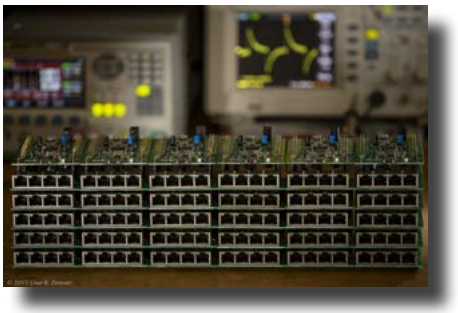
Australian National University

HARMONY & MONEY



Benjamin J. L. Wang & Uwe R. Zimmer

Assignment for all students of
Systems, Networks and Concurrency

This is the second, carefully marked assignment of the course. Extra care is expected on all levels. A good solution will have a surprisingly small amount of code. Make sure that this code is excellently written – in whichever language you prefer.

Building Blocks

Throughout the course, you have seen different types of distributed systems. The challenge with such systems is for distributed nodes to reach an agreement about the state of the application, even when the nodes all have different clocks, geographical locations, and latencies. It gets even more challenging when you consider that messages can be dropped, internet connections are not reliable, and nodes sometimes crash unexpectedly. Distributed systems are all about reaching consensus despite these challenges.

Distributed databases are a common application of distributed systems. The goal in these databases is usually achieving consistency (where all nodes have the same view of the data), availability (where data can always be read and written), and fault tolerance (where failures in nodes do not cause failures in the system overall). To achieve these goals, it is critical that the distributed nodes are able to reach consensus on the sequence in which transactions have happened.

In these systems we have made a subtle assumption — one that often goes unnoticed. It is assumed that the nodes can be trusted, because they are often run and maintained by some central authority. To remove this assumption, we must ask ourselves some difficult questions:

- What happens if the nodes are maintained by any member of the public?
- What happens when no central authority maintains the nodes?
- Why would any one want to design a distributed database like this?
- Is it still possible to reach consensus and provide consistency, availability, and fault tolerance?

In this assignment, we will explore one solution to this problem: **Blockchain**.

Blockchain

A blockchain is just another type of distributed database. It is a record of transactions replicated over many different nodes. These nodes — run by any member of the public — are known as **Replicas**, because each of them maintains a copy of the entire record of transactions. To know the current state of the blockchain, a replica must replay every single transaction that has been recorded in the blockchain. To guarantee consistency, there needs some way to ensure that all replicas are replaying the same history in the same sequence — otherwise each replica might end up with a different view of the state.

Decentralisation and trustlessness

Because replicas are not run or maintained by a central authority, and anyone can join the network; no guarantees can be made about any specific replica. No one can control the location of replicas (often they end up distributed all around the world), or the reliability of its internet connection (some people run them on the cloud, but some people run them from their laptops). To make matters worse, no one can enforce replicas to run the expected software (someone could be running a modified version in an attempt to attack the database for personal gain, or because some people

just want to watch the world burn). Despite all of these challenges, a blockchain must still provide a distributed database that is consistent, available, and fault tolerant.

But there is a reason these challenges are worth solving, and why blockchains have received so much attention: **decentralisation** and **trustlessness**. These two properties are the distinguishing features of a blockchain.

Decentralisation means that no small number of machines, people, or corporations controls the blockchain. There can be no control over the censorship and freedom of information, no limitation of access, and no favouritism. Perhaps more relevant to current affairs: no person, nor corporation, owns the data of the users.

Trustlessness means that the correctness of the blockchain (e.g. the consistency and correctness of data, its availability, and any other feature it attempts to offer) does not require any specific replicas to act honestly. Instead, you trust that some quorum of replicas are honestly following the agreed protocol (but you do not have to trust any specific quorum). Often, to further ensure that there is an incentive to behave honestly, the agreed protocol will define economic incentives for honest participation — **Cryptocurrency** — because people (and corporations) often try to maximise their financial position, and rarely provide services for free.

To achieve both **decentralisation** and **trustlessness** is a feat of precise technical, economic, and political design. But the pay-off is huge: a decentralised database that can be used by anyone, correct for everyone, but controlled by no one.

What's with the blocks?

It is critical to ensure that all replicas reach consensus of what transactions have been recorded, and the sequence of these transactions. To do so, transactions are grouped together into **Blocks**. Replicas will collect transactions from users and produce these blocks, propagating them throughout the network — allowing everyone to see the block and replay the transactions within.

To ensure that all replicas agree on the sequence of transactions, a block must reference exactly one **Parent Block**. In this way, blocks create a **Chain**. Now the goal is to understand how Replicas reach consensus on which chain is considered to be the **Canonical Chain** — a truthful and unanimously agreed upon record of transactions.

Consensus

Reaching consensus on the canonical chain is done differently by different **Blockchain Algorithms**. But, all **Blockchains** address two common issues:

a. Determine when a replica is allowed to produce a block.
b. Determine which chain to choose between when there are multiple different chains.

Bitcoin, the most popular and well known blockchain, uses an algorithm called **Proof of Work**. It is quite simple:

a. To produce a block, a replica must solve a hard cryptographic challenge based on the block (and its parent).
b. When choosing between multiple valid chains, a replica must always choose the **Longest Chain**.

Bitcoin assumes that at least 2 out of 3 Replicas are honest. This is originally claimed that only 1 out of 2 Replicas needed to be honest, which has continue to be commonly wrongly cited.

Assignment 2: Aura

Bitcoin's *proof of work* is the first known blockchain algorithm, and has been battle tested for the last 10 years. But, among other criticisms, it is regarded as inefficient because of the energy expenditure required to solve the cryptographic challenge. In this assignment, you will be investigating a category of much less computationally heavy blockchain algorithms: **Proof of Authority**.

In *proof of authority*, a specific set of replicas is given the authority to generate blocks. These replicas are known as **Authorities**, and in many *proof of authority* algorithms it can change over time. *Proof of authority* algorithms are distinguished by how replicas are chosen to be authorities, and how one specific authority is selected to generate a block.

In this assignment, you will be building an **Aura Blockchain**. It is a simple, but powerful, *proof of authority* algorithm used today in the blockchain industry. Aura assumes that at least 1 out of 2 Replicas are honest.

Authorities

In Aura, the set of replicas chosen to be authorities is decided ahead of time and it is static — it does not change over time.

We define N to be the number of authorities, and we label them from 0 ... $N-1$. It is important to note that this means the authorities are labelled in a sequence, and everyone in the network agrees on the order of this sequence ahead of time.

Steps

Before discussing how an authority is selected to produce a block, we need to discuss time. In Aura, time is broken down into discrete steps.

But how much time is in a step?

This depends on many different factors, and you should experiment with different time steps in your assignments to see what effect it has. Since time deviates slightly between different machines, and network latency creates delays in information propagation, steps that are too small will prevent replicas from agreeing about the time at which something has happened. However, steps are also the smallest granularity at which replicas can discuss time, so steps that are too large results in a slower blockchain.

The current step is defined as s and is given by $s = \lfloor T/q \rfloor$ where T is the time since Unix epoch (in seconds) and q is the duration of a step.

Selecting an Authority

At step s an authority is selected from our ordered set of authorities. We select the authority with label $i = s \bmod N$ (with N being the number of `Authorities`). The label of an authority is either the index in the array of `Authorities` or the position of the authority provided by `Authorities'Pos`.

When receiving a new block, replicas will check that the timestamp of the block is reasonable with respect to the current time and that the authority that produced the block was the selected authority at that time.

Selecting a Blockchain

If an honest replica finds that it must choose between two valid chains, it should always pick the chain that is longest. If the two chains are the same length, then the choice does not matter.

This is critical to ensure that replicas ultimately achieve consensus on the same chain. It is also critical to ensure that a minority of dishonest replicas can not convince a majority of honest replicas to change their minds (resulting in reverting consensus that has previously been reached).

Have a think about why this works, and remember, it is assumed that at least 1 out of 2 replicas are honest.

Byzantine Fault Tolerance

So far so good, but what happens when an authority decides to misbehave. Remember, a blockchain should not assume that any specific replica is honest. It is completely plausible that an authority will attack the blockchain. Trust no one!

How can we ensure that authorities only propose valid blocks, and that they only do so when it is their turn to do so? The answer is simpler than you might expect: you can't.

Instead, honest replicas will just ignore blocks that are invalid, or produced out of step. As long as an honest replica checks all blocks for validity, using the rules below, it cannot be tricked into accepting a bad block.

The rules for a valid block are as follows:

a. The block must only contain valid transactions (in your assignment, blocks are empty so you can assume that this is always true).
b. The block height must be exactly 1 greater than its parent block height.
c. The block timestamp must be later in time than its parent block timestamp.
d. The block timestamp must be in the past.
e. The authority that has signed the block must match the step (where the step is calculated from the block timestamp).
f. The parent block must be valid (this means validation of a block is recursive).

Who produced what data?

For the observant, there are two questions that have gone unanswered[1]:

a. How can we be sure about which authority produced a block?

[1] In classic centralised distributed databases there is usually one master and multiple backup slaves that replicate all data. The master processes all transactions and copies them to the slaves. In the case of the master failing, a slave is promoted to be the new master.

b. How do we make sure the data has not been maliciously mutated?

Understanding the answers requires some cryptography which we will not be going into in this assignment! For the curious:

- To identify authorities with certainty, asymmetric cryptographic signatures are used. The entity holds private key that can produce an unforgivable signature. The general public holds a public key that can verify the signature. Typically, ECDSA secp256k1 elliptic curve cryptography is used.
- To prevent data mutation, a cryptographic hash of the data is produced. This hash is signed by the producer, and if the data is mutated it will no longer produce the hash that has been signed. Typically, SHA2/3 hashes (or variants) are used.

In many ways, this is actually a *proof of authority* algorithm. The only differences are that (a) transactions are not grouped into immutable blocks, and (b) the selection of authorities from available replicas does not make it reasonable for the decentralised world[2].

To prevent users DoS attacking a blockchain, and to protect it against overuse from selfish users, a user that wishes to get a transaction onto the blockchain must spend cryptocurrency with every transaction. Replicas that produce blocks are rewarded with cryptocurrency (you cannot rely on replicas to provide the service for free). In some cases, if a replica can be proven to have acted dishonestly it will be punished by losing cryptocurrency.

This provides an economic incentive to act honestly, since the reward led cryptocurrency can be sold to users of the Blockchain.

Building Blocks: Tasks

Framework

Before beginning the assignment, make sure that you take the time to familiarise yourself with the framework. It exposes many useful structures and methods that will help you complete the various stages of the assignment. It also offers several parameters that can be changed to test your replicas.

[2] So what's all this business about cryptocurrencies? A cryptocurrency is something built into the transactions and blocks of a Blockchain — it is a fundamental requirement to protect Replicas and users.

Structures

`Replica` is a partially complete implementation of an Aura replica. This is where all of your code will be written. The framework will turn on multiple instances of the replica and querying them to see if they are behaving correctly.

`Block` is a structure representing a block in a blockchain. It is the data structure sent between replicas and familiarising yourself with the fields inside is critical to understanding how to validate blocks.

`Blockchains` is used for storing, updating, and querying the longest blockchain known to a replica. It can also be used to blocks that been seen, even if they are not in the longest blockchain. Each replica has its own blockchains store.

`Block_Generator` (`blockGenerator`) is a communication channel that the replica uses to communicate with itself. Whenever it is time for a replica to generate a block (according to the current step, and its position in the ordered set of authorities), this channel will become active and signal to the replica that it is time to generate a block. This signalling is done for you by the framework! All you need to do is make sure your replica knows how to receive the signal.

`Block_Receiver` (`blockReceiver`) is a communication channel on which a replica can receive blocks. Blocks received in this channel need to be validated by the receiving replica, and could result in an update to its view of the longest blockchain. Each replica has one block receiver.

`Block_Query_Receiver` (`blockQueryReceiver`) is a communication channel on which a replica can receives queries for blocks. A query will contain a request to see a range of blocks from the longest blockchain known to the replica. Each replica has one block query receiver.

`Send`, `Query` (`Connection`) are simulated network connections to another replicas. It can be used to send blocks to other Replicas (and these replicas will receive them on their block receiver). Each replica will have multiple connections to other replicas. Connections are not guaranteed to be alive, and sending blocks over a dead connection will block indefinitely. Beware!

Functions

`Random_Block_Header` (`randomBlockHeader`) will generate a random block header that you can use when generating new blocks.

`Sign` (`sign`) will produce a signature for a specific block, from a specific Replica. This signature must be attached to each block produced by a Replica.

`Authority_From_Signature` (`authorityFromSignature`) does the inverse and extracts the authority out of a signature (this may raise an exception if there is no proper signature).

`Genesis_Block` (`genesisBlock`) will provide the starting block for your blockchain.

`Current_Block_Time_Stamp` (`currentBlockTimestamp`) delivers the current time in your network (in total seconds).

`Longest_Blockchain` (`longestBlockchain`) will derive a single blockchain out of your blocks storage, starting backwards from your End_of_Longest_Blockchain (`endOfLongestBlockchain`).

Commandline Parameters

`-s` (`step`) defines the number of seconds in each step. Experiment with different values for this parameter and see how it affects the blockchain agreed upon by your replicas.

`-a` (`attack`) tells the framework to try and send bad blocks to your replicas. This is useful for testing your block validation logic. We may chose to extend this feature with more sophisticated attacks for the marking of the assignments, so make sure you actually covered all cases, and not just the ones currently tested.

`-c` (`connectivity`) defines the connectivity between replicas in a randomly generated mesh network. At 0.0, replicas are not connected to each other at all. At 1.0, replicas are connected to all other replicas.

Stage A

In Stage A, you will not need to worry about communication between replicas. Your replica implementation must do two things:

a. Generate a new block when it is the appropriate step. Since this block should always be appended to the longest blockchain known to the replica, generating a new block should always result in an update to the longest blockchain known to the replica.
b. Receive and respond to queries about what the blocks in the longest blockchain known to the replica.

Doing this will require a basic level of concurrency. Your replica will not know what it will need to do next: generate a block, or respond to a query. Generating new blocks will also mutate state that is needed when responding to a query; some protection is needed to ensure that data races do not happen!

Stage B

In Stage B, you will need to improve your Replica implementation. After generating a new block, your replica should send this new block to other replicas. Likewise, when a replica receives a new block from other replicas, it should validate it and — if necessary — update its view of the longest blockchain known.

In Stage B, you can keep your connectivity parameter set to 1.0. This will ensure that all replicas are connected to all replicas, and there is no chance that a connection will be dead.

To test that your implementation is working, try enabling the attack parameter. This will cause the framework to try and send bad blocks to your replicas. If you see a bad block, then you know there is something wrong! (Remember, testing is never perfect so even if you do not see a bad block there might still be something wrong!)

Stage C

In Stage C, it is time to modify the connectivity parameter. Try setting it to 0.25 and see what happens!

Chances are, your replicas no longer reach consensus on what the blockchain should look like. You will need to modify your replica implementation so that they are still able to reach consensus.

To get started, consider some of the following questions:

a. What happens to your block validation logic if there are missing blocks?
b. How can you decrease the likelihood that blocks are missing?
c. If two replicas are not directly connected, but they are both connected to a third replica, how can you ensure that blocks generated are seen by all three replicas?

As you might expect, there are lots of different ways to resolve these issues and get your replicas to reach consensus again!

Deliverables

You do *not* need to provide a rationale or any report of sorts if you believe that your design is obvious and a few comments inside your code will make the idea sufficiently clear.

Yet you need to provide two items:

a. The sources for a working replicas (and for the students not using any of the provided frameworks: also an environment to test them in). Zip up the `Aura` directory (which should hold all sources which you manipulated or added) for submission.

b. A graphical representation of the workings inside your router (as a pdf file named `Diagram.pdf`). Add this file to your `Aura` directory before you zip it up.

In any case, we will read your sources carefully. The core part of your replicas will be small – take your time to structure this part well.

Different entities in your graphical representation will have different meanings: Use different lines / arrows / colours / shadings / etc. to indicate what is what and provide a legend. Before you start drawing: decide what the main aspect of our diagram should be. Will you primarily provide a call-graph / data-flow-graph / synchronization-graph / dependency-graph / etc. or a mixture of multiple aspects. If you need multiple diagrams for multiple aspect, then please provide multiple diagrams (in the same pdf file).

There is no predefined template for your diagram as the idea is that you communicate your concept with it – whichever format supports this best is for you to decide.

Tutor-eyesight-health-warning: Make sure that your graphics will appear technically in vector (scalable) format inside your pdf file – *never* as an image made out of pixels!

Criteria

When you design your system, you might want to keep the following criteria in mind:

- *Assumptions*: find the assumptions under which your design will still reach consensus.
- *Time*: How long does it take to reach consensus?
- *Traffic*: what is the communication overhead which your design entails? Beware of combinatorial explosions.
- *Performance*: what is the overall effort to reach a solution?
- *Elegance* of the solution. Elegance is often a measure related to compactness and simplicity but not always. You will see whether your design is elegant once you reevaluate/ scrutinize your sources a few days after you achieved a running system.
- *Clarity* and *expressiveness* of the provided graph. A meaningful graph will use a clear legend of graphical elements to distinguish different issues.
- *Robustness* of your design. Is your system water-tight, even if somebody tries really hard to break it?

Marking

Stage A might get you to a pass mark, while stage B may reach credit territory. For a (high) distinction, you will need to go all the way and present a solid stage C solution (and present it very well).

Outlook

Students in Real-Time and Embedded Systems (4th year course) will be working with actual hardware and concurrent and distributed algorithms under strict real-time constraints. There is strong demand for outstanding professionals who can handle high integrity systems interacting with the physical world. Your course provides many foundations for any such career – academic or industrial.