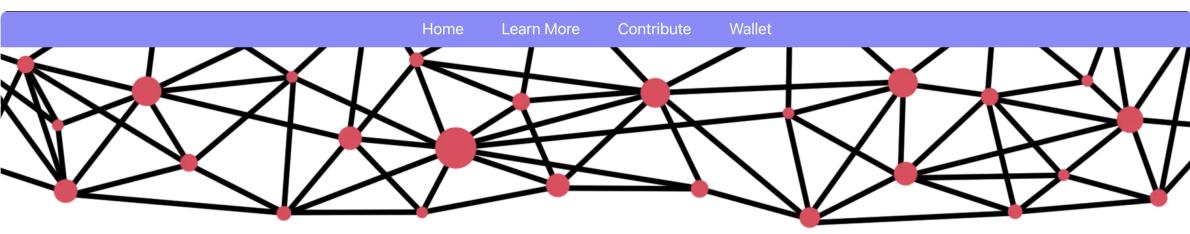
Cycle 1

Test Table

Test	Instructions	Expected Result
1	Run the webportal on localhost.	Webportal launches and some text is displayed.
2	Deploy the web-portal by git pushing to Github.	Vercel should automatically detect the changes and deploy the new version.
3	Run the node program	Output some kind of welcome message.
4	Download the node program by clicking the download button on the website.	The Node program should be downloaded to the test computer

Evidence

Test 1



Welcome to the MonoChain

The MonoChain is a new generation of decentralised computing built alongside the "Proof of Trust" protocol.

What is Proof of Trust?

'Proof of Trust' is a custom built consensus protocol that uses a 'trust' parameter in order to be faster and eco-friendlier than 'Proof of Work' whilst having much lower entry requirements than



I trust you,
so I'll process
this data for you!

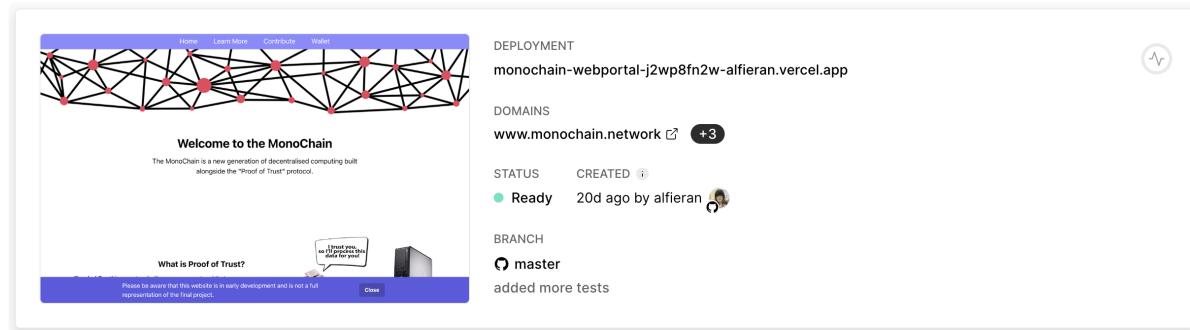


Please be aware that this website is in early development and is not a full representation of the final project.

Close

The webportal running locally.

Test 2



Vercel automatically deploying the newest version of the git repo

Test 3

```
○ alfier@Alfies-MacBook-Pro ~ % yarn start
yarn run v1.22.19
$ v run ./src/
[Main] ***** MonoChain Mining Software *****
[Main] Getting configuration settings...
[utils] Reading file: ./monochain/config.json
```

The node software running

Test 4

A screenshot of the MonoChain download page. At the top, there's a blue navigation bar with links for Home, Learn More, Contribute, and Wallet. Below the navigation, the main title is "Mine for the Monochain Network". Underneath, there's a section titled "Download a Node". It contains a paragraph about cloning the git repository and compiling it. Two buttons are shown: "Download Node Software" and "Clone the Repo". Further down, there are two tabs: "Create the Config File" and "Editing the Config File". The "Create the Config File" tab is active, showing a "Setup a Node" section with instructions. It includes a terminal-like window showing command-line output related to setting up a node. The "Editing the Config File" tab is also visible.

The download page of the webportal

Cycle 2

Test Table

Test	Instructions	Expected Result
1	Run "v --version" in the terminal.	Output of "V" then some version number and a hash of that upda
2	Run "yarn start" in the node package's source folder.	The node software to be compil and ran.
3	Run "yarn build" in the node package's source folder.	The node software to be compil zipped and put into the webport public folder.
4	Same as test 2.	Node software outputs a welcor message.
5	Same as test 2.	Node's web api should launch leading to "http://localhost:8000" displaying a welcome message
6	Navigate to " https://monochain.network/download "	Be shown download instruction:
7	Download and unzip the node software.	Should download and unzip to an executable file.
8	Run the downloaded node software on an arm based macOS device.	Should run as tested in test 4 ar

Evidence

Test 1

```
● alfier@Alfies-MacBook-Pro ~ % v --version
V 0.3.1 5c716af
```

Test 2

```

○ alfier@Alfies-MacBook-Pro node % yarn start
yarn run v1.22.19
$ v run ./src/
[Main] ***** MonoChain Mining Software *****
[Main] Getting configuration settings...
[utils] Reading file: ./monochain/config.json

[config] Config Loaded...
[Main] Starting database on a docker container, to change this edit your config...
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Server] Connecting to database...
[Server] Database is external: false
[Server] Database host: localhost
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Websockets] Node is private, starting client...
[Websockets] Created client.
[Server] Starting http server
[Server] Initiating entry point to network
[Server] Private node, connecting to ws://nano:8001 using ws
[Websockets] Connecting to server ws://nano:8001

```

Test 3

```

● alfier@Alfies-MacBook-Pro node % yarn build
yarn run v1.22.19
$ yarn clean && yarn clean-zip && yarn compile && yarn zip && echo "zipped node"
$ rm -rf dist/code && mkdir dist/code && echo "cleaned dist/code"
cleaned dist/code
$ rm -f node.zip && rm -f ../webportal/public/node.zip && echo "cleaned node.zip"
cleaned node.zip
$ yarn compile-native
$ v ./src/ && mv ./src/src ./dist/code/node-native && echo "compiled for the native OS of this system"
compiled for the native OS of this system
$ zip ../webportal/public/node.zip -9r ./dist/ && echo "zipped"
adding: dist/ (stored 0%)
adding: dist/ExitCodes.txt (deflated 51%)
adding: dist/.DS_Store (deflated 94%)
adding: dist/code/ (stored 0%)
adding: dist/code/node-native (deflated 61%)
adding: dist/README.txt (deflated 48%)
zipped
zipped node
:+ Done in 2.46s.

```

Test 4

 See evidence for test 2.

Test 5



Test 6

Home Learn More Contribute Wallet

Mine for the Monochain Network

Download a Node

This is an example node (mining) software written in Vlang, it is recommended that you clone the git repository and compile it yourself so that you can edit any pieces of code you wish to and to ensure that it runs properly on your system as V compiles to an executable that will be different for different systems.

The below zip file contains an arm64 macOS executable called 'node-native' (native because that's the native OS for the laptop I do the development for this project on) and an x86 windows executable called 'node-windows'. If you don't use either of these systems then you must clone the git repository and use the v compiler to compile the project yourself.

[Download Node Software](#) [Clone the Repo](#)

Create the Config File

Once you've downloaded the node software or compiled it directly, you will need to setup a configuration file. The simplest way of doing this is by first running the program once with no config file. The program will then ask you if you wish to generate a new config file which can be accepted by typing 'y' and then pressing enter. This will generate the file under the path './monochain/config.json' in the same directory as the executable. Assuming you also do not have a keypair file, the program will then ask if you would like to generate a new keypair file which can be accepted by typing 'y' and then pressing enter. This will generate under the path './monochain/keys.config' and can then be replaced with your own keypair if you already have one.



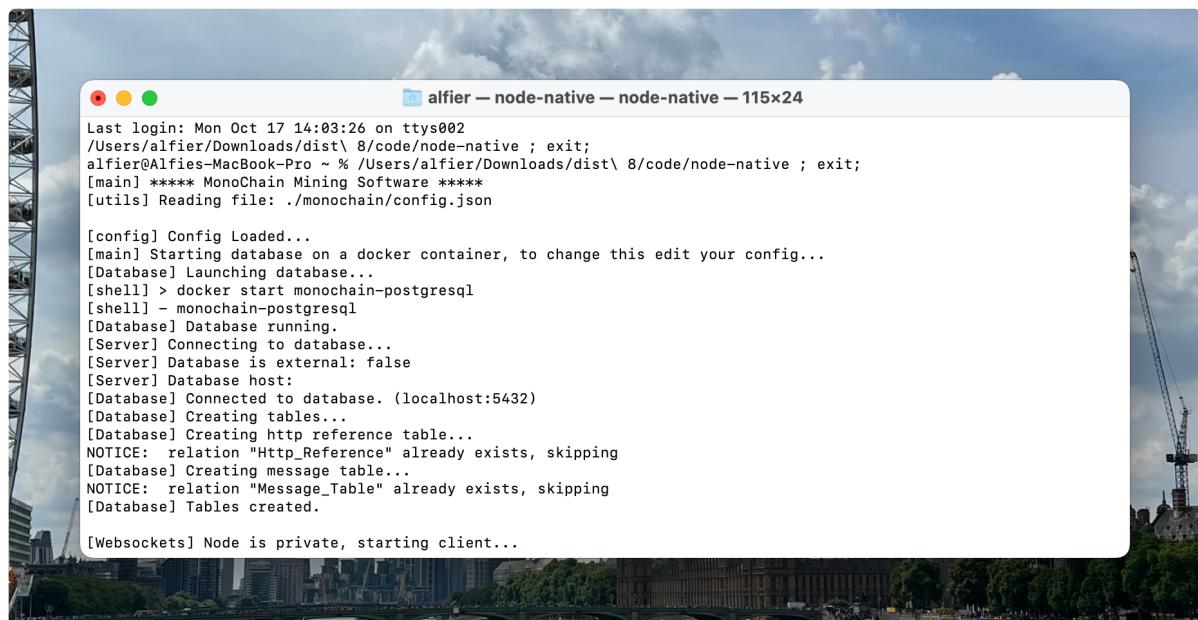
Editing the Config File

Setup a Node

Test 7

dist 8	Unzipped folder	-- Folder	Today at 13:59
README.txt		959 bytes Plain Text	Today at 13:59
code		-- Folder	Today at 13:59
node-native	Executable	2.7 MB Unix Executable File	Today at 13:59
ExitCodes.txt		2 KB Plain Text	Today at 13:59
node (8).zip	Zip file	1.1 MB ZIP archive	Today at 13:59

Test 8



Cycle 3

Test Table

Test	Instructions	Expected Result	Type Test
1	Create a new configuration file using "create_configuration()"	Outputs a log to the console confirming the operation is commencing and a config file should be generated.	Au
2	Load the new config file using "load_config()"	Config file should be loaded and returned.	Au
3	Load the new config file using "get_config()"	Config file should be loaded and returned.	Au
4	[Additional Test] Save and then load a configuration.	The data saved and then loaded back in should be exactly the same.	Au

Test Code

Test 1

```
const test_config_path = "./test_config.json"

fn test_config_creation() {
    println("[Testing - Configuration] Running test: config_creation")
    config := configuration.create_configuration(test_config_path)
    assert true // if code gets to this point, the value has to be a valid config
}
```

Test 2

```
const test_config_path = "./test_config.json"

fn test_config_load() {
    println("[Testing - Configuration] Running test: config_load")
    config := configuration.load_config(test_config_path)
    assert true // if code gets to this point, the value has to be a valid config
}
```

Test 3

```
fn test_get_config() {
    get_config := configuration.get_config()
    assert true
}
```

Test 4

```
const test_config_path = "./test_config.json"

fn test_config_save_then_load() {
    println("[Testing - Configuration] Running test: config_save_then_load")
    config := configuration.create_configuration(test_config_path)
    configuration.save_config(config, 0, test_config_path)
    loaded_config := configuration.load_config(test_config_path)
    assert config == loaded_config
}
```

Evidence

Test 1

```
[Testing - Configuration] Running test: config_creation
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[config] Sucessfully saved configuration file.
[config] A default configuration file has been created at ./monochain/config.json
Please edit it according to https://monochain.network/download/
Then relaunch the program.
OK [1/4] 1.803 ms 1 assert | main.test_config_creation()
```

Test 2

```
[Testing - Configuration] Running test: config_load
[utils] Reading file: ./test_config.json
OK [2/4] 0.073 ms 1 assert | main.test_config_load()
```

Test 3

```
[utils] Reading file: ./monochain/config.json
[config] Config Loaded...
OK [4/4] 0.578 ms 1 assert | main.test_get_config()
```

Test 4

```
[Testing - Configuration] Running test: config_save_then_load
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[config] Sucessfully saved configuration file.
[config] A default configuration file has been created at ./monochain/config.json
Please edit it according to https://monochain.network/download/
Then relaunch the program.
[config] Sucessfully saved configuration file.
[utils] Reading file: ./test_config.json
OK [3/4] 0.543 ms 1 assert | main.test_config_save_then_load()
```

Cycle 4

Test Table

Test	Instructions	Expected Result
1	Compile for Windows from a MacOS device.	Node program to compile and execute normally.
2	Compile for Linux from a MacOS device.	Node program to compile and execute normally.
3	Run the Windows version on a non-windows system.	Computer running incorrect program should report some kind of error and close the program.

Evidence

Test 1

```
© alfier@Alfies-MacBook-Pro node % yarn compile-windows
yarn run v1.22.19
$ v ./.src/ --embed vlib --os windows && mv ./src/src.exe ./dist/code/node-windows.exe && echo "compiled for windows"
Cross compiling for Windows...
/tmp/v_501/src.8266642098488871673.tmp.c:2048:2: error: #error VERROR_MESSAGE Header file <arpa/inet.h>, needed for module `pg` was not found. Please install the corresponding development headers.
2048 | #error VERROR_MESSAGE Header file <arpa/inet.h>, needed for module `pg` was not found. Please install the corresponding development headers.
| ~~~~~
Cross compilation for Windows failed. Make sure you have mingw-w64 installed.
brew install mingw-w64
error: Command failed with exit code 1.
info: Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
● alfier@Alfies-MacBook-Pro node % brew install mingw-w64
Running 'brew update --auto-update...'
=> Auto-updated Homebrew!
Updated 3 taps (homebrew/core, homebrew/cask and homebrew/services).
=> New Formulae
cpuid      emqx      envd      osv-scanner      pdf-diff      retdec      skeema      stuffbin      syslog-ng      tproxy
bamboo-studio

You have 44 outdated formulae installed.
You can upgrade them with brew upgrade
or list them with brew outdated.

Warning: mingw-w64 10.0.0.3 is already installed and up-to-date.
To reinstall 10.0.0.3, run:
  brew reinstall mingw-w64
© alfier@Alfies-MacBook-Pro node % [ ]
```

Vlang reporting that I don't have a header file which I have since it is the wrong version

Test 2

```
© alfier@Alfies-MacBook-Pro node % v -os linux ./.src/
Cross compilation for Linux failed (first step, cc). Make sure you have clang installed.
builder error: /tmp/v_501/src.13014226172111377526.tmp.c:1597:2: error: VERROR_MESSAGE Header file <sys/syscall.h>, needed for module `crypto.rand` was not found. Please install the corresponding development headers.
#error VERROR_MESSAGE Header file <sys/syscall.h>, needed for module `crypto.rand` was not found. Please install the corresponding development headers.
^
/tmp/v_501/src.13014226172111377526.tmp.c:1705:2: error: VERROR_MESSAGE Header file <sys/un.h>, needed for module `net` was not found. Please install the corresponding development headers.
#error VERROR_MESSAGE Header file <sys/un.h>, needed for module `net` was not found. Please install the corresponding development headers.
^
/tmp/v_501/src.13014226172111377526.tmp.c:1759:2: error: VERROR_MESSAGE Header file <arpa/inet.h>, needed for module `net` was not found. Please install the corresponding development headers.
#error VERROR_MESSAGE Header file <arpa/inet.h>, needed for module `net` was not found. Please install the corresponding development headers.
^
/tmp/v_501/src.13014226172111377526.tmp.c:2632:20: error: use of undeclared identifier 'SYS_getrandom'
    int _t1 = syscall(SYS_getrandom, buffer, bytes_needed, 0);
4 errors generated.
```

A variety of errors occurred.

Test 3

```
✗ alfier@Alfies-MacBook-Pro % ./dist/code/node-windows.exe  
zsh: exec format error: ./dist/code/node-windows.exe
```

I ran this test on an old version of the software I still had on my laptop

Cycle 5

Test Table

Test	Instructions	Expected Result
1	Generate a key pair.	Keys to be generated and validated successfully without crashing.
2	Sign some random input data.	A signature to be generated without crashing.
3	Sign then validate some random input data.	A signature to be generated then validated successfully.

Automated Test Code

Test 1

```
pub fn test_gen_keys() {
    println("[testing] Testing Key Pair Generation")
    cryptography.gen_keys()
    // because the gen_keys function hasn't exited yet, we know that the keys are generated
    println("[testing] Key Pair Generation Test Passed")
    assert true
}
```

Test 2

```
pub fn test_sign() {
    println("[testing] Testing Signing")
    keys := cryptography.gen_keys()

    println("[testing] Generated a new key pair successfully")
    message := time.utcnow().str().bytes()

    keys.sign(message)
    println("[testing] Signed a message successfully")
    println("[testing] Signing Test Passed")

    // because the sign function hasn't exited yet, we know that the signature is valid
    assert true
}
```

Test 3

```
pub fn test_verify() {
    mut failed := false
    println("[testing] Testing Verification")
    keys := cryptography.gen_keys()

    println("[testing] Generated a new key pair successfully")
    message := time.utcnow().str().bytes()

    signature := keys.sign(message)
    println("[testing] Signed a message successfully")

    if cryptography.verify(keys.pub_key, message, signature) {
        println("[testing] Verified a signature successfully")
        println("[testing] Verification Test Passed")
        failed = false
    } else {
        println("[testing] Verification failed")
        println("[testing] Verification Test Failed")
        failed = true
    }

    assert failed == false
}
```

Evidence

Test 1

```
[testing] Testing Key Pair Generation
[cryptography] Signature verified
[testing] Key Pair Generation Test Passed
OK [1/3] 55.581 ms 11 asserts | main.test_gen_keys()
```

Test 2

```
[testing] Testing Signing
[cryptography] Signature verified
[testing] Generated a new key pair successfully
[testing] Signed a message successfully
[testing] Signing Test Passed
OK [2/3] 77.983 ms 17 asserts | main.test_sign()
```

Test 3

```
[testing] Testing Verification
[cryptography] Signature verified
[testing] Generated a new key pair successfully
[testing] Signed a message successfully
[testing] Verified a signature successfully
[testing] Verification Test Passed
OK      [3/3]    90.132 ms    19 asserts | main.test_verify()
```

Cycle 6

- (i) I've changed the reference of the node that will be used in the test from "https://nano.monochain.network" to "http://nano:8000", this won't affect the actual test and just reflects a change of switching from handshaking with a publicly hosted server to using a locally hosted server.

Test Table

Test	Instructions	Expected Result
1	Send a ping request to "http://nano:8000"	The requester to successfully complete the ping request.
2	Receive a ping request as "http://nano:8000"	The receiver to successfully receive and process the request.

Evidence

Test 1

```
[Handshake Requester] Starting Handshake over HTTP
[Handshake Requester] Start Handshake initiated
[Handshake Requester] Sending handshake request to http://nano:8000.
[Handshake Requester] Message: 2022-12-15 18:52:58.301563
[Vweb] Running app on http://localhost:8002/
[Handshake Requester] http://nano:8000 responded to handshake.
[Handshake Requester] Verified signature to match handshake key
[Handshake Requester] Handshake successful.
[Handshake Requester] Completed handshake
[Handshake Requester] Adding ref to refs
[Database] Checking if a reference to http://nano:8000 already exists
[Database] Checking if domain http://nano:8000 is in database
[Database] Found a reference to http://nano:8000
[Database] Reference to http://nano:8000 already exists, updating it...
[Database] Updating http reference for http://nano:8000
[Handshake Requester] Handshake Request Finished
```

Test 2

```
[Handshake Receiver] Received handshake request from node claiming to be: http://192.168.1.20:8002
[Handshake Receiver] Time parsed correctly as: 2022-12-15 18:52:58
[utils] Reading file: ./monochain/config.json

[config] Config Loaded...
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[Database] Checking if domain http://192.168.1.20:8002 is in database
[Database] Found a reference to http://192.168.1.20:8002
[Handshake Receiver] Node has come into contact with initiator before, no need to send a handshake request
[Handshake Receiver] Handshake Analysis Complete. Sending response...
```

Cycle 7

Test Table

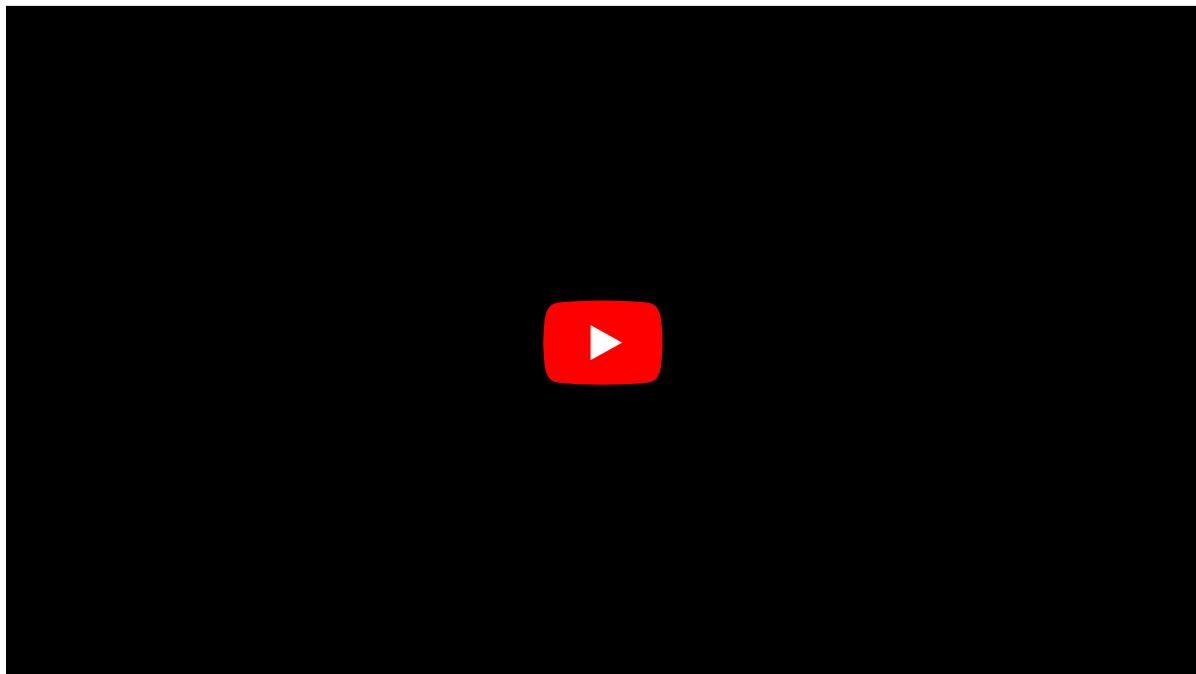
Test	Instructions	Expected Result
1	Navigate throughout the website on a desktop/laptop device	The website to stay "intact" with no components becoming hidden or breaking.
2	Navigate throughout the website on a phone/small device	The website to stay "intact" with no components becoming hidden or breaking.
3	Open the site on desktop device.	The desktop styled navigation bar should appear along the top of the screen.
4	Open the site on mobile device.	The mobile styled navigation bar should appear with a menu button to open and close the bulk of the contents.

Evidence

Test 1 & 3



Test 2 & 4



Cycle 8

Test Table

Test	Instructions	Expected Result	T T
1	Generate a key pair.	The program not to exit, which will happen if the key pair cannot be generated.	A
2	Generate a key pair and sign some data.	The program not to exit, which will happen if the key pair cannot be generated or sign the data.	A
3	Generate a key pair, sign and then verify some data.	The program not to exit and to verify that the message matches the signature.	A

Test Code

Test 1

```
pub fn test_gen_keys() {
    println("[testing] Testing Key Pair Generation")
    cryptography.gen_keys()
    // because the gen_keys function hasn't exited yet, we know that the keys are available
    println("[testing] Key Pair Generation Test Passed")
    assert true
}
```

Test 2

```
pub fn test_sign() {
    println("[testing] Testing Signing")
    keys := cryptography.gen_keys()

    println("[testing] Generated a new key pair successfully")
    message := time.utc().str().bytes()

    signed := keys.sign(message)
    println("[testing] Signed a message successfully: $signed")
    println("[testing] Signing Test Passed")
```

```

    // because the sign function hasn't exited yet, we know that the signature
    assert true
}

```

Test 3

```

pub fn test_verify() {
    mut failed := false
    println("[testing] Testing Verification")
    keys := cryptography.gen_keys()

    println("[testing] Generated a new key pair successfully")
    message := time.utcnow().str().bytes()

    signature := keys.sign(message)
    println("[testing] Signed a message successfully")

    if cryptography.verify(keys.pub_key, message, signature) {
        println("[testing] Verified a signature successfully")
        println("[testing] Verification Test Passed")
        failed = false
    } else {
        println("[testing] Verification failed")
        println("[testing] Verification Test Failed")
        failed = true
    }

    assert failed == false
}

```

Evidence

Test 1

```

● alfier@Alfies-MacBook-Pro node % v -stats test ./src/modules/cryptography
---- Testing...
----- V source code size: 30683 lines, 822597 bytes
----- generated target code size: 30073 lines, 1135483 bytes
----- compilation took: 649.861 ms, compilation speed: 47214 vlines/s
----- running tests in: /Users/alfier/WebstormProjects/A-Level-Project/packages/node/src/modules/cryptography/crypto_test.v
[testing] Testing Key Pair Generation
[cryptography] Signature verified
[testing] Key Pair Generation Test Passed
OK [1/3] 53.888 ms 11 asserts | main.test_gen_keys()

```

Test 2

```

[testing] Testing Signing
[cryptography] Signature verified
[testing] Generated a new key pair successfully
[testing] Signed a message successfully: [153, 178, 114, 175, 234, 235, 138, 250, 148, 172, 12, 175, 243, 201, 191, 69, 253, 123, 183, 186, 30, 219, 140, 100, 41, 236, 252, 118, 49, 113, 121, 135, 210, 101, 79, 73, 237, 213, 186, 168, 142, 37, 174, 233, 96, 51, 145, 63, 99, 210, 130, 79, 197, 188, 171, 93, 145, 9, 220, 237, 62, 26, 38, 41]
[testing] Signing Test Passed
OK [2/3] 78.469 ms 17 asserts | main.test_sign()

```

Test 3

```
[testing] Testing Verification
[cryptography] Signature verified
[testing] Generated a new key pair successfully
[testing] Signed a message successfully
[testing] Verified a signature successfully
[testing] Verification Test Passed
    OK [3/3] 83.395 ms 19 asserts | main.test_verify()
Summary for running V tests in "crypto_test.v": 47 passed, 47 total. Runtime: 215 ms.
```

```
Summary for all V _test.v files: 1 passed, 1 total. Runtime: 1136 ms, on 1 job.
```

Cycle 9

There were no tests for this cycle and hence nothing to check.

Cycle 10

Test Table

Test	Instructions	Expected Result
1	Completing a correct and valid handshake using a time object as the message.	A series of console logs on both nodes confirming that the handshake was successful.
2	Completing an invalid handshake using the string "invalid data" as the message.	For both nodes to record the handshake as unsuccessful and log such to the console.

Evidence

Test 1

```
[Handshake Requester] Starting Handshake over HTTP
[Handshake Requester] Start Handshake initiated
[Handshake Requester] Sending handshake request to http://nano:8000.
[Handshake Requester] Message: 2022-12-15 18:57:40.655632
[Handshake Requester] Assembled handshake request: server.HandshakeAssembleResult('{"initiator":{"key":[44,26,69,99,22,13,108,35,46,148,249,139,213,219,85,67,96,172,8,193,17
4,83,118,226,185,218,115,111,178,110,233,92],"ref":"http://192.168.1.20:8002"},"message":"2022-12-15 18:57:40.655632"}')
[Handshake Requester] http://nano:8000 responded to handshake.
[Handshake Requester] Verified signature to match handshake key.
[Handshake Requester] Handshake successful.
[Handshake Requester] Completed handshake.
[Handshake Requester] Adding ref to refs
[Database] Checking if a reference to http://nano:8000 already exists
[Database] Checking if domain http://nano:8000 is in database
[Database] Found a reference to http://nano:8000
[Database] Reference to http://nano:8000 already exists, updating it...
[Database] Updating http reference for http://nano:8000
[Handshake Requester] Handshake Request Finished
```

The sender's information

```
[Handshake Receiver] Received handshake request from node claiming to be: http://192.168.1.20:8002
[Handshake Receiver] Time parsed correctly as: 2022-12-15 18:57:40
[utils] Reading file: ./monochain/config.json

[config] Config Loaded...
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[Database] Checking if domain http://192.168.1.20:8002 is in database
[Database] Found a reference to http://192.168.1.20:8002
[Handshake Receiver] Node has come into contact with initiator before, no need to send a handshake request
[Handshake Receiver] Handshake Analysis Complete. Sending response...
```

The recipient information

Test 2

```
[Handshake Requester] Starting Handshake over HTTP
[Handshake Requester] Start Handshake initiated
[Handshake Requester] Sending handshake request to http://nano:8000.
[Handshake Requester] Message: invalid data
[Handshake Requester] Assembled handshake request: server.HandshakeAssembleResult('{"initiator":{"key":[44,26,69,99,22,13,108,35,46,148,249,139,213,219,85,67,96,172,8,193,17
4,83,118,226,185,218,115,111,178,110,233,92],"ref":"http://192.168.1.20:8002"},"message":"invalid data"}')
[Handshake Requester] Failed to shake hands with http://nano:8000, may have sent incorrect data, response body: 500 Internal Server Error
```

The sender's information

```
[Handshake Receiver] Received handshake request from node claiming to be: http://192.168.1.20:8002
[Handshake Receiver] Incorrect time format supplied to handshake by node claiming to be http://192.168.1.20:8002
```

The recipient information

Cycle 11

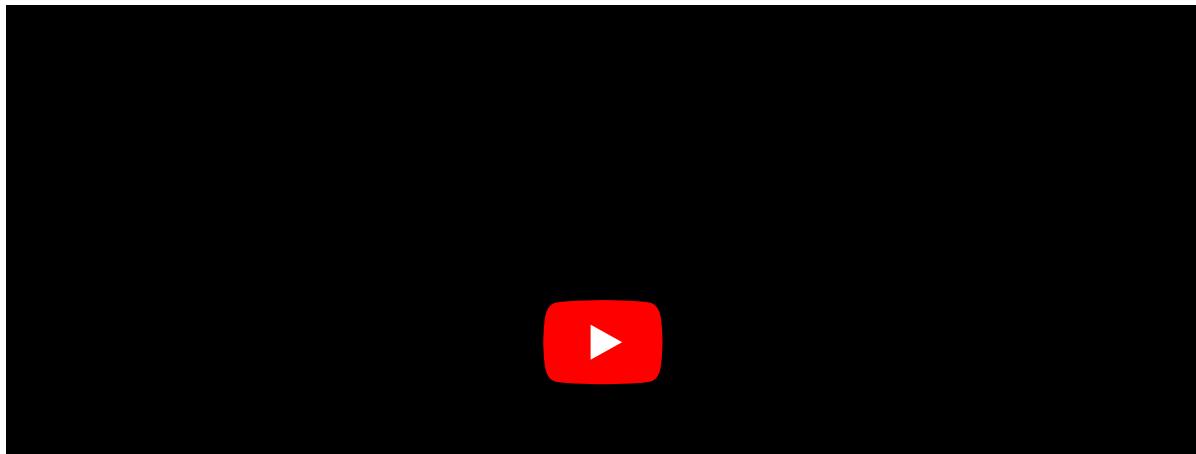
The way in which references were stored and used was completely refactored in cycle 11 and as such the code that was tested in this cycle no longer exists or is distributed in separate parts of the project, however the new references are still tested in the [tests for cycle 13 below](#).

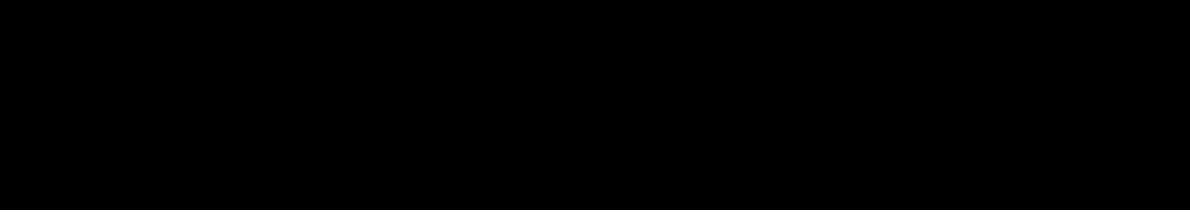
Cycle 12

Test Table

Test	Instructions	Expected Result
1	Navigate to 'https://localhost:8000/dashboard'	The dashboard page should redirect to the login page with an option to input a token or generate one.
2	Click the 'generate token' button	A token to be output in the node's terminal output.
3	Enter an invalid token into the input field and click 'submit'	The dashboard should not give the user access.
4	Enter an valid token into the input field and click 'submit'	The dashboard should give the user access to itself.
5	Type a message and click send.	The node should receive the message from the dashboard and attempt to send it across the network.
6	Connect to and then send a message to another node using the dashboard.	The node should connect to a pre-existing node and then send a message to it after a message is sent from the dashboard.
7	Send a message to the node from another node.	The node on the recipient end of test 6 should receive the same message as was sent from the other node.

Evidence





Cycle 13

Test Table

Test	Instructions	Expected Result
1	Use the Shell function to echo a message	The message to be echo'ed from the terminal
2	Launch the database.	The database to launch and no crashes to occur.
3	Stop the database.	The database to stop and no crashes to occur.
4	Launch and then connect to the database.	The database to launch then accept the program's connection without any crashes.
5	Get the last 5 messages from the database.	An array of messages to be returned in the expected type.
6	Create a node reference in the database.	The reference to be created in the database.
7	Create a node reference and then check if the software is aware of that node.	The "aware_of" check should return true for the reference test that already exists.
8	Collect all references from the database.	Should return an array of node references in the expected type.

Automated Test Code

```

import database
import pg

// This is a function used just to make the other tests easier
fn create_test_ref(db database.DatabaseConnection) {
    mut aware := db.aware_of('test')

    if !aware {
        // just in case the test ref doesn't exist
        db.create_ref('test', 'test'.bytes())
    }
}

```

```
        aware = db.aware_of('test')
    }
}
```

Test 1

```
fn test_shell() {
    cmd := 'echo "hello world"'
    assert database.sh(cmd)
}
```

Test 2

```
fn test_db_launch() {
    database.launch()
    // this test will crash if it fails
    assert true
}
```

Test 3

```
fn test_db_stop() {
    database.launch()
    database.stop()
    // this test will crash if it fails
    assert true
}
```

Test 4

```
fn test_db_connection() {
    database.launch()
    db := database.connect(false, pg.Config{})
    database.stop()
    // this test will crash if it fails
    assert true
}
```

Test 5

```
fn test_messages() {
    eprintln("WARNING - THIS TEST RELIES ON THE DATABASE HAVING ATLEAST 1 MESSAGE")
    database.launch()
    db := database.connect(false, pg.Config{})
```

```
messages := db.get_latest_messages(0, 5)
// if the incorrect format is returned here the program will exit.
database.stop()
assert messages.len > 0
}
```

Test 6

```
fn test_create_ref() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)
    database.stop()
    // this test will crash if it fails
    assert true
}
```

Test 7

```
fn test_aware_ref() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)
    aware := db.aware_of('test')
    database.stop()
    assert aware
}
```

Test 8

```
fn test_refs() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)      // create a ref if the test one doesn't already exist
    refs := db.get_refs()
    database.stop()
    assert refs.len > 0
}
```

Evidence

Test 1

```
[shell] > echo "hello world"
```

```
[shell] > echo "Hello world"
[shell] - hello world
    OK      [1/8]     4.637 ms     1 assert | main.test_shell()
```

Test 2

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
    OK      [2/8]   3541.004 ms     1 assert | main.test_db_launch()
```

Test 3

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
    OK      [3/8]   3342.387 ms     1 assert | main.test_db_stop()
```

Test 4

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
    OK      [4/8]   3511.885 ms     1 assert | main.test_db_connection()
```

Test 5

```
WARNING - THIS TEST RELIES ON THE DATABASE HAVING ATLEAST 1 MESSAGE, IF IT DOES NOT, IT WILL FAIL
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
```

```
NOTICE: relation "message_table" already exists, skipping
[Database] Tables created.

[Database] Getting latest messages
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [5/8] 3497.346 ms 1 assert | main.test_messages()
```

Test 6

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [6/8] 3486.599 ms 1 assert | main.test_create_ref()
```

Test 7

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [7/8] 3478.739 ms 1 assert | main.test_aware_ref()
```

Test 8

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
OK [8/8] 3483.221 ms 1 assert | main.test_refs()
```

Summary

```
Summary for running V tests in "database_test.v": 8 passed, 8 total. Runtime: 24346 ms.
```

Cycle 14

- ⓘ One of the tests in Cycle 14 has been skipped here as it was meant purely for during development as a proof of concept and the code that it was testing was then removed.
The code has also not changed between the end of cycle 14 and now so I have provided the same table and evidence as before.

Test Table

Test	Instructions	Expected Result
1	Send a message using web sockets	The message sent to be transmitted and received successfully.
2	Send multiple messages for a prolonged period of time (once every 0.5 seconds for 30 seconds).	The messages to continue to be transmitted and validated consistently during the entire time.

Evidence

Test 1

```
[Broadcaster] Assembling message with data: hi from linux :)
[utils] Reading file: ./monochain/config.json

[config] Config Loaded...
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[Database] Saving message to database.
[Database] Message saved.
[Broadcaster] Message assembled, broadcasting to refs...
[Broadcaster] Sending message to all known nodes.
[Broadcaster] Sending message to public nodes.
[Broadcaster] Attempting to send message to https://nano.monochain.network
[Broadcaster] Attempting to send message to http://192.168.1.20:8000
[Broadcaster] Sending message to websocket nodes.
[Broadcaster] Attempting to send message to http://192.168.1.20:8002
[Broadcaster] Created threads to send message to all known nodes.
[Broadcaster] Waiting for threads to return.
/tmp/v_1000/node_4283625912010421186.tmp.c:13185: at server__Websocket_Server_send_to_all_thread_wrapper: RUNTIME ERROR: invalid memory access
00755a28 : by ???
[Broadcaster] Failed to send a message to http://192.168.1.20:8000. Node is probably offline. Error: dial_tcp failed for address 192.168.1.20:8000
tried addrs:
    192.168.1.20:8000: net: socket error: 111; code: 111

0074a770 : by ???
7f9f272088 : by ???
7f9f3670cc : by ???
```

Websocket message attempted to be sent from a Linux machine

```
[Broadcaster] Assembling message with data: hi from macbook to macbook
[utils] Reading file: ./monochain/config.json
```

```
[config] Config Loaded...
[utils] Reading file: ./monochain/keys.config
[keys] Keys loaded from file.
[Database] Saving message to database.
[Database] Message saved.
[Broadcaster] Message assembled, broadcasting to refs...
[Broadcaster] Sending message to all known nodes.
[Broadcaster] Sending message to public nodes.
[Broadcaster] Attempting to send message to http://192.168.1.2:8000
[Broadcaster] Attempting to send message to http://192.168.1.4:8000
[Broadcaster] Attempting to send message to http://nano:8000
[Broadcaster] Attempting to send message to test
[Broadcaster] Sending message to websocket nodes.
[Websockets] Sending message to all clients...
[Websockets] Sending a message to all 1 clients
[Websockets] Sending message to socket with 8d753e7f933f9b84e38f812e6d6ae61a
[Websockets] Message sent to 8d753e7f933f9b84e38f812e6d6ae61a
[Websockets] Message sent to all clients
```

Websocket message successfully sent between two macOS devices.

Test 2

```
[Broadcaster] Received message:  
[Broadcaster] Sender: [75, 226, 169, 219, 199, 15, 152, 247, 10, 70, 102, 135, 149, 89, 193, 167, 70, 81, 84, 215, 5, 39, 178, 0, 169, 3, 165, 45, 124, 178, 107, 202]  
[Broadcaster] Sent at: 2022-09-22 12:03:11  
[Broadcaster] Message: jumagi1663848191170  
  
[Broadcaster] Sending message to all known nodes.  
[Broadcaster] Sending message to public nodes.  
[Broadcaster] Have not seen message before.  
[Broadcaster] Saving message to database.  
V panic: array.get: index out of range (i == 0, a.len == -1)  
v hash: bbf1ba4  
0 src 0x000000001006ccccf8 array.get + 224  
1 src 0x000000001007d6174 database__DatabaseConnection_get_refs + 968  
2 src 0x000000001007d68d8 server__forward_to_all + 172  
3 src 0x000000001007d5768 server__broadcast_receiver + 1120  
4 src 0x000000001007bea00 server__App.broadcast_route + 364  
5 src 0x000000001007ba39c web__handle_conn_T_server__App + 6580  
6 src 0x000000001006c5f80 web__handle_conn_T_server__App_thread_wrapper + 96  
7 src 0x000000001007fd25c GC_start_routine + 104  
8 libsystem_pthread.dylib 0x000000001906b326c _pthread_start + 148  
9 libsystem_pthread.dylib 0x000000001906b308c thread_start + 8  
  
error Command failed with exit code 1.  
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.  
alfier@Alfies-MacBook-Pro node %
```

Errored after approximately 10 seconds of 1 message every 0.5 seconds

4 Evaluation

4.1 Evaluation of Success Criteria

Node Software & Protocol

Individual blocks

Criterion	Description	When was it achieved
1	Blocks must refer to the previous block in the chain.	Not met.
2	Blocks must be detected as invalid and disregarded by the node software.	Not met.
3	Blocks must be detected as valid and approved by node software.	Not met.

Purely in terms of the actual software aspect none of these three criterion were met since I did not reach the stage in development of the project which would allow for the addition and translation of protocol rules into the actual software networking to allow the blockchain to run and send theorised data through it, however in the protocol form these criterions were all laid out as discussed in each individual one.

Criterion 1

-  Blocks must refer to the previous block in the chain.

This criterion was not implemented in software due to time constraints as discussed in summary paragraph above. However in terms of the protocol, the framework for how this would work and hence would be implemented into the node software can be found in the [initial project analysis here](#).

Criterion 2 and 3

-  2. Blocks must be detected as invalid and disregarded by the node software.
3. Blocks must be detected as valid and approved by node software.

Both of these cycles relate to the same thing: validating blocks and acting accordingly; with criterion 2 being in relation to disregarding invalid blocks and cycle 3 approving valid blocks.

Sadly neither of these criterion were implemented in the node software due to time constraints as discussed in summary paragraph above.

Although this was not implemented on the scale of blocks, this criterion was partially met in the fashion of the node software detecting and disregarding transactions and messages that were invalid. This was first implemented in [cycle 10](#) where it was made that the software should only sign specific types of data that it received and was then expanded in more detail throughout cycles [12](#) and [14](#).

```
// V code - within the "pong" route in /src/modules/server/main.v

// with this version of the node software all messages should be time objects
time := time.parse(req_parsed.message) or {
    eprintln("Incorrect time format supplied to /pong/:req by node claiming to be $req_parsed")
    return app.server_error(403)
}

// time was okay, so store a slight positive grudge
println("Time parsed correctly as: $time")
```

Therefore it can be seen that there is evidence that this criterion would've been met if the project's development continued over a longer time span.

Network Communication

Criterion	Description	When was it achieved
4	Nodes should have an open api server running using http that contains enough pathways to complete all the needed types of communication.	First developed in Cycle 1 , met in Cycle 12 .
5	Node should have web-socket connection abilities for server or client connections.	Partially met in Cycle 14 .
6	Nodes should be able to add and remove each other to their communication lists which should be stored locally.	Met in Cycle 11 , improved in Cycle 13 .
7	Nodes should be able to store a local "grudge" object that just remembers who the node has a good or bad experience with to change their trust level when they create their next block.	Not met.
8	Nodes should send through a digital signature with each communication.	Met in Cycle 12 .
9	Nodes should be able to remember each other through a combination of their wallet address and their ips/domains.	Met in Cycle 11 , improved in Cycle 13 .

Criterion 4

- ✓ Nodes should have an open api server running using http that contains enough pathways to complete all the needed types of communication.

← → ⌂ ⚡ localhost:8002

This is the api route for a node running on the Monochain network. To get to the dashboard, go to /dashboard

The above image shows the home route of the http api server running on the node software, since there are quite a few api routes the rest are shown below:

User Pages	/ /dashboard /login
Internal Api	/dashboard/gentoken /dashboard/login /dashboard/send_message
External Api	/handshake /broadcast

These routes not only allow for the http server to be used for communication between nodes but also allow for the user to interact and send messages over the network. This can be seen through the categorisation supplied with the 'User Pages' being used to show the user some data or allow them to interact with their node through a nicer ui than having to use the terminal window; the 'Internal Api' is then used to ensure the person accessing the node does actually have control of the node and let them send messages using the web interface; and finally the 'External Api' are the routes that will be called by other nodes looking to interact with or send a message to the recipient node.

Criterion 5

- ! Node should have web-socket connection abilities for server or client connections.

This criterion was partially met in [cycle 14](#) where websockets were implemented using VLang's net.websocket module and the functionality tested well initially but had some serious issues upon continued use.

```
[Broadcaster] Received message:  
[Broadcaster] Sender: [75, 226, 169, 219, 199, 15, 152, 247, 10, 70, 102, 135, 149, 89, 193, 167, 70, 81, 84, 215, 5, 39, 178, 0, 169, 3, 165, 45, 124, 178, 107, 202]  
[Broadcaster] Sent at: 2022-09-22 12:03:11  
[Broadcaster] Message: jumagi1663848191170  
  
[Broadcaster] Sending message to all known nodes.  
[Broadcaster] Sending message to public nodes.  
[Broadcaster] Have not seen message before.  
[Broadcaster] Saving message to database.  
V panic: array.get: index out of range (i == 0, a.len == -1)  
v hash: bbfb1ba4  
0 src 0x00000001006ccccf8 array_get + 224  
1 src 0x00000001007d6174 database_DatabaseConnection_get_refs + 968  
2 src 0x00000001007d68d8 server_forward_to_all + 172  
3 src 0x00000001007d6768 server_broadcast_receiver + 1120  
4 src 0x00000001007beec server_App_broadcast_route + 364  
5 src 0x00000001007ba39c vweb_handle_conn_T_server_App + 6580  
6 src 0x00000001006c5fbd0 vweb_handle_conn_T_server_App_thread_wrapper + 96  
7 src 0x00000001007fd25c GC_start_routine + 104  
8 libsystem_pthread.dylib 0x00000001906b826c _pthread_start + 148  
9 libsystem_pthread.dylib 0x00000001906b308c thread_start + 8  
error Command failed with exit code 1.  
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.  
alfier@Alfies-MacBook-Pro ~ %
```

Messages being received and then crashing from Cycle 14

The above screenshot shows a message being successfully received and then the software crashing after attempting to forward it onto the other nodes it was aware of. This occurred approximately 20 messages into a test where a message was sent every 0.5 seconds as such showing that the websockets partially work but cannot handle continued use very well.

Sadly this is primarily an issue with VLang and although improved code on my half would most likely expand the length of time that the software runs for before crashing there is a high probability that these sorts of issues would continue to happen. This is because the panic shown above and similar errors that were encountered occurred upon the websocket module calling other vlang standard library modules through my middleware meaning that the issues were within the modules supplied to the vlang standard library which I cannot easily fix.

Criterion 6

- Nodes should be able to add and remove each other to their communication lists which should be stored locally.

In order to prove that this functionality works quickly and easily after changes I built the following automatic testing functions:

```

// This tests the creation of a node reference
fn test_create_ref() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)
    database.stop()
    // this test will crash if it fails
    assert true
}

// This tests to check if the node will correctly state if it is aware of a node
fn test_aware_ref() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)
    aware := db.aware_of('test')
    database.stop()
    assert aware
}

// This tests to see if the node will collect all of it's references properly
fn test_refs() {
    database.launch()
    db := database.connect(false, pg.Config{})
    create_test_ref(db)      // create a ref if the test one doesn't already exist
    refs := db.get_refs()
    database.stop()
    assert refs.len > 0
}

// This is used to generate a testing reference object when needed
fn create_test_ref(db DatabaseConnection) {
    mut aware := db.aware_of('test')

    if !aware {
        // just in case the test ref doesn't exist
        db.create_ref('test', 'test'.bytes())
        aware = db.aware_of('test')
    }
}

```

These functions are part of the larger 'database_test.v' file which I designed in [cycle 13](#) but are the only tests necessary to prove this criterion has been met. Upon running these tests the following is output to the terminal by VLang's built in testing functionality.

```
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
    OK [6/8] 3474.500 ms    1 assert | main.test_create_ref()
[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
    OK [7/8] 3492.888 ms    1 assert | main.test_aware_ref()
```

The first two tests passing

```

[Database] Launching database...
[shell] > docker start monochain-postgresql
[shell] - monochain-postgresql
[Database] Database running.
[Database] Connected to database. (localhost:5432)
[Database] Creating tables...
[Database] Creating http reference table...
NOTICE: relation "Http_Reference" already exists, skipping
[Database] Creating message table...
NOTICE: relation "Message_Table" already exists, skipping
[Database] Tables created.

[Database] Checking if domain test is in database
[Database] Found a reference to test
[Database] Stopping database...
[shell] > docker stop monochain-postgresql
[shell] - monochain-postgresql
[Database] Stopped database.
    OK      [8/8]  3489.015 ms    1 assert  | main.test_refs()
Summary for running V tests in "database_test.v": 8 passed, 8 total. Runtime: 24111 ms.

```

The final test passing.

As shown above, not only did the three tests discussed pass, but so did the other 5 in this testing module, hence proving that this criterion is met.

Criterion 7

- ⚠ Nodes should be able to store a local "grudge" object that just remembers who the node has a good or bad experience with to change their trust level when they create their next block.

Although I did begin work on this criterion while [introducing the sql database in cycle 13](#), I quickly realised that there would be no way of actually using grudges until block creation was introduced later on in the project so removed it to be reintroduced later. Sadly however this later point was never reached so this criterion has not been met, and it should be introduced shortly after block creation has been implemented and criterion 1,2 and 3 have been met.

Criterion 8

- ✓ Nodes should send through a digital signature with each communication.

As of [cycle 12](#), all messages and methods of communication sent between nodes contains a signature except for some specific exceptions in which it is not required such as when one node request the initiation of a handshake from another node, since the signatures are then sent within the handshake. Therefore this criterion is currently met, although it would need to be continued to be met throughout future development and the introduciton of more communication methods and usecases between nodes.

Criterion 9

- Nodes should be able to remember each other through a combination of their wallet address and their ips/domains.

As of [cycle 10](#) all messages between nodes contain the sending node's ip address or dns domain (acts the same as an ip address in all forms of communication used within the project) and whenever a node encounters a message from a node that it does not recognise the ip address/domain of it will send it a handshake request in order to check its validity and then assuming it is a valid monochain node will store the data it learns about that node locally.

Initially this data was stored in JSON files but since I realised this could get slow and complicated quickly in cycle 13 this was improved to be done through a PostgreSQL sql database which the node software queries and talks to as often is required. By default this database will be automatically setup and controlled by the node software through the use of a docker container but it can be hosted manually by the user if they wished to have more control over the database, for example to give multiple nodes access to the same database.

WHERE		ORDER BY	
	id	domain	key
1	4	http://192.168.1.2:8000	[44, 26, 69, 99, 22, 13, 108, 35, 4...
2	5	http://192.168.1.4:8000	[75, 226, 169, 219, 199, 15, 152, 2...
3	6	http://nano:8000	[223, 72, 85, 158, 120, 139, 141, 2...
4	7	test	[116, 101, 115, 116]
			2022-09-22 11:42:06.615361
			2022-09-22 11:48:04.064173
			2022-09-22 17:46:55.876386
			2022-09-26 11:19:59.083813

An example database of the reference table of a node.

General features

Criterion	Description	When was it achieved
10	A configuration handler.	Met in Cycle 3 .
11	The configuration handler should be usable and accessible to technical users with non-technical users being able to use it with additional help.	Met in Usability testing .
12	Nodes should have some kind of web based access dashboard.	Met in Cycle 12 .
13	The node dashboard should be easily accessible and usable by all forms of users.	Partially met in Usability testing

Criterion 10

- A configuration handler.

The configuration handler for the node software was originally introduced in cycle 3 but it has been continued to be improved, had more settings added and restructured for easier use throughout many various points of the project and should continue into the future as development proceeds.

This handler controls the generation of new configurations, alongside the loading, saving and use of those configuration settings as is needed by other parts of the project.

Criterion 11

- The configuration handler should be usable and accessible to technical users with non-technical users being able to use it with additional help.

In [usability testing](#), three of the questions in the user survey were examined to show that all users had atleast heard of the configuration file existing, with the majority knowing what it could be used for and some of the more technical users within that pool knowing how to use it without additional help but the majority of users requiring assistance.

This meets the criterion since the majority of users are non-technical users and those users who did not require assistance were the technical users, however in the future this could still be improved by including some basic documentation to assist users and hopefully allow more users to be able to use the configuration handler.

Criterion 12

- Nodes should have some kind of web based access dashboard.

This criterion was first met in cycle 12 and uses a simple token based system to ensure that users who gain access to the dashboard are in fact in control of the node which hence meets the criterion.

Although this could be improved by adding more functionality to the dashboard to give users more of a reason to use it alongside making the way in which users both login to and the dashboard itself look nicer and contain more context to explain how to login and use it so as to make the dashboard simpler to use as some users did not understand how to login without assistance.

Criterion 13

- The node dashboard should be easily accessible and usable by all forms of users.

This criterion was proven to be partially met in [usability testing](#) as the majority of users seemed to know how to access the dashboard but due to the mis-communication during testing some users did not realise they were navigating to and using the dashboard which would probably increase the number of users who stated they were able to navigate and use it.

I suggest this as I discovered more users said they knew how to access the dashboard than the amount who said they knew what it was and after the survey I talked to all the users who did not agree to either question relating to this topic in the survey they all seemed to realise what it was after a very brief explanation, hence suggesting some form of basic documentation or tutorial explaining the essentials to them could bring this up to being met.

Although alongside this I believe that the login page to access the dashboard should be a lot more clear about where to find the token that was generated to login with (it is output within the node's terminal output) as most users who tested this had to search around for a few minutes to find it if they did so without assistance and in reality this should be a very simple task. Hence in the future it would be a good idea to add a basic tutorial/summary on how to login so as to speed this up.

Non-functional

Criterion	Description	When was it achieved
14	The node software should not crash.	Partially met, as of Function and Robustness testing .
15	The software should be capable of receiving multiple messages per second.	Not met, as of Function and Robustness testing .
16	The software should attempt to retry any failed processes (such as loading a file) and should only exit after multiple failed attempts.	Met, as of Function and Robustness testing .

Criterion 14

- ! The node software should not crash.

This criterion was tested in [function and robustness testing](#) and it was found that the node software will not crash under casual use on Windows and MacOS systems or in specific scenarios on Linux systems, but it will crash if put under constant/heavy load or communication involving websockets (which are on by default with a Linux system).

Since the websocket functionality can be disabled using the configuration handler, all three system types can be setup such that the software does not crash under casual use but they will all crash under consistant heavy use. Example of such consistant heavy use is the sending of a message more than once per second for atleast 10 messages in a row which has a high probability of causing a crash, with that probability seemingly increasing as more messages are sent.

Some parts of this are due to my own code, such as the number of messages sent in a row before the software crashes, which can very likley be increased as to increase stability through the refactoring of networking code. However, other parts are due to the limitations of the language that the software is written in itself, such as the Linux systems behaving differently to the Windows and MacOS systems whilst running the same code.

Criterion 15



The software should be capable of receiving multiple messages per second.

As summarised in the criterion above, the software cannot handle multiple messages per second for longer than about 10 seconds. While sending one message every 0.5 seconds I found the software would regularly crash after about 10 seconds which proves this criterion has not been met. This should then be worked on in the future.

Criterion 16



The software should attempt to retry any failed processes (such as loading a file) and should only exit after multiple failed attempts.

As shown in detail in [testing for function and robustness](#) the node software will attempt to retry any failed processes that do not exit the program completely (such as the websocket failures which crash the entire program) and will only exit after atleast 3 reattempts.

Any scenario in which a process is not completed after failing is due to the entire program crashing which is out of my control as programs compiled by Vlang should theroretically not do that, but as they do these failures are out of my control.

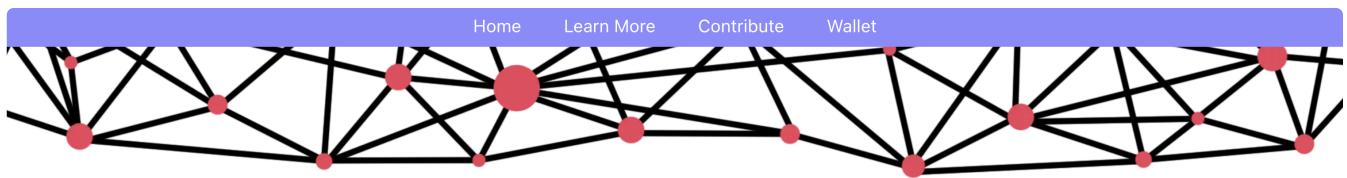
Webportal

Page types

Criterion	Description	When was it achieved
17	A "home" page that summarises what the project is about.	Created in Cycle 1 , improved in Cycle 7
18	An "info" page that explains the project in more detail.	Created in Cycle 1 , improved in Cycle 7
19	A "downloads" page that lets the user download the node software.	Created in Cycle 1 , improved in Cycle 7
20	A "wallet" page that allows the user to see any data or items stored within their digital wallet.	Created in Cycle 1 , has not been met it's functionality.

Criterion 17

- ✓ A "home" page that summarises what the project is about.

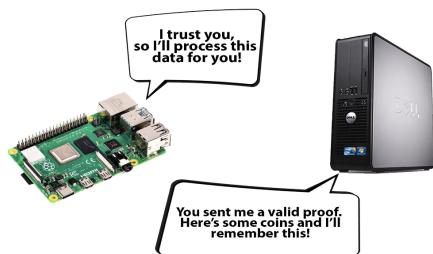


Welcome to the MonoChain

The MonoChain is a new generation of decentralised computing built alongside the "Proof of Trust" protocol.

What is Proof of Trust?

'Proof of Trust' is a custom built consensus protocol that uses a 'trust' parameter in order to be faster and eco-friendlier than 'Proof of Work' whilst having much lower entry requirements than 'Proof of Stake'.



The Home page

The webportal has a home page that was introduced in [cycle 1](#) to give visitors an understanding of the basic summary of the project and was then improved significantly in [cycle 7](#) to be easier and simpler to use whilst also making it look nicer to visitors.

Criterion 18

- ✓ An "info" page that explains the project in more detail.

Home Learn More Contribute Wallet

The MonoChain Project

The MonoChain is a decentralized network built upon the proof of trust consensus protocol, designed and built by Alfie Ranstead, a computer scientist with a strong interest in complex problems and their solutions. Blockchain technology is just a natural part of this group of technology, but it is far from perfect, and that is why the proof of trust protocol - and this project - exist.

What is the Proof of Trust Protocol?

The proof of trust protocol is a custom built consensus protocol built in order to define how nodes [computers that contribute to the network] decide how to trust or not trust other nodes. It uses a system of parameters used to store both grudges and appreciations for other nodes, alongside a few other values, which help decide whether or not to re-process whatever information other nodes have sent to it. A key addition here is that nodes will also choose to randomly reprocess data that they have received, in order to make sure that the seemingly 'trustworthy' nodes are still actually trustworthy and haven't been tampered with.

Why is the Proof of Trust Protocol important?

The two most used consensus protocols are currently 'proof of work' and 'proof of stake'; both of these protocols are fundamentally flawed, and as such are not suitable for use in a widely adopted, decentralised system. The 'proof of work' protocol requires nearly all the nodes on the network to calculate the same data, only to reward a singular node for it per block, which is not only heavily favouring nodes comprised of more expensive hardware, but is also terrible for the environment by wasting massive amounts of energy. The 'proof of stake' protocol requires the nodes to put forward a large amount of money to be able to stake, which is more energy efficient, but extremely favouring of nodes that have large backings which defeats part of the reasoning for a decentralised system, as it just gives power to those who have money - which is against a lot of the ideas of the decentralised system. This is where proof of trust comes in, as it is a consensus protocol that is more energy efficient than 'proof of work', and although it is not as energy efficient as 'proof of stake', it does allow anyone to use and operate a node, removing the need for a large amount of money to be staked.

So Why the MonoChain?

The info page

The info page was originally introduced in [cycle 1](#) and was simply a large block of text explaining some of the technical side of the project but after [cycle 7](#) it was refactored to explain more concepts relevant to the project under general questions which act as headers and give the project a lot more detail.

Criterion 19

- ✓ A "downloads" page that lets the user download the node software.

Mine for the Monochain Network

Download a Node

This is an example node (mining) software written in Vlang, it is recommended that you clone the git repository and compile it yourself so that you can edit any pieces of code you wish to and to ensure that it runs properly on your system as V compiles to an executable that will be different for different systems.

The below zip file contains an arm64 macOS executable called 'node-native' (native because that's the native OS for the laptop I do the development for this project on) and an x86 windows executable called 'node-windows'. If you don't use either of these systems then you must clone the git repository and use the v compiler to compile the project yourself.

[Download Node Software](#)[Clone the Repo](#)

Create the Config File

Editing the Config File

Setup a Node

Once you've downloaded the node software or compiled it directly, you will need to setup a configuration file. The simplest way of doing this is by first running the program once with no config file.

The program will then ask you if you wish to generate a new config file which can be accepted by typing 'y' and then pressing enter. This will generate the file under the path './monochain/config.json' in the same directory as the executable.

Assuming you also do not have a keypair file, the program will then ask if you would like to generate a new keypair file which can be accepted by typing 'y' and then pressing enter. This will generate under the path './monochain/keys.config' and can then be replaced with your own keypair if you already have one.

***** MonoChain Mining Software *****

The download page

The downloads page for the webportal was originally introduced in [cycle 1](#) and was simply a button which allowed a user to download the first versions of the node software but throughout continued development it has gained a nicer look, a basic tutorial on how to setup a node and automated configuration which ensures the node software being downloaded is the newest compiled version.

Criterion 20

- ! A "wallet" page that allows the user to see any data or items stored within their digital wallet.

Wallet

As soon as our full and thorough testing phase is completed you will be able to create a wallet on this page allowing you to begin using the MonoChain from a web browser.

The wallet page

This page was created in cycle 1 and simply contained a message to the user that this page would be introduced in the future, the message has changed slightly since then but as this page requires functionality from the node software that does not currently exist it has not progressed past this form.

In order to get this page to a point at which it would meet this criterion transactions, blocks and blockchain state would need to be in working order such that the contents of a specific wallet could actually be viewed and shown on this page.

Non-functional

Criterion	Description	When was it achieved
21	Non-technical users must be able to identify what the point of the project is just from the homepage of the website.	Met as of usability testing .
22	Users should be able to get to the majority of what they would want to get to within 3 clicks.	Met in Cycle 7 .
23	The web-portal should be available and working on a variety of device sizes including mobile and desktop.	Partially met in Cycle 7 .
24	The web portal should run without crashing under any circumstance.	Met as of testing for function and robustness .

Criterion 21

- ✓ Non-technical users must be able to identify what the point of the project is just from the homepage of the website.

As examined in [usability testing](#) the vast majority of users understood the idea of the project after being given free range on the webportal and since not all of the users were likely to have read all of the information on the info page it can be assumed that the home page was the primary force supporting this. Therefore it can be stated that users could identify the point of the project from the homepage of the website.

Criterion 22

- ✓ Users should be able to get to the majority of what they would want to get to within 3 clicks.

As examined in [usability testing](#), all pages can be visited within an absolute maximum of 2 clicks of the homepage which including the 1 click it takes to get back to the home page from any other page it can be stated that all pages can be navigated to within 3 clicks from any other page.

Criterion 23

- ! The web-portal should be available and working on a variety of device sizes including mobile and desktop.

This was partially met in cycle 7 as the webportal originally only worked on desktop but in this cycle it was updated to work with almost all mobile devices, however the tablet screen sizes were overlooked and as such some specific elements (the home page buttons, tutorial on how to setup the node software) go off of the screen which I would classify as not working. Although as technically the site is available on all device types and it only 'breaks' on devices that aren't as commonly used this criterion is partially met but this could be fixed in the future.

Criterion 24

- The web portal should run without crashing under any circumstance.

This criterion was met in [testing for function and robustness](#) and as the webportal has still not crashed since its initial public debut it has met this criterion.

It is worth noting here that client sided exceptions such as a client going offline while using the website do not count as "crashing" the webportal since they can be fixed by refreshing the page and do not effect anyone except that client/computer.

4.2 Evaluation of Usability Features

Effective

Feature No.	Description	Success/Fail
1	Ensure that the user can use the majority of the project without having to get much support and thus the software is intuitive.	Partial Success
2	Make the software as easy to understand as possible such that the user has high autonomy.	Partial Success

The questions from [usability testing](#) relevant to this section:

Question	Summary of Responses
Do you understand the idea of the project?	80% Agreed, 20% Disagreed. The vast majority of users had atleast a basic understanding of the idea of the project.
Do you understand the different parts of the project?	40% Agreed, 40% were Neutral, 20% Disagreed. A large minorit of users understood the different projects with the majority being somewhat unsure.
Do you know how to setup the node software?	40% Agreed, 40% were Neutral, 20% Disagreed. A large minorit of users knew how to setup the node software with the majority o users indicating that they would probably need atleast some leve of assistance.
Did you know how to use the website	80% Strongly Agree, 20% Agree. All users knew how to use the website at atleast some basic level with the vast majority being confident in how to use it.

Feature 1

- ! Ensure that the user can use the majority of the project without having to get much support and thus the software is intuitive.

Based upon the usability testing responses supplied in the above table we can immediately see that the majority of users understood the general idea of the project and how to use the web portal/website but only 40% seemed to understand the node software. This then means that that 40% were likely the same 40% who stated that they understood all the different parts of the project.

This is sufficient evidence to state that this feature was partially met as it was not required that all users can use and understand all of the project but rather the majority of the project and since the vast majority of users understood the fundamental idea of the project and how to use the website/webportal it is clear that the vast majority of users understood the essential parts of the project. However it can also be shown that the vast majority also did not grasp the concepts and understanding of the node software which is a large portion of the project which hence prevents this feature from being a complete success.

Feature 2

- ! Make the software as easy to understand as possible such that the user has high autonomy.

Similar to feature 1, we can see from the responses given in the above table from usability testing that the vast majority of users knew how to use the website but were less stable on the node software meaning that they would have high autonomy with anything related to that website but not the node software. Therefore this feature is partially met.

Efficient

Feature No.	Description	Success/Fail
1	Ensure users can get from one point on the website to any other within 3-7 clicks, with 3 being for major sections such as the home page and 7 being the more detailed sections such as the documentation for the configuration file for the Node software.	Success
2	Allow users to setup a node and have it start running within as small a time period as possible.	Partial Success

Feature 1

- ✓ Ensure users can get from one point on the website to any other within 3-7 clicks, with 3 being for major sections such as the home page and 7 being the more detailed sections such as the documentation for the configuration file for the Node software.

All pages on the webportal can be reached within 2 clicks of the homepage and since it takes a maximum of 1 click to get to the homepage from any other page, any page can be reached from any other page within 3 clicks.

All major sections can be reached in 1 click of the homepage on desktop meaning that all major sections can be reached within a maximum of 2 clicks on any platform and any page can be reached within 3 clicks on any platform meeting the requirements for this feature.

Feature 2

- ! Allow users to setup a node and have it start running within as small a time period as possible.

If the user is on macOS or a precompiled architecture and wishes to setup a worker node, setup and running can be done very quickly. This is because the supplied example configuration file needs very little setup and can be done through the terminal control of the node software itself which is built to be fast and simple to use.

The software can also be compiled for other operating systems such as Windows and Linux which would allow this to be the case for an even larger pool of users but that would require doing so on a windows/linux computer which I did not have time to do every time I updated the software. Although if the project was being ran by a large enough user pool to warrant doing so it could be done.

The issue here is that if the user wishes to setup a leader node they must carefully configure their configuration file manually and ensure they have set up the correct open ports and networking settings within their local network which can take some time. Or if the user is trying to run the software on a non-precompiled architecture they must download the source code of the project and the Vlang compiler in order to compile and run the software. In either case the node cannot be setup very quickly hence preventing this feature from being completely successful.

Engaging

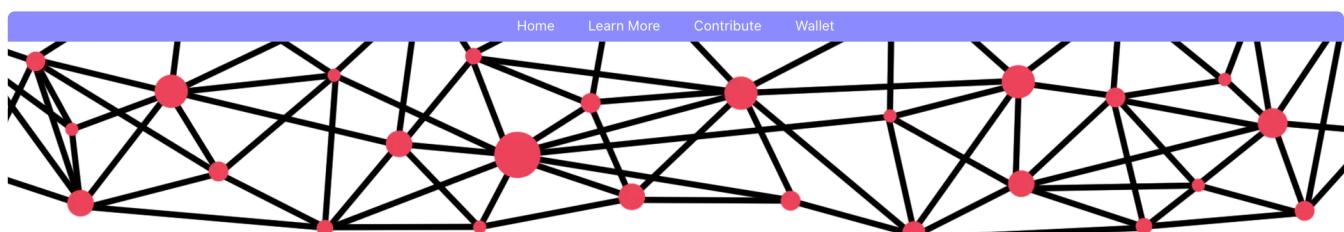
Feature No.	Description	Success/Fail
1	Create a simple, clear Homepage for the web-portal.	Success
2	Stick to a simple, repetitive art style so the user recognises all parts of the web-portal as being a part of the project.	Success
3	Create more detailed sections/pages of information for users who wish to understand the technical detail better.	Success

The questions from [usability testing](#) relevant to this section:

Feature 1

- ✓ Create a simple, clear Homepage for the web-portal.

Homepage is a simple, somewhat minimalist page that gives a basic insight into the project without going into too much detail and gives the users some options for where they could navigate to next.

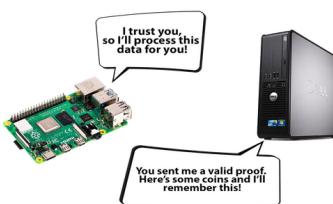


Welcome to the MonoChain

The MonoChain is a new generation of decentralised computing built alongside the "Proof of Trust" protocol.

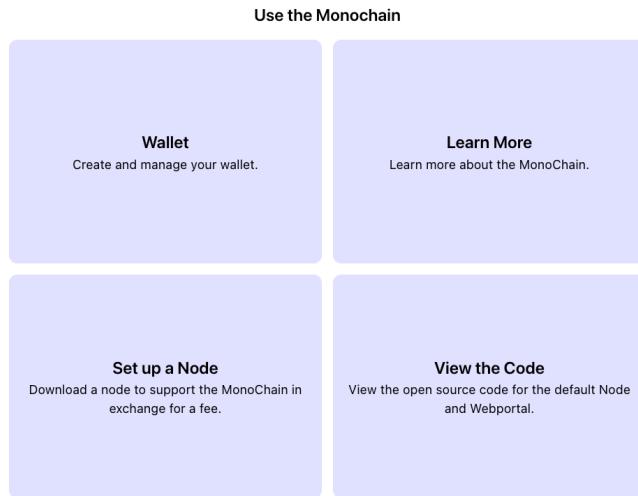
What is Proof of Trust?

'Proof of Trust' is a custom built consensus protocol that uses a 'trust' parameter in order to be faster and eco-friendlier than 'Proof of Work' whilst having much lower entry requirements than 'Proof of Stake'.



I trust you, so I'll process this data for you!

You sent me a valid proof. Here's my response and I'll remember this!



Feature 2

- Stick to a simple, repetitive art style so the user recognises all parts of the web-portal as being a part of the project.

The website sticks to a consistent colour scheming and art style throughout the web-portal, using a solid white background, a solid black or white typeface with a consistent font and the use of purples ranging from mid to light in order to emphasise certain parts of the site

The navigation bar is also consistent throughout the site dependent upon platform type

The MonoChain is a decentralized network built upon the proof of trust consensus protocol, designed and built by Alfie Ranstead, a computer scientist with a strong interest in complex problems and their solutions. Blockchain technology is just a natural part of this group of technology, but it is far from perfect, and that is why the proof of trust protocol - and this project - exist.

What is the Proof of Trust Protocol?

The proof of trust protocol is a custom built consensus protocol built in order to define how nodes [computers that contribute to the network] decide how to trust or not trust other nodes. It uses a system of parameters used to store both grudges and appreciations for other nodes, alongside a few other values, which help decide whether or not to re-process whatever information other nodes have sent to it. A key addition here is that nodes will also choose to randomly reprocess data that they have received, in order to make sure that the seemingly 'trustworthy' nodes are still actually trustworthy and haven't been tampered with.

Showcase of the purple navigation bar with white background and black text

Mine for the Monochain Network

Download a Node

This is an example node (mining) software written in Vlang, it is recommended that you clone the git repository and compile it yourself so that you can edit any pieces of code you wish to and to ensure that it runs properly on your system as V compiles to an executable that will be different for different systems.
The below zip file contains an arm64 macOS executable called 'node-native' (native because that's the native OS for the laptop I do the development for this project) and an x86 windows executable called 'node-windows'. If you don't use either of these systems then you must clone the git repository and use the v compiler to compile the project yourself.

[Download Node Software](#)[Clone the Repo](#)

Create the Config File

Editing the Config File

Setup a Node

Once you've downloaded the node software or compiled it directly, you will need to setup a configuration file.

The simplest way of doing this is by first running the program once with no config file.

The program will then ask you if you wish to generate a new config file which can be accepted by typing 'y' and then pressing enter. This will generate the file under the path '/monochain/config.json' in the same directory as the executable.

Assuming you also do not have a keypair file, the program will then ask if you would like to generate a new keypair file which can be accepted by typing 'y' and then pressing enter. This will generate under the path '/monochain/keys.config' and can then be replaced with your own keypair if you already have one.

```
**** MonChain Mining Software *****
Reading file: ./monochain/config.json
File ./monochain/config.json does not exist
No config file found, configuration needed.

Would you like to create a new configuration file (y/n)?
WARNING - If you already have a config file this will overwrite it!
$y
```

Showcase of purple navigation bar and tabs section on white background and black text

Welcome to the MonoChain

The MonoChain is a new generation of decentralised computing built alongside the "Proof of Trust" protocol.

What is Proof of Trust?

'Proof of Trust' is a custom built consensus protocol that uses a 'trust' parameter in order to be faster and eco-friendlier than 'Proof of Work' whilst having much lower entry requirements than 'Proof of Stake'.

I trust you, so I'll process this data for you!

You sent me a valid proof. Here's some coins and I'll remember this!

MonoChain

- Home
- Wallet
- Setup a Node
- Learn More
- Contact

By Alfie Ranstead

Feature 3

- ✓ Create more detailed sections/pages of information for users who wish to understand the technical detail better.

There are various sections throughout the site that give users slightly more information on anything they may wish to learn about such as the proof of trust summary on the homepage, and there is an "info" page which contains some key questions and technical ideas to explain those in a lot more detail to users who are interested.

The screenshot shows a blue header bar with navigation links: Home, Learn More, Contribute, and Wallet. Below the header, the title "The MonoChain Project" is centered. A paragraph of text follows, describing the project as a decentralized network built upon the proof of trust consensus protocol. Below this, a section titled "What is the Proof of Trust Protocol?" contains a detailed explanation of the protocol's mechanics, comparing it to 'proof of work' and 'proof of stake'. Another section, "Why is the Proof of Trust Protocol important?", discusses the energy efficiency and decentralization benefits of the protocol. At the bottom, a section titled "So Why the MonoChain?" provides a brief history of the project's development and its purpose.

The MonoChain Project

The MonoChain is a decentralized network built upon the proof of trust consensus protocol, designed and built by Alfie Ranstead, a computer scientist with a strong interest in complex problems and their solutions. Blockchain technology is just a natural part of this group of technology, but it is far from perfect, and that is why the proof of trust protocol - and this project - exist.

What is the Proof of Trust Protocol?

The proof of trust protocol is a custom built consensus protocol built in order to define how nodes [computers that contribute to the network] decide how to trust or not trust other nodes. It uses a system of parameters used to store both grudges and appreciations for other nodes, alongside a few other values, which help decide whether or not to re-process whatever information other nodes have sent to it. A key addition here is that nodes will also choose to randomly reprocess data that they have received, in order to make sure that the seemingly 'trustworthy' nodes are still actually trustworthy and haven't been tampered with.

Why is the Proof of Trust Protocol important?

The two most used consensus protocols are currently 'proof of work' and 'proof of stake', both of these protocols are fundamentally flawed, and as such are not suitable for use in a widely adopted, decentralised system. The 'proof of work' protocol requires nearly all the nodes on the network to calculate the same data, only to reward a singular node for it per block, which is not only heavily favouring nodes comprised of more expensive hardware, but is also terrible for the environment by wasting massive amounts of energy. The 'proof of stake' protocol requires the nodes to put forward a large amount of money to be able to stake, which is more energy efficient, but extremely favouring of nodes that have large backings which defeats part of the reasoning for a decentralised system, as it just gives power to those who have money - which is against a lot of the ideas of the decentralised system. This is where proof of trust comes in, as it is a consensus protocol that is more energy efficient than 'proof of work', and although it is not as energy efficient as 'proof of stake', it does allow anyone to use and operate a node, removing the need for a large amount of money to be staked.

So Why the MonoChain?

The MonoChain is being developed as proof that behind all the hype, buzzwords and roller-coaster like evaluations of decentralised technologies, the technology can still be used for some cool stuff. This project is not aiming to be the newest trend in the tech world, but rather to be a proof of concept of how a new technology can still be useful and that it shouldn't be thrown away as just some get rich quick scheme. Oh also because I needed an A-Level Project and I thought it would be fun to do.

The info page

Error Tolerant

Feature No.	Description	Success/Fail
1	The Node should never crash completely, parts of it may stop working until the user comes to check on it but it shouldn't completely stop processing.	Fail
2	The Webportal should have no code-breaking bugs, with the only errors that make it through to production being styling/graphics based bugs.	Success

The questions from [usability testing](#) relevant to this section:

Question	Summary of Responses
Did any parts of the project crash?	60% of Users responded "lots", 40% responded "some", showing that all users encountered atleast one crash.
Have you found any errors/bugs in the node software?	80% of Users found atleast one error/bug in the Node software.
If so, what are they?	All bugs reported were in relation to the node software's messaging demo, with 50% referencing that errors occurred when lots of messages were sent in a short period of time.

Feature 1



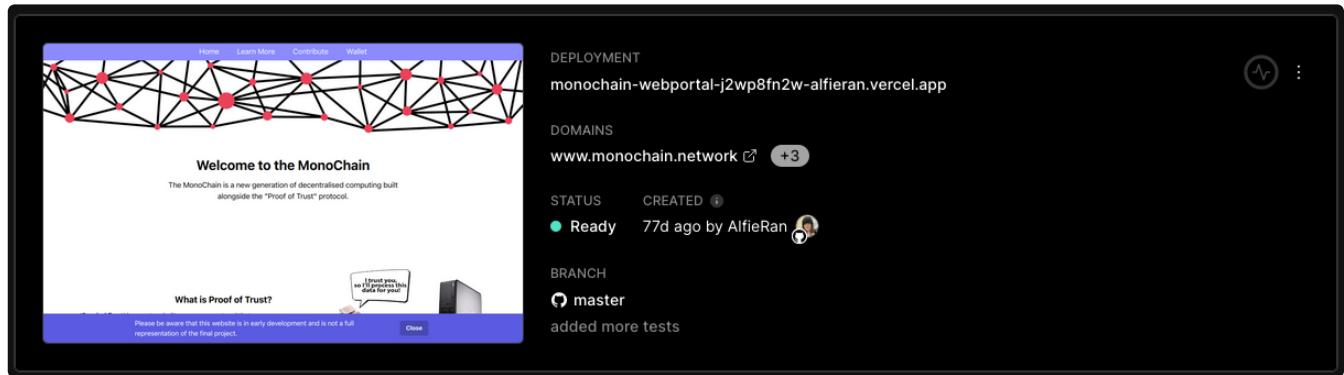
The Node should never crash completely, parts of it may stop working until the user comes to check on it but it shouldn't completely stop processing.

As shown in both the question responses from [usability testing](#) summarised above and the tests within the [checking of development](#) tests, the node software crashed under continued use or heavy load. This means that when more than 2-3 users tried to talk to each other using the messaging demo built upon the communication layer or when a user sent more than 1 message a second for atleast 10 consecutive messages their own or someone else's node would crash. This would need to be addressed within the short term fixes for this project but as I suspect that it is partially due to limitations in the language of which the node software is written in it would likely also become a long term limitation of the project.

Feature 2

- ✓ The Webportal should have no code-breaking bugs, with the only errors that make it through to production being styling/graphics based bugs.

The Webportal has never gone down after initial publication. This can be shown both by the lack of bug reports by testing users and can be reinforced by the fact that as of writing this documentation the last webportal update was pushed and uploaded 77 days ago and the server has not logged an error since.



Server hosting of the webportal

Some users may encounter client side exceptions due to losing connection whilst loading the site or other similar client side issues but simply refreshing the page will fix these aslong as this server stays up.

This has been accomplished due to the server having to be compiled from typescript and Next.js before it can be ran which catches errors that would otherwise crash the site and hence prevents the site ever having a non-working version being pushed to the public as the server will just run the most recent working version in the occourance of a faulty version being pushed.

Easy To Learn

Feature No.	Description	Success/Fail
1	Minimise the amount of user input required to operate the Node software.	Success
2	Categorise web-portal transportation methods to simple, easy to understand groupings such that users are not overwhelmed with choice but instead can make a few simple choices to get to where they need to go.	Success

Feature 1

- Minimise the amount of user input required to operate the Node software.

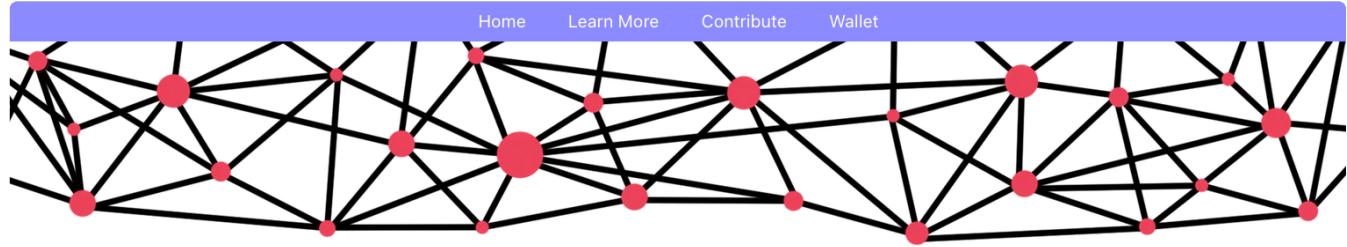
Assuming that the user wishes to setup a worker node all they are required to do is launch the software and run through some very basic inputs to generate the default configuration file and wallet/keys and then the software can be ran without any further input. This limits user input to very minimal levels.

If the user wishes to setup a leader node they have the same amount of inputs required to generate their configuration file and wallet/keys but they must then edit that configuration file to meet their networking setup but since this cannot be avoided without forcing leader node users to use specific networking setups it can be classified as the minimal level of inputs hence achieving this feature.

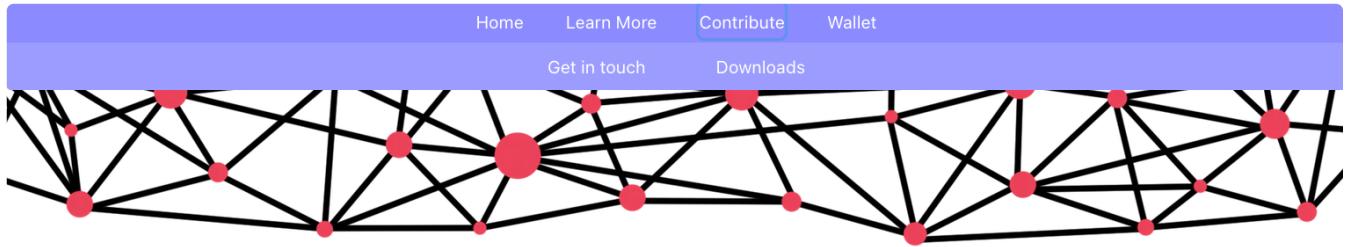
Feature 2

- Categorise web-portal transportation methods to simple, easy to understand groupings such that users are not overwhelmed with choice but instead can make a few simple choices to get to where they need to go.

Web portal navigation is grouped and on the desktop view does not show more than 4 sections per grouping so as to make navigation as simple as possible. This is done both on the home page and the navigation bar, with the mobile navigation bar showing 5 sections due to the fundamental difference in use case.



Desktop Navigation bar without any expanded sections



Welcome to the MonoChain

The MonoChain is a new generation of decentralised computing built alongside the "Proof of Trust" protocol.

Desktop navigation bar with contribute section expanded to show more sections

Use the Monochain

Wallet

Create and manage your wallet.

Learn More

Learn more about the MonoChain.

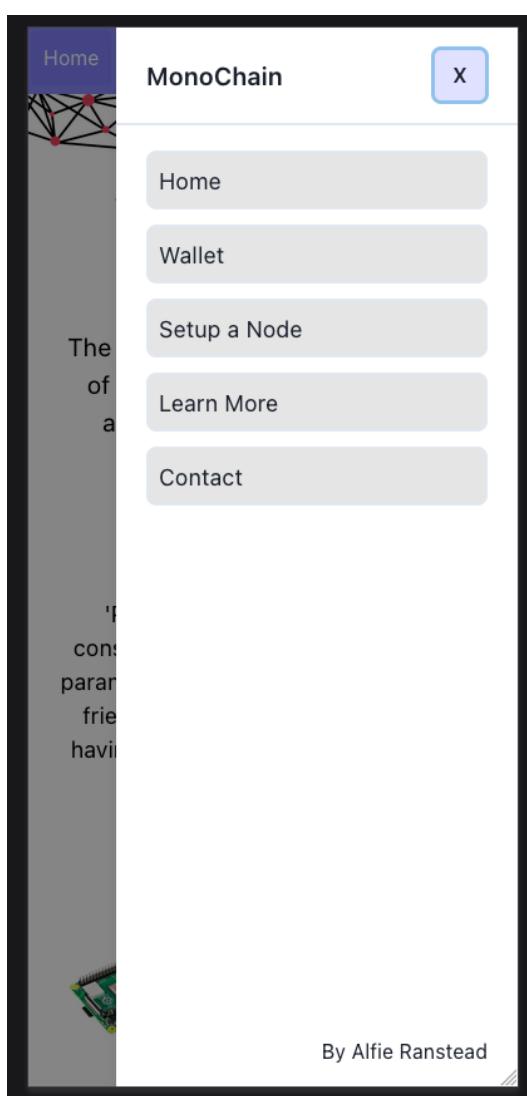
Set up a Node

Download a node to support the MonoChain in exchange for a fee.

View the Code

View the open source code for the default Node and Webportal.

Desktop Home page quick navigation options



A desktop browser screenshot showing the MonoChain website. The top navigation bar has 'Home' and 'Menu' buttons. The 'Wallet' section is displayed, featuring a purple header with the title 'Wallet' and the subtext 'Create and manage your wallet.' Below this are three light purple callout boxes: 'Learn More' (with subtext 'Learn more about the MonoChain.'), 'Set up a Node' (with subtext 'Download a node to support the MonoChain in exchange for a fee.'), and 'View the Code' (with subtext 'View the open source code for the default Node and Webportal.').

Home

Menu

Wallet

Create and manage your wallet.

Learn More

Learn more about the MonoChain.

Set up a Node

Download a node to support the MonoChain in exchange for a fee.

View the Code

View the open source code for the default Node and Webportal.

4.3 Maintenance and Future Development

Current Limitations of Project

The two most substantial limitations the project faced were the choice of programming language for the node software and the time in which I had to complete the development of the project. Although due to the fact that in order to progress much further past the point at which I reached in the development of the node software I would have to completely refactor the software into another language which would take time it can be summarised that the only real limitation is that of time, which sadly is a limitation that all projects have to face, although I will go into how the project could progress if it was given more time and that is what I shall do in the rest of this document.

Straight away we can dissect what limitations in the project's current state of development this language limitation has caused:

Node Software:

- Performance:
 - Crashes under continued or heavy use.
 - Limited by the computer/server that the software is ran on.
 - Primarily operates on a singular cpu core rather than all that a computer has to offer.
- Platform Support:
 - Must have websockets disabled in order to operate on Linux.
 - Is currently only precompiled for macOS ARM systems meaning that other systems have to spend longer setting up nodes.
 - Will only run on ARM and x86 systems (desktops/laptops) and not mobile devices.
- Usability - Very little documentation or explanations on how to use various parts of the software which raises the entry level of users who can actually use the software.
- Internet Access - The node software requires internet access to communicate with other nodes on the network.

Web portal:

- Features - Missing various features (wallet access, online market) due to node software not being far enough on in development to allow this to be used.
- Platform Support:
 - Only accessible on a browser with access to the internet.
 - Some parts of the site do not work on mobile or tablet devices.
- Internet Access - The webportal servers are ran on the internet and as such internet access is required in order to request the website from these servers and to view it.

The majority of these limitations can be assessed and changed in future development to meet the direction in which the project is growing.

Maintaining the Project

Based upon the project's current state one of two primary strategies can be followed depending on whether the aims for the project remained to complete all that was originally conceptualised and further expansion past that point or to optimise and fix what currently exists of the project so as to use it as some form of decentralised communication system as is done in the messaging demo.

Optimising it's Current State

In order to optimise the project for its current state then it would need to undergo the short term development plan alongside the optional aspect of rewriting the node software in a new language as according to the long term development plan if the plans for the project require a different level of higher stability that is out of range of what is currently possible with VLang without significant and perfect optimisations.

If the software was to be rewritten it should then be brought up to at least the standards set by the previous version whilst ensuring that it fixed all the problems addressed in the short term development plan whilst hopefully improving the stability significantly. If the software is not rewritten then it should still have had some form of stability increase but it is unlikely to be as significant as rewriting the entire thing would do.

After these issues have been addressed and the stability of the node software has been increased as mentioned in the above paragraph the project should then go through another, more thorough testing phase that includes separate usability testing by technical and non-technical users. This version of the project can then be implemented in whatever way was planned, whether that be simply the messaging demo being used as a decentralised chat room or something more advanced.

In order to do this, the project could be developed by a singular person or it could be completed by multiple developers working together, especially since all of the code of this project is open source and publically available on GitHub. This means that other developers could publish changes that are as small as fixing a singular bug they discovered or as vast as rebuilding a chunk of the project. If it was only a singular developer working on the project, the only maintenance that would need to be continually carried out past the point of publication would be ensuring the service is still working and fixing any bugs that occur following user reports. On the contrary, if multiple, open source, developers were contributing to the project, more comments would likely be required to explain the code in further detail so as to allow as many people as possible to be able to understand it and one or more developers would need to be dedicated to reading through and checking the other developer's contributions so as to ensure they aren't acting maliciously or accidentally breaking other parts of the project.

Growing into an Ideal Blockchain

Initially all of the optimisations and fundamental upgrades mentioned in the other strategy can be used so as to bring the project up to a stable level, with the requirement that the node software is rewritten in a different language that fully supports all that the project requires so as to ensure that there are no (or minimal) further issues with the node software that are not due to the project itself.

The initial short term bug fixes and even the node software rewrite could be completed by a singular developer if they were given enough time, however based upon the current size of the project and the knowledge that it is approximately half way to becoming a working blockchain and that the long term development plan expands past this, it is likely that additional developers would be needed. This could occur in two ways: A small team of dedicated developers, starting at 2-3 but could then expand further if the project continued to grow past this; and through a large body of developers contributing via the "open source model" which was mentioned briefly in the other maintenance strategy.

The small team of dedicated developers would initially require heavy training and assistance by myself such as to bring them up to speed with the inner workings with all of the code such that they understand it significantly enough to not cause issues for each other when working on the code, as the different parts of the project should integrate into each other significantly by the later sections of the long term development plan. The individual developers could also be designated to separate parts of the project so as to optimise their skill sets, such as one developer working to make the web portal as high quality as possible with the others working on the node software. As the code base expands it would also be important to ensure that the code is commented at an increasing rate since it would be getting increasingly complicated.

Alternatively, a larger team of open source developers could be adopted past an initial rewriting of the node software. This would require that all parts of the code are well documented and commented and that these comments/documentation are kept up to date as frequently as possible, this is because open source developers can contribute as little as a singular line change to patch a bug or as much as rewriting an entire module to optimise the software on the whole, therefore meaning that lots of people will need to be able to read and understand individual parts of the project without having to see any other part. It would also be important that as the number of developers contributing increased as with the size of the project, that there are designated developers who are given control over who's changes are actually contributed to the project and these developers would have to check through and run any code that has been changed in order to ensure that it does not break any other parts of the project and is not malicious.

Future Development

Short Term Bug Fixes

Bug Fixes

The first stage of continuing development for this project would be to fix the bugs that currently limit the project, these bugs are listed below.

- Webportal has visual glitches on some tablets or phones - To fix this, different components should be used for the node setup tutorial just like the navigation bar and the home page buttons should be configured to switch modes to mobile mode based upon the screen aspect ratio rather than the raw pixel width.
- Node software's dashboard doesn't work very well on mobile - To fix this, the dashboard and login screens should react to the size of the screen and change component sizes to make it more usable like on the webportal.
- Messaging demo isn't stable while using websockets - To fix this, the way in which the http and websocket servers interact should be changed to reduce the chance of triggering a memory issue (which currently happens due to Vlang's websockets module being very new and not as stable as I would like.)
- Linux Versions cannot use websockets at all - To fix this, I would look for some work around that does not cause the issues encountered or edit Vlang's websocket module further to fix my own problems like I did in [cycle 14](#), although this is very time consuming.

Documentation and Information

Following the bug fixes, the documentation and information throughout the project should be increased. It was found throughout usability testing that the individual sections of the project are not explained in enough detail and this needs to be expanded so that all users understand the very basic information about all parts of the project and understand which parts are aimed at them and which aren't.

Furthering on from this the node software needs significant improvement both internally and externally. Internally more commenting is required to explain how different modules interact with each other since this is somewhat lacking. Externally documentation should be written for all parts of the configuration handler alongside a general briefing on how the node software is doing so as to ensure users have a much more solid idea of what they are actually doing. Alongside this the login sequence for the node dashboard needs some basic information to explain step by step how to login and use the dashboard, alongside the dashboard having some additional information added to the page to further clarify what everything means.

Testing

Assuming both of the stages above are completed, a full and detailed testing should be completed similarly to the testing done in [section 3](#), with a significant difference being that user testing should be completed with a separate standard/technical user focus groups to see the difference between the two groups and ensure the users can accomplish what their group should be able to do.

If any tests or criteria that were planned to be met are not met at this stage then suitable bug fixes and improvisations should be made to pass those test and/or criteria. If not then development can be continued or halted as according to the maintenance plan.

Long Term Addressing Limitations

Rewriting the Node Software

One of the fundamental issues with the current project is the node software and the language it is written in, specifically the maturity of the language. Although I did do some research into VLang before I decided to rewrite the node software from typescript into it, most of the decisions were made on what I read on the language's website, although it turns out that most of the features and claims made by that website are intended for the first full release version, V1, and the language is currently in beta, at V0.3.2 as of writing.

This means that various features and the standard library modules are not as polished as I would like them to be, primarily the websockets module which has various issues including the incompatibility with its servers being treated as purely an object in memory and passed around through varying components as such, even though VLang allows this to happen and does not prevent alternative methods to access those servers from multiple places at once.

Therefore, Although rewriting the node software to perfectly match what is recommended in Vlang's documentation would increase its stability, it would likely not fix its underlying issues and may still allow for the software to not be as stable as would be required to meet the criterion.

The solution to this is then to rewrite the software in a different programming language, of which is not decided, except for the requirement that the language is fundamentally stable and is in a fully released production version, unlike VLang. Some options include Rust, Go, C and even going back to Typescript - although that would have the same issues as mentioned in cycle 2 since it is not primarily intended to be run directly on an operating system itself but rather through some kind of browser or replacement. If I were to have continued this project for a longer period of time I would have chosen to rewrite the node software in Rust somewhere around cycle 12, which is where I started to discover some of the more serious issues with VLang, since Rust is much more modern than C yet is supposedly so stable that it has been included in the Linux kernel, which is as high a praise as a low level language can receive.

Regardless, rewriting the node software into a more stable language is essential to expanding further onto completing all the initial goals of the project.

Producing a Working Blockchain

After the node software has been rewritten and the websocket and http communication methods have been proved to be working consistently, a solid, decentralised communication layer will have been produced, since the current code for the node software is essentially an unstable, decentralised communication network. This means that past this point the next step in the projects development is to introduce the protocol ontop of this layer, converting the messages from the messaging demo into transactions, grouping those transactions into blocks and building up the actual blockchain.

Then continuing onto introduce trust, grudges and everything else that is needed for the proof of worth system to work properly. After which the network should be secure enough that wallets and root state can be added, which allows the webportal features that were originally planned to also come into fruition.

At this point the network will have gained a reason to gain users, with the webportal's market and wallet viewer allowing non-technical users to purchase and sell digital contracts and items.

Introducing Additional Features

After the blockchain functionality has been introduced and the project has exceeded the original plan presented in this series of documents, additional features can begin to be introduced. These features can be based upon how the system is being used so as to best fit the users and encourage the growth and spread of the system, however a suitable next step may be a market exchange for users to be able to exchange between their monocoins and fiat currency by buying/selling them to other users, since this would allow users who do not wish to run a node to also gain access to the items on the blockchain and introduce 'real world value' to the items.

Additional features will need to be carefully decided so as to prevent the introduction of a flood of mediocre yet somewhat interesting sounding features that have no real use, and to ensure the security and stability of the system is not compromised.

Testing and New Versions

As the project expands and adds new features, it is important that all new code and changes are carefully tested and presented to users for their opinions before proper release. All Node software changes should also be grouped into major releases so as to prevent users having to download new versions all the time and should also aim to remain backwards compatible such that node servers can continue to run old versions of the software for very long periods of time without having to be interrupted. Although this is only required after the system has reached the point at which its contents have monetary value and the user count has reached a point at which it is clear that it is no longer in an alpha/beta state. Therefore only after changes have been carefully examined and all tests passed should new versions of the various parts of the project be released.

Future Limitations

After the project has reached this far expanded point, the points mentioned in the testing section begin to take effect, meaning that the limitations that the system faces will be somewhat changed.

- Backwards Compatibility - After the node software meets its planned state and the number of users has increased, it should be capable of communicating with and working with versions of itself that may be many months old so as to limit the disruption to those servers for as long as possible without sacrificing security and stability.
- Hardware Requirements - The project may reach a point at which the node software exists in multiple languages so that it can run in a variety of places, at this point the hardware requirements of the devices that run it may vary, but they will likely need to remain somewhat modern to meet the cryptography and memory requirements of the nature of the software.
- Platform Support - As the project gains users, it becomes increasingly logical to ensure that the node software is available for as many different systems as possible, so different versions of the software should be compiled and packaged for each release to support a wider array of systems.
- Internet Access - Both the web portal and node software will both still require internet access to operate and communicate with the network.

Reference list

- Abdullah-Othman, A., Alhabshi, S., Kassim, S. and Haron, R. (2020). *The Impact of Monetary Systems on Income Inequity and Wealth Distribution*. [online] researchgate.net. Available at: https://www.researchgate.net/profile/Adam-Abdullah-3/publication/340536565_The_impact_of_monetary_systems_on_income_inequity_and_wealth_distribution_A_case_study_of_cryptocurrencies_fiat_money_and_gold_standard/links/615977204a82eb7cb5e82d2e/The-impact-of-monetary-systems-on-income-inequity-and-wealth-distribution-A-case-study-of-cryptocurrencies-fiat-money-and-gold-standard.pdf [Accessed 7 Apr. 2022].
- Anon, G.A. (2022). *Decentralized Storage*. [online] ethereum.org. Available at: <https://ethereum.org/en/developers/docs/storage/> [Accessed 22 Mar. 2022].
- Anonymous, Blockchain.com (2022). *n-transactions-per-block*. [online] Blockchain.com. Available at: <https://www.blockchain.com/charts/n-transactions-per-block> [Accessed 1 Apr. 2022].
- Apple (2009). *Safari 4 for Mac - Technical Specifications*. [online] support.apple.com. Available at: https://support.apple.com/kb/sp552?locale=en_GB [Accessed 7 Jun. 2022].
- Chen, J. (2021). *Fiat Money*. [online] Investopedia. Available at: <https://www.investopedia.com/terms/f/fiatmoney.asp>. [Accessed 7 Apr. 2022]
- Cloud 66 (n.d.). Recommended minimum server sizes • Node. [online] help.cloud66.com. Available at: <https://help.cloud66.com/node/references/non-recommended-server-sizes.html#:~:text=We%20recommend%20the%20following%20minimum>.
- crypto.com (2022). *When the Peg Breaks - Deppegging Events in Fiat and Crypto*. [online] crypto.com. Available at: https://content-hub-static.crypto.com/wp-content/uploads/2022/05/20220531_CDC_1_When_the_Peg_Breaks.pdf [Accessed 7 Jun. 2022].
- Ethereum (2015). *State Tree Pruning*. [online] blog.ethereum.org. Available at: <https://blog.ethereum.org/2015/06/26/state-tree-pruning/> [Accessed 8 Jun. 2022].
- Firefox (2022). *Firefox 101.0 System Requirements*. [online] Mozilla. Available at: <https://www.mozilla.org/en-US/firefox/101.0/system-requirements/> [Accessed 7 Jun. 2022].
- GeeksforGeeks (2020). *Difference between RSA algorithm and DSA*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/difference-between-rsa-algorithm-and-dsa/> [Accessed 4 Sep. 2022].
- Google. (2012). *Chrome Browser system requirements - Google Chrome Enterprise Help*. [online] Available at: <https://support.google.com/chrome/a/answer/7100626?hl=en> [Accessed 7 June 2022].
- Hall, M. (2021). *What's the Difference Between a Monopoly and an Oligopoly?* [online] Investopedia. Available at: <https://www.investopedia.com/ask/answers/121514/what-are-major-differences-between-monopoly-and-oligopoly.asp#:~:text=Oligopoly%3A%20An%20Overview>
- Napoletano, E. (2022). *Proof of Stake Explained*. [online] Forbes Advisor. Available at: <https://www.forbes.com/advisor/investing/cryptocurrency/proof-of-stake/#:~:text=With%20proof%20of%20stake%2C%20participants> [Accessed 28 May 2022].

O'Neill, A. (2020). *Value of the US dollar 1635-2020*. [online] Statista. Available at: <https://www.statista.com/statistics/1032048/value-us-dollar-since-1640/> [Accessed 7 Apr. 2022].

Thornton, M. (2015), “Gold and economic inequality”, available at: <https://mises.org/library/gold-and-economic-inequality> [Accessed 7 April 2022].

Wackerow, P. (2022). *Proof-of-stake (PoS)*. [online] ethereum.org. Available at: <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/> [Accessed 20 May. 2022].

WebStorm Help. (2022). *Install WebStorm - Help | WebStorm*. [online] Available at: <https://www.jetbrains.com/help/webstorm/installation-guide.html#requirements> [Accessed 4 Jun. 2022].

Terminology

Backend Server

The server(s) that run the part of the internet you don't see, this is how facebook stores your user information, how youtube calculates what videos to show you next, and how WhatsApp sends your messages to other users.

Nfts

Non Fungible Tokens are items stored on a blockchain that are not currencies in themselves, currently they tend to take the form of images or "art" and are mainly used for monetary gain. However this technology will be very important to the development of platforms that run on "[Web 3.0](#)" and should grow to be used for actually useful purposes other than just "investments" in the future. (An example of a more elegant use of nfts is the ownership certification for a piece of content on the internet - a [web3.0](#) platform could use this certification to calculate where to send advertisement revenue generated from the content)

Nodes

A Node is a server or computer that contributes to a network, typically through 'mining' for the blockchain which is a term used to describe validating blocks and transactions up to the schema required by the protocol of the network.

ROI

Return on Investment, how much someone earns relative to how much it cost them to earn that amount.

Web 3.0

This is a buzzword used to cover all types of websites/software that either are or are built upon blockchain technology, this includes things like [nfts](#), cryptocurrencies and more recently games that are built using nfts as in game items. There's a lot of argument on the internet about whether or not this is just a buzzword meant to hype people up and get them to buy cryptocurrencies/[nfts](#) so as to benefit other "investors".