Applied Learnings for Adverserial Distributed Systems

Methods, Explorations for everything blockchains

BY

Supragya Raj

PUBLISHED IN THE WILD

Contents

Chapter 1

ZKVMs: Zero Knowledge Virtual Machines

This chapter intends to give a brief overview of the zero-knowledge virtual machines and their current state of usability.

1.1 RISC-V as a common target

As soon as we find ourselves exploring ZK-VMs, we quickly find ourselves with a bunch of projects such as RISC-Zero, Succinct's SP1, Mozak, Polygon Miden, Jolt etc, all of which for some reason are RISC-V compatible. What that means is that they expect input instructions to be composed of RISC-V instruction set.

1.1.1 Why RISC-V though?

Modern compiler infrastructure such as that of LLVM makes the code generation process modular. A lot of programming languages can be supported if we target RISC-V, for example C++ through Clang, Rust through RustC, Golang through TinyGo can all convert into LLVM bytecode. This can at the end be compiled down to a well knows small instruction set target such as RISC-V. RISC-V essentially is the 5th reduced-instruction-set-architecture that came out of UC Berkeley; hence the name. The main reason however that RISC-V becomes a compilation target of interest is because of two simple reasons apart from language-support: **minimalism** and **modularity**.

Minimalism in RISC-V: RISC-V is relatively simple ISA. It has 32 registers that can hold 32 bits each (`x0`-`x31`) alonside a special register `pc` that holds the address of currently executing instruction's address. The base 32-bit version of RISC-V dealing in integers is `riscvi` and has well-known simple instructions like `ADD`, `JAL`, `BNE` etc. All of these are relatively simple to describe.

Modularity in RISC-V: RISC-V provides a base, but also provides a bunch of useful extensions that one can opt-in to. For example, `+m` provides functionality such as multiplication (and division), `+f` provides floating point numbers, `+a` gives access to atomics among others. This modularity allows us to choose the tightest subset of features to implement for a ZKVM.

RISC-V compatible ZK-VMs are mostly targeted towards `riscvi+m`.

1.2 The general design of ZKVMs

1.3 Writing constraints for these instructions

1.4 Self-modifying code and dealing with it

1.5 Identifying the program

1.6 Proof generation without pre-compiles

While the numbers described below present the situation as seen in December 2024, they still give good information on what we can expect in the signature verification regime. ED25519 is one of the signature schemes used quite commonly in blockchains like Solana, Tendermint-based chains like Cosmos as well as Polkadot.

Test Description	**Execution Trace Generation Time**	**RISC-0 Segments Count**	**Total Cycles**	**User Cycles**	**Proving Time**
ED25519 1 signature no-precompile 3-byte msg	91ms	4	3.67M	3.12M	130s
ED25519 1 signature no-precompile 3000-byte msg	91ms	4	4.19M	3.48M	147s
ED25519 2 signatures no-precompile 3-byte msg	171ms	7	6.81M	6.06M	242s
ED25519 2 signatures no-precompile 3000-byte msg	191ms	8	7.60M	6.78M	270s

Analysis of the above gives us the following key points:

- The proof generation procedure is undertaken in two stages: a simulation phase where the execution "trace" is generated and the proof generation phase where the execution trace is proven.

- Compared to the actual proving time in each which includes both the stages, the first stage (execution "trace" generation) takes miniscule amount of time. For example in the test running 1 ed25519 signature verification in zkvm, 91ms were taken in execution "trace" generation step while around 130,000ms were taken for proof generation.

- On average, on a very beefy machine without GPU, Risc-0 can clock in around **28,000 instruction / seconds** of proving.

- In large calculations, the verification is broken down into multiple smaller proving segments, limiting on average, proving time **30s / segment**, **8GB RAM usage / segment**. Maybe we can use parallelization of these segments for faster proving, but we need to see who "proves things". Also, segmentation of proof provides larger overhead in proof accumulation and continuation checks not accounted for in these benchmarks.

Another important test we run is for a hash function: SHA256. Alongside ED25519, SHA256 becomes the primary underlying algorithm used extensively in chains like Solana

Test Description	Execution Trace Generation Time	RISC-0 Segments Count	Total Cycles	User Cycles	Proving Time
SHA256 1 byte	3.21ms	1	131K	8.9K	4.5s
SHA256 10 byte	3.25ms	1	131K	8.9K	4.5s
SHA256 100 byte	3.39ms	1	131K	15.4K	4.62s
SHA256 1,000 byte	4.9ms	1	262K	100K	9.26s
SHA256 10,000 byte	20.5ms	1	1.05M	0.96M	37s
SHA256 100,000 byte	175.43ms	10	10.48M	9.5M	368s

1.7 Proof generation with pre-compiles

A few other things we learn from the above experiments:

- We find that zkVM needs to run at least 2^{17}**cycles** of instructions regardless of what our code does. If instructions are less, the rest of the rows are padded, which amounts to extra zkvm cycles.
- The growth of total cycles required for execution grows linearly with amount of bytes in question.

Chapter 2

Version Control using Git

2.1 Using Git via SSH

We can use GIT via a specific SSH command override, useful for when we want to use a specific key for `git clone` or `git push`. This can be done via:

```
GIT_SSH_COMMAND='ssh -i ~/.ssh/your_private_key' git push
```

Chapter 3

Time, Synchrony and Node Failures

This chapter focuses on establishing the foundational building blocks for talking about distributed systems in general. This chapter first establishes key definitions and scope of problems at hand. Later, explorations are presented while arguing about different trade-offs.

3.1 Key defintions

We begin by setting the stage up for problem at hand. In distributed systems, almost always, we are dealing with a distributed set of nodes that wants to present themselves as a single entity to the client. While earlier, this distribution of nodes was done in the efforts of increasing the uptime of the system (if one node dies, the other node can stay alive and service requests); in modern times with blockchain setting, the same is desirable for a different reason: decentralization (no single node or small set of nodes can disrupt the entire system).

For all further discussions, assume there to be N different **nodes** in some distributed system S. We can then denote each node as $n_0, n_1, ..., n_{N-1}$. A client C communicates with S by communicating with one of the nodes in S, as such each node responds to C as if being a spokesperson of S, i.e. it presents the services to the client according to the "global view" of the distributed system. Each such node $n_i \in N$ maintains a local *append-only* data structure maintaining the history of "actions" / "transactions" it has seen since start of the protocol which should mimic closely the "global view" of the system. We can call it the "local view" of n_i. The content of messages can be arbitrary but are chosen from a message space M, all possible messages.

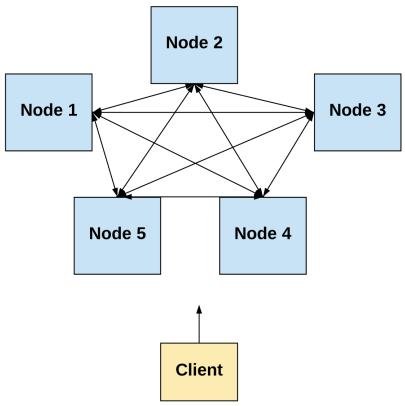

Figure 3.1: An example distributed system with 5 nodes. It may not be necessary that all nodes are connected directly to each other, but all nodes together present themeselves as a single unit to server client.

3.1.1 Consistency and Liveness

We begin our analysis of a distributed systems by describing what properties we intend to see from such systems. In very crude terms we want the following two properties:

- **Bad things don't happen / consistency**: We want all nodes to agree on the common history, i.e. no two nodes have a conflict over which transaction happened first. This is especially significant if two transactions t_1 and t_2 both intend to consume a single resource, A `write(a, 20)` followed by `write(a, 40)` is not equivalent to `write(a, 40)` followed by `write(a, 20)`.

- **Good things happen / liveness**: We want *valid* transactions from users to be accepted by the system. No user should be left starving.

Of course, these are strong claims, and we need to quantify the amount of consisteny and liveness that we desire. Here are some quantifications for the same:

- **Strong consistency vs Eventual consistency**: Strong consistency presents with some upper bound time guarantee δ after which it is assured that each node in the network will have the right view of some transaction, i.e. if a transaction t was included at time T, then after some $T+\delta$ all nodes have a consistent view of the status of the transaction. Often, a weaker version **eventual consistency** is however more practically feasible though. Eventual consistency provides similar guarantee of global view of transactions, however without some finite time bound δ. This can also be thought of as consistency where $\delta \to \infty$.
- **Strong liveness vs Eventual liveness**: Strong liveness presents with some upper bound time guarantee δ after which it is assured that client C's transaction will be included in the *append-only* history of S, i.e. if a transaction t was presented by client C at time T, then after some $T+\delta$, S includes it. Often, a weaker version **eventual liveness** is however more practically feasible though. Eventual liveness provides similar guarantee of liveness, however without some finite time bound δ. This can also be thought of as liveness where $\delta \to \infty$.

To elaborate on the above, let's take the following example:

A distributed system S consists of 3 nodes $N = \{n_0, n_1, n_2\}$. The client C can interact with any one of the three nodes in N. Assume the local ledgers of all three nodes are empty, as in:

$$n_0 : \texttt{local view} = [\phi], \texttt{known unprocessed txs} = \{\}$$

$$n_1 : \texttt{local view} = [\phi], \texttt{known unprocessed txs} = \{\}$$

$$n_2 : \texttt{local view} = [\phi], \texttt{known unprocessed txs} = \{\}$$

Later, C presents transaction t to n_2, hence the state becomes:

$$n_0 : \texttt{local view} = [\phi], \texttt{known unprocessed txs} = \{\}$$

$$n_1 : \texttt{local view} = [\phi], \texttt{known unprocessed txs} = \{\}$$

$$n_2 : \texttt{local view} = [\phi], \texttt{known unprocessed txs} = \{t\}$$

We want the final state of the system, if it is acting as a single unit to settle at the following configuration:

$$n_0 : \texttt{local view} = [t], \texttt{known unprocessed txs} = \{\}$$

$$n_1 : \texttt{local view} = [t], \texttt{known unprocessed txs} = \{\}$$

$$n_2 : \texttt{local view} = [t], \texttt{known unprocessed txs} = \{\}$$

Here, the movement of t from "known unprocessed txs" to the "local view" of all nodes in the system constitutes "liveness". However, this happens in stages. First, n_2 moves t from "unprocessed" to "local view". Later, it ask all the other nodes to updates themselves with his local view. This aspect of other nodes updating their local view to match n_2's local view constitutes "consistency".

3.1.2 Node honesty

In distributed systems, we generally deal with different kind of node faults. We call a node "faulty" when it deviates intentionally or unintentionally from the regular, online, honest participation in the network.

The following are the different kind of faults categorized:

- **Crash fault**: When some node n_i behaves honestly from some origin time t_0 to some aribitrary time t_c, after which it crashes / goes offline. This can be further subdivided into nodes that present a **Fail-stop** failure and nodes that present a **True crash fault** . Nodes that fail-stop halts, notifies everyone of its death and does not respond to any further messages. The nodes that experience true crash fault do not notify on the other hand. Evidently, true crash faults are harder to detect and manage as the rest of the network has to infer through indirect means the halt of the node.
- **Omission fault**: Omission fault is when a node n_i does not present required messages that it should have presented in an ideal setting to the rest of the network. This could be acknowledgements, votes or any other message. It is to be noted that omission of a message may be due to unreliability in the underlying network instead of a malice at the node's end.
- **Byzantine fault**: Byzantine fault is where a node n_i can behave arbitrarily, i.e. we can assume nothing about the node's behavior. Some of the actions exercised by such node could be: behaving like an honest node for some time range, behaving like a crashed node for some other time range, omitting messages, sending deceptive messages in hopes of disrupting the consensus. It should be noted however that we still assume the cryptography is not broken, as breaking that would mean in a system secured by public-key cryptography, the byzantine node can imitate any other node with no restrictions.

3.1.3 Network Synchrony

3.1.4 Consensus

The replication problem in distributed systems could be summarised as follows. Assume a node n_i has a local view $v* \in V$ with V being the set of all possible local views. A honest n_i would intend to disseminate $v*$ to all the other nodes.

Consensus in distributed systems is defined as any algorithm A that satisfies the following properties:

- **Termination**: All non-faulty processes eventually decide on some value $v_i \in V$.
- **Agreement**: All non-faulty processes decide on the same value $v \in V$, i.e. $v_i = v \forall n_i \in N$.
- **Validity**: All non-faulty processes decide on a value $v*$.

It is important to understand that all the above properties are for "non-faulty" processes, a.k.a. only the honest processes. See 3.1.2 for definition of node faults.

It is also worthwhile noting that aiming for any 2 out of the above 3 properties does not present us with any useful protocol. This can be argued as below:

If we only aim for *termination and agreement*, all the nodes at each step may decide on some hardcoded value $v' \in V$.

If we only aim for *agreement and validity*, all the nodes at each step will try to convince every other node to accept their value v_i and never respond to other's messages. The consensus halts when all nodes are convinced. Unless all nodes started with the same value v, the consensus never halts.

If we only aim for *validity and termination*, all the nodes at each step will decide on their own local v_i.

3.1.5 Permissioning

3.1.6 Global Clock

3.1.7 Public Key Interchange Infrastructure

3.2 State Machine Replication

The **State Machine Replication** problem is to provide a solution that presents a fault-tolerant distributed system where nodes within have the ability to replicate the local state they hold between themselves. The classic examples

here would be database replicas.

To solve for the SMR problem, we need to argue about the following things:

- **Node honesty**: What kind of

3.3 Dolev-Strong: Byzantine-broadcast, SMR, in a synchronous world

3.4 FLP: Can't handle even 1 faulty node in asynchronous world

The domain of distributed systems say a major breakthrough in 1978 when Leslie Lamport in his paper "Time, Clocks, and the Ordering of Events in a Distributed System" introduced the concept of logical clocks. Effectively, this paper presented an affirmative answer to the following question:

> **1978 Leslie Lamport.** Can we impose total ordering to events received by different nodes in a distributed system?

As a primer, a partial ordering is an order on a set of elements which satisfies the following properties:

- **Reflexivity**: $a \leq a$ i.e an element is always ordered with itself
- **Antisymentry**: If $a \leq b$ and $b \leq a$, then $a = b$, i.e. no two distinct elemts precede each other.
- **Transitivity**: If $a \leq b$ and $b \leq c$, then $a \leq c$, i.e. if a precedes b and b precedes c, then a precedes c.

A total ordering is a partial ordering which satisfies the following additional property that each pair of elements is comparable, i.e. for any two elements a and b, either $a \leq b$ or $b \leq a$.

This seemed at the time to be promising for the distributed systems community, as it lead to seeming guarantees of consensus over a strict order in distributed systems.

However, a paper that came out in 1985 by Fischer, Lynch and Paterson, titled "Impossibility of Distributed Consensus with One Faulty Process" showed the following claim is not always true:

> **1985 Fischer, Lynch, Paterson.** Can we always come up with a way of getting computers to agree on a common value in a distributed system, even if some of the computers are faulty?

This paper is often referred to as the **FLP Impossibility Theorem**. Let's understand this further.

3.4.1 The FLP Impossibility Claim

We take the FLP paper's abstract verbatim that reads the following:

> **FLP Impossibility Theorem abstract.** The consensus problem involves an **asynchronous system** of **processes**, some of which may be **unreliable**. The problem is for the **reliable processes to agree** on a binary value. In this paper, it is shown that every protocol for this problem has **the possibility of nontermination**, even with only **one faulty process**.

The key takeaway from this abstract, without key terms being defined is: If we exist in a distributed system where each "process" / node is willing to participate in a consensus protocol, which may take infinite amount of time to reach a consensus, with message **delays being unbounded** and **no shared clock**, with at-most one process being faulty, i.e. dies without notice to others, then it is not always possible to reach to a common global value.

> **Async model guarantees eventual message delivery** Do note that when we say the model of message delivery between processes is **asynchrous** , we mean that there is no upper bound on the time it takes for a message to be delivered. However, the subtle point is that the message **eventually gets delivered and**

is not lost.

Before we establish this however, we need to define some key concepts:

3.4.2 Key concepts

Let's begin with a setting which is not distributed in nature. Consider a client process `CL` and a server node S. The server S saves a single bit value with itself and presents two distinct operations to the client `CL`:

- `write(v: 0/1)` : This operation writes the value of a variable given by client at the server S.
- `read() → 0/1` : This operation returns the value of a variable stored at the server S.

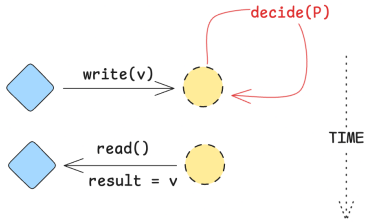

Figure 3.2: The single node case of read and write

A crucial component in this system is existence of a function `decide()`, which can be anything that operates on the internal state of the server. A simplest form of `decide()` could be a function that returns the latest value stored in the server. Another variant could be a function that returns the XOR value of the last 5 writes.

In our case, we can extend the same construction but to a distributed system outwardly presenting itself as a unit but internally being composed of multiple servers called **node**s. In essence, we desire that the same `write()`, `read()` and `decide()` functions that were present in the single node case, be present in the distributed system as well and **they should act exactly the same**.

The FLP paper imagines the distributed setting to be describable as follows:

There are n nodes in the distributed system. The nodes are connected to each other in connected but not necessarily a complete way in a graph-theoretic sense (a.k.a between any two nodes a path exist but not necessarily of length 1). Each process p_i is defined by $p_i = (I_i, O_i, f_i)$ where I_i is input in the set $\{0, 1\}$, O_i is a one-time write only output in the set $\{\phi, 0, 1\}$, ϕ being the value it exercise if it has not yet been written to, and f_i is a deterministic function that defines the internal state transition algorithm of p_i.

Each process p_i can send a message $m \in M$ (M being the "message universe") to some other process p_x at any time as a tuple (p_x, m) submitted to a globally singleton multi-set **message buffer** (note, source p_i is not described in the tuple).

The **message buffer** is a special non-deterministic (the only component in the system that is not deterministic) entity that presents each process with a function `receive(i)` $\rightarrow \{\phi, m\}$. This function when called with index i, either returns ϕ, a null message or m, one of the messages the buffer has received for delivery to process p_i. In the latter case, upon delivering the message, a tuple (p_i, m) is removed from the message buffer. Note how message buffer sufficiently describes an asynchronous message passing system. It can choose to returen the null message any number of times, in effect simulating unbounded delay.

In a **synchronous** setting however, we have some upper bound on the time it takes for a message to be delivered. In an **asynchronous** setting, we have no such bound. As such, it is impossible to detect if the counterparty process has failed or is just taking a long time to respond.

One should also discuss a bit about what it means to "fail" in a distributed system. Typically we categorize failures in 4 different categories:

- **Fail-stop**: The process halts, notifies everyone of its death and does not respond to any further messages.
- **Crash**: The process halts, does not notify anyone of its death and does not respond to any further messages.
- **Byzantine**: The process can send arbitrary messages to other processes, including lying about its own state.
- **Permissionless Byzantine**: The process can send arbitrary messages to other processes, including but not limited to impersonating a third-party.

In context of the FLP paper, we are concerned with the second type of failure, the "crash" without notification.

We define a **System Configuration** C as a unique combination of the internal states of all processes and the messages in the message buffer.

3.4.3 Consensus, System Configurations, Events and Schedules

Consensus in distributed systems is defined as any algorithm that satisfies the following properties:

- **Agreement**: All non-faulty processes decide on the same value.
- **Termination**: All non-faulty processes eventually decide on a value.
- **Validity**: All non-faulty processes decide on a value that was some process's internal state at the start. *Corollary*: If all non-faulty processes start with the same input value, then the decision value must be the same as the input value.

The system configuration C advances to a new configuration C' when it encounters **events**.

Events are messages (p, m) from the message buffer. Since there are many different messages that can be extracted from C due to message buffer being non-deterministic, many different configurations can stem out from single configuration C. A list of events extracted from C is called a **schedule**.

If at any system configuration C, all schedules lead to a final terminating consensus decision of value v, then we say that the system configuration C is **decisive** / **univalent** for value v. This can also be written as *v-valent*. Since we are in binary domain, decisive configurations are either 0-valent or 1-valent.

If at any system configuration C, different schedules lead to different consensus decisions, then we say that system configuration C is **indecisive** / **multivalent**. Since we in binary domain, it also is called *bi*-valent.

3.4.4 Lemma 1: Disjoint Schedules Commute

Suppose that from some configuration C, the schedules σ_1, σ_2 lead to configurations C_1, C_2 respectively. If the sets of processes taking steps in σ_1 and σ_2, respectively, are disjoint, then σ_2 can be applied to C_1 and σ_1 can be applied to C_2, and both lead to the same configuration C_3. See 3.3 for a visual representation.

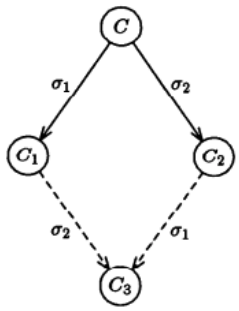

Figure 3.3: Schedules applying to disjoint processes can be out of order

3.4.5 Lemma 2: There exists a bi-valent initial configuration

Let us assume the contrary to be the case, i.e. all initial system configurations C are decisive, uni-valent. Then, either all are 0-valent, 1-valent or a mix of both with some states being 0-valent and others 1-valent.

A strictly 0-valent or 1-valent configuration is trivial in nature and does not require any consensus protocol to reach a decision. However, a mix of both 0-valent and 1-valent configurations would require a consensus protocol to reach a decision.

Assume n processes participating in the system. Then, all decisive initial configurations are 2^n in number. Since all of them are uni-valent. There would exist some adjecency boundary between 0-valent and 1-valent configurations in the n dimensional lattice, the only process differentiating this being one of the n processes in the system. If such process goes down without notice, the system would be left in a bi-valent state if it intends to resolve the consensus deterministically as it cannot choose 0-valency or 1-valency.

3.4.6 Lemma 3: All roads lead to indecision

Claim: If we start from some bi-valent system configuration C_i, with some specific event $e = (p, m)$ extractable from the message buffer from C_i, there exists a schedule S_{C_i} that leads to a bi-valent configuration C_{i+1} with event e consumed in the process.

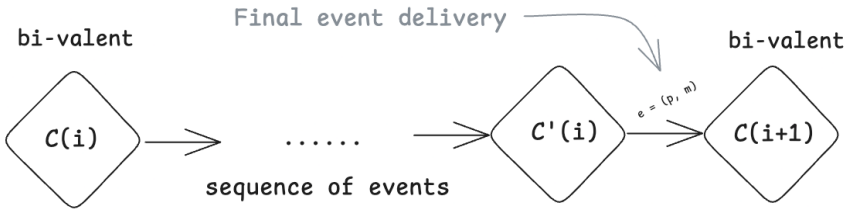

Figure 3.4: A bi-valent configuration can lead to another bi-valent configuration

It is noteworthy that we operate in a asynchronous network model, where-in while it can take a really long time for a message to be delivered, the guarantee exists that it is eventually delivered.

This lemma intends to present that there is a sequence of message deliveries that we can do at each bi-valent configuration that leads to another bi-valent configuration, and also that e getting delivered. Multiple applications of this approach will essentially prove the FLP impossibility theorem.

Let us set-up the proof in the following way:

- Start with a bi-valent configuration C_i.
- Let there be a sequence of events $S_{C_i} = \{e_1, e_2, \ldots, e_n\}, e_i \neq e \forall i \in [0, n]$ that lead to a bi-valent configuration C_{i+1}. It should be noted that $|S_{C_i}| = 0$ is possible.
- Let C_i' be the "penultimate configuration" that is reached by applying the schedule S_{C_i} to C_i. Since $|S_{C_i}| = 0$ is possible, C_i' can be the same as C_i.

All configurations in 3.4 are bi-valent: It is easy to see that in our definition of system configurations, we can never go from a uni-valent configuration to a bi-valent configuration. In this setup of C_i and C_{i+1} both being bi-valent, all the intermediate configurations including the penultimate configuration C_i' must also be bi-valent.

We now define a few possible configuration categories for intermediate and penultimate configurations shown in 3.4.

- **Pre-uni-valent**: Some configuration C^* is pre-uni-valent, if it is reachable from C_i via some sequence of events without delivering $e = (p, m)$ and application of e to it would transition into a new configuration C^{**} which will be uni-valent. For example, if C' is pre-uni-valent, then C_{i+1} is uni-valent which we do not want.
- **Pre-bi-valent**: Some configuration C^* is pre-bi-valent, if it is reachable from C_i via some sequence of events without delivering $e = (p, m)$ and application of e to it would transition into a new configuration C^{**} which will be bi-valent. For example, if C' is pre-bi-valent, then C_{i+1} is uni-valent which we want.

Note that Pre-X-valency resolving into X-valency for X being uni or bi, is a property of the event $e = (p, m)$.

Note that all Pre-uni-valent configurations necessarily need to be bi-valent. As by definition, at that configuration if a decision is already made, the configuration would reduce to uni-valency. Hence, a Pre-0-valent configuration will have both 0-valent and atleast one of 1-valent and bi-valent configurations as its reachable children configurations. Similar argument goes for Pre-1-valent (atleast one of 0-valent or bi-valent) and Pre-bi-valent (atleast one of 0-valent or 1-valent).

So, what can we say about C_i? Well, if C_i is Pre-bi-valent, then by definition application of e to it will lead to a bi-valent configuration C_{i+1}. In this case, we have proven bi-valency leading to bi-valency.

The other case however is more interesting. What if C_i is Pre-uni-valent? Well, direct application of e to it will lead to a uni-valent configuration which we want to avoid. We need to path our way from a Pre-uni-valent configuration C_i to some Pre-bi-valent C_i' so that we can transition into a bi-valent C_{i+1} by application of e. **Can we always path a way from a Pre-uni-valent configuration to a Pre-bi-valent configuration?**

We begin with an assumption that C_i is Pre-0-valent. Assume another event $e' = (p', m')$ distinct from $e = (p, m)$ such that application of e' to some reachable configuration X from C_i leads to a non Pre-0-valent configuration Y. See figure 3.5 for a visual representation.

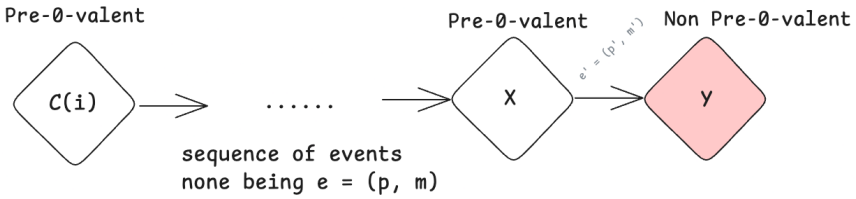

Figure 3.5: Finding a non Pre-0-valent configuration from a Pre-0-valent configuration

We claim to **show that Y is Pre-bi-valent**.

This is because, if Y is Pre-1-valent and e is applied to it, the resulting configuration Y' will be 1-valent. However, since e can be applied to X as well, the resulting X' from X will be 0-valent. If later on, e' is applied to X', the resulting configuration X'' will also be 0-valent.

However, we note that X'' is identical to Y' by lemma 1 if e and e' are disjoint, i.e. $p' \neq p$. This is a contradiction as Y' is 1-valent and X'' is 0-valent.

On the other hand if $p' = p$, then resolution of deterministic value of choice rests on a single process p. If this process crashes, the system will be left in a bi-valent state.

Hence, under the assumption of possibility of single node failure, transition from Pre-0-valency to Pre-1-valency is impossible. This implies that Y indeed is a Pre-bi-valent configuration.

3.4.7 Proving the theorem

Let us start our protocol from some initial bi-valent configuration C_0. Existence of such initial configuration is given by lemma 2.

For any transition to C_{i+1} from C_i, we choose the oldest message e_{old} known to be in the message buffer at C_i. Given C_i and e_{old}, we can construct a schedule S_{C_i} that leads to C_{i+1} as per lemma 3 leading to a bi-valent configuration. This also, leads to the consumption of the oldest message e_{old} known at C_i.

We continously apply the above scheme till the point we are left with no messages in the message buffer. We eventually still reach a bivalent state C_∞.

3.4.8 Conclusion

- The FLP impossibility theorem makes clear that **trade-offs are required** in the design of distributed systems. Whether it is loosening up on deterministic guarantees or on tightening up on timing guarantees, you cannot have both.
- The above can also be described as, in event of asynchrony, do you compromise on consistency (determinism) or on liveness? **Longest chain (Nakamoto) based consensus protocols like Bitcoin and Ethereum compromise on consistency, while protocols like PBFT compromise on liveness**.

Further Resources

Original Paper : https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf

Overview - FLP Impossibility Theorem : https://www.youtube.com/watch?v=Vmlj-67aymw

Extending the bi-valent configuration (Lemma 3) : https://www.youtube.com/watch?v=cwEXmcszbos

Consize summary : https://www.youtube.com/watch?v=2fnS-f-mwIo

3.5 The CAP principle

Chapter 4

Ethereum

4.1 Legacy Ethereum (Ethereum 1.0)

4.1.1 Accessing Historical Block Hashes: EIP: 2935

For various applications, it is necessary to access the historical block hashes of the Ethereum blockchain. One such reasons is that it provides a good source of randomness. Another good reason is to build an "introspection engine" where contracts on Ethereum can introspect its own history.

Block hashes of the current block can not be known during the context of execution of a transaction. One of the reasons for this is that during execution of a transaction, the block's receipt hash is not built yet, since that in turn depends on all the transactions in the block combined, each transaction cannot have the block-hash.

Ethereum however, has an opcode called `BLOCKHASH` which allows contracts to access the block hash of the 256 most recent blocks. This opcode is useful for applications that require randomness, but it is not enough for applications that require access to the entire history of block hashes.

But what limits the access to historical block hashes more than 256 blocks old? There hash been two separate known attempts at solving this. One by Vitalik himself via EIP-210 and another by Vitalik, Tomasz (Netermind) etc via EIP-2935. As of writing, EIP-210 is marked as stagnant and EIP-2935 is in "Review" state since 4 years.

The primary idea of EIP-2935 was to create a new storage trie that stores the block hashes of all blocks. This would have been located at a special location: `0xffffffff_ffffffff_ffffffff_ffffffff_fffffffe`. This address would essentially be just a state without a bytecode, making it different from all other accounts. Hence, a new opcode `STOP` could

4.2 Ethereum after Merge (Ethereum 2.0)

4.2.1 The Proof-of-Stake Block

In every slot (spaced twelve seconds apart) a validator is randomly selected to be the block proposer. They bundle transactions together, execute them and determine a new 'state'. They wrap this information into a **block** and pass it around to other validators.

Other validators re-execute and validate: Validators who hear about the new block re-execute the transactions to ensure they agree with the proposed change to the global state. Assuming the block is valid, they add it to their own database.

If a validator hears about two conflicting blocks for the same slot they use their fork-choice algorithm to pick the one supported by the most staked ETH.

Previously, the Ethereum's Proof-of-Work Block (PoW Block) only contained execution related information. Post-Merge, the new Proof-of-Stake Block (PoS Block) will contain ACE information:

- **A**dministration (the metadata of the block)
- **C**onsensus (Beacon Chain co-ordination)
- **E**xecution (Block data)

This makes PoW Block more or less a subset of the new PoS Block. See 4.1.

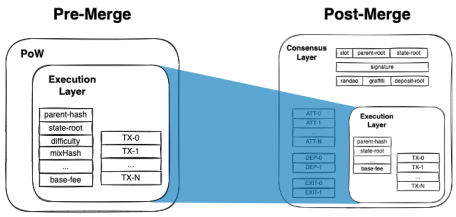

Figure 4.1: The Proof-of-Stake Block

4.2.2 Finality Metalayer: Casper FFG

Talk about:

- Plausible Liveness
- Accountable Safety

Chapter 5

Solana

Solana seems to be an interesting trade-off choice in the blockchain trilemma when compared to the likes of Ethereum. The blockchain trilemma is a trade-off between decentralization, security and scalability. Solana seems to be focusing more on the scalability, while security and decentralization are arguably worse as compared to Ethereum.

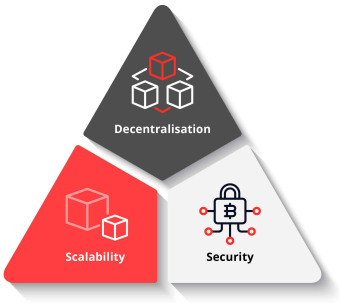

Figure 5.1: The blockchain trilemma

Very briefly, the blockchain trilemma goes as follows:

- **Decentralization**: The more decentralized a blockchain is, the more secure it is. This is because it is harder for a single entity to take over the network. This is often measured using the *nakamoto coefficient*[1].
- **Security**: The more secure a blockchain is, the more guarantees it can provide, the harder it becomes for an adversary to compromise the system. Guarantees come in the form of economically-backed guarantees (which are subject to elements such as total tokens staked in a network etc) or mathematically-backed guarantees (such as zk proofs, hash non-collision etc.).
- **Scalability**: The more scalable a blockchain is, the more user requests, often measured in form of "transactions per second" or TPS it can perform.

Solana focuses a whole lot more on the "transactions per second" part of the blockchain, a.k.a. the "scalability" aspect.

5.1 The Transaction Processing Unit

5.2 Proof of History: VDF based transaction ordering

5.3 Entries, Slots, Bank

[1]**Nakamoto Coefficient**: First proposed by Balaji Srinavasan and Leland Lee and named in honor of Satoshi Nakamoto, the pseudonymous creator of Bitcoin, the Nakamoto Coefficient is a measure of the smallest number of independent entities that can act collectively to shut down a blockchain. It is often closely related to the "Gini Coefficient" of the economical system under question. For current nakamoto number of public blockchains using Proof-of-Stake, see https://nakaflow.io/

Chapter 6

General Concepts in Adversarial Distributed Systems

6.1 Proof of Stake: The Nothing at Stake problem

At a first glance, Proof of Stake seems to be a great idea. It is more energy efficient than Proof of Work, and it is more democratic in the sense that the more tokens you have, the more say you have in the network.

However, it is not quite comparable to Proof of Work. To understand why, we need to notice the following two properties of PoW:

- **Finite prowess**: In proof of work systems, the prowess (often calculated as the number of puzzles a miner can try) of a miner is limited by the amount of computational power they have. Whether they use COTS, ASICs, FPGAs, all are limited inherently. Whenever a miner sees a fork in the chain, they have a choice to make: which chain to mine on. Should they dedicate their prowess to the `fork #1` or should they dedicate their prowess to `fork #2`? Or should they mine on both equally? Regardless of their choice of prowess distribution, their total prowess is limited.
- **Wrong choice is costly**: If a miner chooses to mine on the wrong fork, they lose out on the rewards they could have gotten had they mined on the right fork. This is because the right fork will have more blocks, and hence other miners due to longest chain fork-choice rule will focus their effort onto it, making it "the canonical chain" in due time, leaving the wrong fork blocks to be wasted effort / orphaned.

The above two properties in Proof-of-Work chains, coupled with the assumption that each miner will make their own choice of fork-selection independently, make them inherently a good canonicalization machine, i.e. given enough time all forks will converge to a single chain.

However, in Proof of Stake, the above two properties are not true. The prowess of a validator is not limited by computational power, but by the amount of tokens they have staked. Any validator can freely attest on both the forks. Secondly, since they are not making a choice here, there is no concept of "wrong choice" being costly. This leads to a problem called "Nothing at Stake".

6.1.1 No penalties = No canonicalization

In many early (all chain-based) proof of stake algorithms, including Peercoin, there are only rewards for producing blocks, and **no penalties**. This has the unfortunate consequence that, in the case that there are multiple competing forks, it is in a validator's incentive to try to make blocks on top of every fork at once, just to be sure.

The result is that if all actors are narrowly economically rational, then even if there are no attackers, a blockchain may never reach consensus on a "canonical state", and all validators will keep building on both forks. If there is

an attacker, then the attacker need to only overpower the altruistic nodes (who would exclusively stake on the original chain), and not rational nodes (who would stake on both the original chain and the attacker's chain), in contrast to proof of work, where the attacker must overpower both altruists and rational nodes.

6.1.2 Example: The 1% attacker

TODO: Show how a person controlling >0% stake can canonicalize the wrong fork and thus you cannot trust any fork until its made canonical

6.1.3 The Tragedy of the Commons

The Tragedy of the Commons is a concept which states that if many people enjoy unfettered access to a finite, valuable resource, such as a pasture, they will tend to overuse it and may end up destroying its value altogether. Even if some users exercised voluntary restraint, the other users would merely replace them, the predictable result being a "tragedy" for all.

In the context of Proof of Stake, each individual stakeholder might only have a 1% chance of being "pivotal" (i.e. being in a situation where if they participate in an attack then it succeeds and if they do not participate it fails), and so the bribe needed to convince them personally to join an attack would be only 1% of the size of their deposit; hence, the required combined bribe would be only 0.5-1% of the total sum of all deposits. Additionally, this argument implies that any zero-chance-of-failure situation is not a stable equilibrium, as if the chance of failure is zero then everyone has a 0% chance of being pivotal.

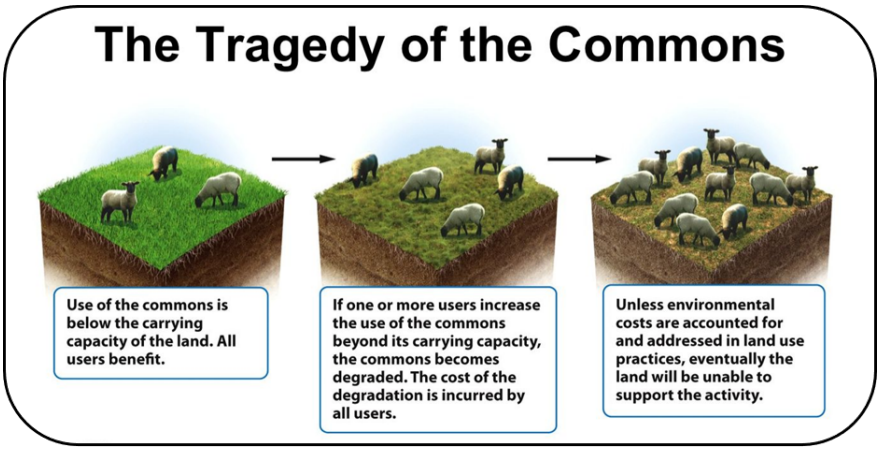

Figure 6.1: Tragedy of the commons

6.1.4 Simulating the "wrong choice is costly"

There are two ways in which one can try solving the "Nothing at Stake" problem, both of which involve making the "wrong choice" costly:

- **Slasher**: In the Slasher algorithm, if a validator is caught signing two conflicting blocks on two different forks, they lose their part / entire stake. TODO: requires elaboration, see why they call it too harsh
- **Casper FFG**: Every validator is forced to choose a fork to validate on. If there are two competing chains, A and B, then if a validator creates a block on B, they get a reward of +R on B, but the block header can be included into A (in Casper this is called a "dunkle" or *proof of malfeasance*) and on A the validator suffers a penalty of -F (possibly F = R). TODO: show in Casper design how this is done

6.2 Wealth Inequality and the Gini Coefficient

Many popular blockchains follow a Proof-of-Stake model of consensus. In PoS, the more tokens you have staked, the more faith you present to the network. In return, the network rewards you with more tokens for protocol validation services you provide to the network.

This mainly means that a certain actor with more tokens can stake more tokens, and assuming does follow protocol right, earn more tokens as a result. This is a positive feedback loop, and is often referred to as the "rich get richer" problem. This is a problem because it leads to wealth inequality in the network.

TODO: talk about lorenz curve and gini coefficient

6.3 Miner's Extractable Value (MEV)

Miner's Extractable Value (MEV) is a term coined by Phil Daian et al. in their Flash Boys 2.0 paper. It refers to the value that miners can extract from the transactions they include in a block. MEV is a result of the fact that miners have the ability to order transactions in a block and can choose which transactions to include or exclude. This gives them the power to front-run transactions, re-order transactions, and censor transactions.

Sometimes MEV is also described as "maximal extractable value".

General purpose blockchains provide an almost turing-complete environment[1] for executing smart contracts / programs. Smart contract / programs are sequence of steps or rules that can be deployed at will. Some of these major programs are Decentralized Exchanges (DEX), Automated Market Makers (AMM), and lending protocols. These programs are often used to trade assets, provide liquidity, and borrow assets.

6.3.1 Basic on-chain DeX based arbitrage

If an arbitrage opportunity exists of the following format:

- **Transaction 1**: Decentralized exchange $D1$ is selling asset A in exchange of B at rate $x : 1$
- **Transaction 2**: Decentralized exchange $D2$ is buying asset A and giving B at rate $y : 1$
- **Arb Condition**: $x > y$, i.e. $D1$ gives more A per unit of B than $D2$ needs to exchange for a unit of B

6.4 The ever growing state, state expiry

State expiry refers to removing state from individual nodes if it hasn't been accessed recently. There are several ways this could be implemented, including:

Expire by rent: charging "rent" to accounts and expiring them when their rent reaches zero. This design is implemented in Solana.

Expire by time: making accounts inactive if there are no reading/writing to that account for some amount of time

6.5 Rebuilding the chain / Chain Derivation

6.6 Supporting a different virtual machine

Execution of code in any blockchain assumes an "execution environment". This execution environment is often called the "virtual machine" that blockchain is running. For example, Ethereum runs the EVM, the **E**thereum **V**irtual **M**achine. Solana runs the SVM similarly.

A curious question arises: **Can a virtual machine non-native to a blockchain be introduced as a co-execution environment?**. The answer to this is a resounding yes. However, one needs to question the intent of doing this. Almost all execution environments are turing complete, however the cost of executions may be prohibitive at times.

[1] Almost in the sense that execution is still bounded to some maximum number of steps, called gas limit in Ethereum and compute limit in Solana

One such use-case is extending the EVM with capabilities to run WASM or RISC-V code. A couple interesting projects doing this would be Arbitrum Stylus and R55.

Arbitrum Stylus

Stylus extends the execution environment to run WASM-targeted smart contracts. Instead of writing in solidity, one writes the code in languages like Rust, Zig, Go, Sway, Swift. Rust is supported as primary language and stylus-sdk is used to build such programs.

R55-Eth

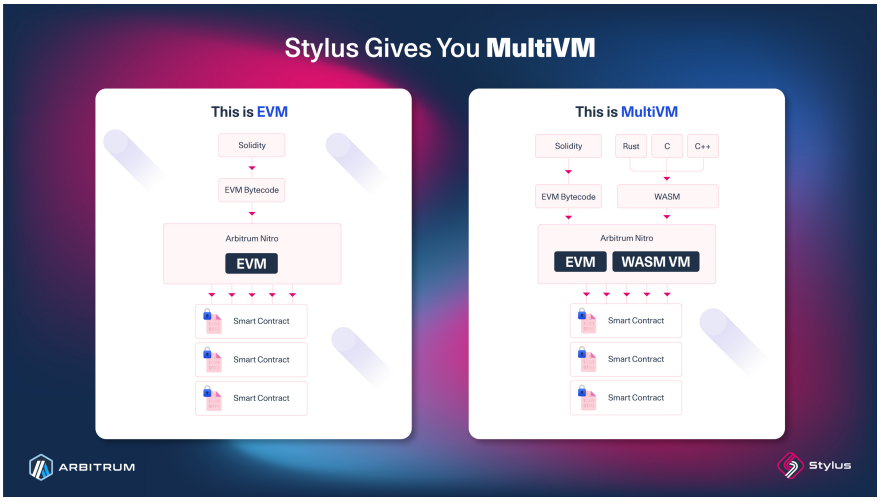

Figure 6.2: Arbitrum Nitro

Chapter 7

Layer 2 Scaling Solutions

7.1 General classification

We classify Layer 2 scaling solution according to their setup:

- **Plasma**: A tree of sidechains (chains of chains) where each user is asked to ensure the part of state it owns is available. Here, significant interaction with L1 is only required in case of dispute resolution. This came much before other solutions (proposed by Joseph Poon and Vitalik Buterin in 2017) but it was too complex to build a useful plasma.
- **Rollup**: An ecosystem running in parallel to a Layer 1 solution that has it's own state transition function (STF). After a bunch of transactions and blocks are created, the rollup posts transaction data / state-diff on chain and proves the combined state transition via opmistic or succinct zk proof.
- **Validium**: Similar to a rollup, just that the data-availability of the transaction is kept not on L1, but a separate *data-availability committee* or the validium operator themselves. Validium saves on recurring cost to make data available on L1, but at the cost of security.

7.2 Where sits the enourmous data?

Whenever a layer 2 scaling solution is built, the layer 1 contract is provided with the root commitment of the updated state. But a commitment is useless unless it can be opened. A *witness* links a leaf to the commitment and it requires a view of the state / transaction data for things like disputes and withdrawals.

Where does this data exists for generating a witness to do anything useful? A lot of answers can come up:

- Let the data sit on the Layer 1 itself, the rollup route. This is the simplest of the solutions but seldom the most practical. Layer 1s are expensive to post data to.
- Let the data sit on some other special data-availability layer which has some presence on the Layer 1 concerned. For example, posting on celestia. This is the validium route. Here you still pay to post on Celestia albeit much less.
- Let the data sit on a data-availability layer even cheaper than celestia, maybe with a smaller quorum. This is an even cheaper validium route. But the trade-off is security, as the data-availability layer may not be as secure as for example celestia.
- Let the data sit with the users themselves. Let they maintain the data and post it when required. This is the plasma route. This is the most decentralized of the solutions but also the most complex to build. Also, it requires the users to be online all the time. A pretty bad UX.

7.3 Plasma: Each one save one

The setup of Plasma entails that if each part of the state has a well-defined owner, then the owner can be made responsible for keeping their part of the system available.

Essentially, all Plasma chains have an **Operator** who keeps the chain moving. The system is setup optimistically with an intinsic level of trust on this operator. If the operator misbehaves, it remains the responsibility of the users to exit the chain.

Plasma chains are multi-layered chains where every "Parent Chain" settles for its child chains. The topmost "Parent Chain", the root chain settles on a well-known settlement platform and calls it the Layer 1. Each parent chain is responsible for three basic events through it's contract:

- **New Deposit**: Whenever anyone submits a new token deposit of the form `(depositAddress, amount, blockHeight)`.
- **Block Progression**: Only submit the root-hash of the block, *not the data contained within it*. The data within hash to be kept in a good condition (and available) by the user. This takes the form `(blockHeight, rootHash)`.
- **New Withdrawal**: If the operator misbehaves, the users are supposed to exit the plasma chain. This request is posted directly on Layer 1, as it may be assumed that the operator is not available / misbehaving. This takes the form `(withdrawaladdress, amount, blockheight, txIndex, ouputIndex)`.

7.3.1 Operator misbehaviors and exits

7.3.2 Plasma Cash

In **Plasma Cash**, since users are only keeping track of their own tokens, they don't know who owns any of the other tokens. When a user wants to send their token to another user, they need to prove that they actually own that token! This creates a problem where each user is expected to be *perennially online* to ensure they can generate a witness against the current state-root of plasma chain.

In such a setup however, other inefficiencies also arise, because of the non-fungible nature of the tokens involved. In a ledger model of accounting, spending 0.1 Token or spending 20 Token out of a 50 Token balance are the same complexity-wise. However, the same is much harder with NFTs given we have to establish the denominations beforehand for all possible transfers in the future, or deal with splitting and merging of tokens.

7.4 Bottlenecked by DA

Layer 2 rollups have been around for a while. The first Layer 2 scaling solution was Plasma, proposed by Joseph Poon and Vitalik Buterin in 2017. The idea was to create a tree of sidechains, each of which would be able to process transactions in parallel. The root chain would be responsible for the security of the system, while the sidechains would be responsible for processing transactions.

Rollups however needs to guarantee that the state of rollup is **data-available**. Otherwise, if state is not available and the rollup-operator stops working, users cannot withdraw. As, even though the root commitment to the entire state is available, the opening are not. This makes rollups constrained by data-availability capacity of the Layer 1 chain.

Ethereum provides 375 `KB` blobs every 12 seconds.

The following question hence often comes us: *Can we eliminate need for state availability in L2 scalability solutions?*

7.5 Rollups and levels of confirmation

7.6 Faster Layer 2 Interoperability

In late 2020, Ethereum community started focusing on *rollup-centric roadmap*. The idea of this was to enable various Layer 2 scaling solutions that may potentially provide a whole lot more of execution capacity than the Ethereum (henceforth to be referred as Layer1 or L1) will be capable of.

However, Layer 2 come with following new concerns:

- **Security**: The security of Layer 2's is the concern of how easy it is to compromise a L2 in comparison to an L1. Generally, in a proof-of-stake system like ethereum, a lot of security is borrowed from stake at play. As of writing, this is around 33.8 million ETH staked / around 136.8 billion USD securing the ecosystem. Can the L2 match this level of security?
- **Decentralization**: Control of the system especially with chains like Ethereum is fairly decentralized. As of writing, around 1 million validators run the nodes for the main chain[1]. Layer 2 solutions hardly have decentralized sequencers and hence sit almost on the far end of decentralization argument. Ca the L2 match this level of decentralized controls?
- **Fragmentation**: Every L2 is almost like a separate "room", although situated within the same "house", the L1. This eventually leads to fragmentation of both users, and the liquidity. On a DEX, liquidity is directly tied to the quantity of tokens users have committed to the liquidity pools. If a crypto asset lacks sufficient liquidity, token holders may face difficulty selling their tokens when they wish. L2 brings in another dimenstion to this problem. Not only are tokens spread thin among various DeFi apps on the same chain, they are now spread among various DeFi apps on *various* chains.

The Interoperability problems is concerned with the *fragmentation* issue brought in by the advent of L2s.

In its most basic essence, Interoperability is concerned with a transaction that intends to affect state of two different L2 rollup chains. For example, a token transfer from A's account on rollup X to B's account on rollup Y is a transaction that affects the state of both X and Y.

[1] This number is a bit misleading given a lot is contributed by single institutions like stake.fish running multiple validator nodes given each validator can only stake 32 ETH per validator. The Ethereum Pectra Upgrade (EIP-7251) is supposed to fix it with increased `MAX_EFFECTIVE_BALANCE` the staking balance per validator, in hopes consolidating many such large validators into fewer entities

Chapter 8

Golang

Golang things

Chapter 9

Rust

RUSTY things

9.1 Iterators

IMPLEMENT RLE ITERATOR here: https://leetcode.com/problems/rle-iterator/description/

Chapter 10

Solidity

10.1 The memory model

10.1.1 Memory types: `calldata` vs `memory` vs `storage`

Working with solidity requires us to think deeply about the memory model. Three data locations exist [1]:

- `calldata`: This is a read-only area where function arguments are stored. It is immutable and cannot be modified.
- `memory`: This is a temporary area where data can be stored and modified and available during the execution of the call. It is erased between function calls and does not persist on the blockchain.
- `storage`: This is a persistent storage area on the blockchain.

10.1.2 The on-chain storage

Each contract can only store data in its own storage [2]. Ethereum gives each contract 2^{256} slots of storage, where each storage is 32 bytes. Slots are contiguous and are referenced by indexes, starting at 0 and ending at 2^{256}. All slots are initialized to a value of 0. Each contract's storage area has more slots than all the stars in the universe.

Static sized object representation

Every time we declare a struct, it follows the same rules of representation as C / C++ / Rust. For example, the following contract stores the struct as follows:

Listing 10.1: In-storage representation

```
contract UnpackedRepresentation {
    bool boolVal;   // 1 bit of data, represented in slot 0
    uint256 largeVal; // 32 bytes of data, represented in slot 1
    uint8 smallVal; // 1 byte of data, represented in slot 2
}
```

A better alternative of this could have been:

[1] Alchemy doc explains with an example: https://docs.alchemy.com/docs/what-is-the-difference-between-memory-and-calldata-in-solidity

[2] Alchemy doc on storage: https://docs.alchemy.com/docs/smart-contract-storage-layout

Listing 10.2: In-storage representation(better)

```
contract PackedRepresentation {
    bool boolVal;   // 1 bit of data, represented in slot 0
    uint8 smallVal; // 1 byte of data, represented in slot 0 (alongside boolVal)
    uint256 largeVal; // 32 bytes of data, represented in slot 1
}
```

Dynamically sized object representation

Two primary dynamically sized objects exist in solidity: Dynamically sized arrays and Mappings. For each, the following is done:

1. Similar to static sized objects, a slot is chosen for the object, say slot 3, this is the "marker slot" and holds only the length of the array or the mapping.
2. The first element of the array is stored at `keccak256(3)+0`. The second element is stored at `keccak256(3)+1` and so on.
3. In case of mappings, the mapping of some key k, in a mapping whose marker slot is m, is stored at `keccak256(h(k), m)`. The function h is dependent on the type of the key. For example, if the key is a string, then h is the key value unpadded. If the key is an integer, then h is the identity function padded to nearest 32 bytes.

10.2 Function and their modifiers

10.2.1 Visibility Modifiers: `private`, `internal`, `external`, `public`

Solidity has four visibility modifiers for functions[3]. In increasing order of access they are (mnemonic **PIEP**):

- `private`: Accessible by functions in the **same contract**, not even derived contracts.
- `internal`: Accessible by functions in the same contract **and derived contracts**.
- `external`: Accessible by functions from **other contracts**, but not internally (no call from the same contract).
- `public`: Accessible by functions everywhere.

The general rule of thumb should be applied. Give each function the least amount of accessibility as possible.

10.2.2 Function Modifiers for preconditions and postconditions

Solidity provides us with a way to modify the behavior of functions using "function modifiers". Modifiers are often used to add preconditions or postconditions to functions, or to restrict access to certain functions.

The following is an example of a function modifier:

Listing 10.3: Solidity isOwner modifier

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

[3]This youtube video has a good explanation: https://www.youtube.com/watch?v=KTEp9xfu2tc

Here, the `onlyOwner` modifier checks if the sender of the message is the owner of the contract. The `_;` statement is a mandatory placeholder for the function body that uses this modifier. A good explaination of this can be found here from freecodecamp.

Listing 10.4: Contract with modifiers

```
contract BaseContract{
    address owner;

    modifier onlyAllowedUser(address user) {
        require(owner == user);
        _;
    }
}

contract DerivedContract is BaseContract {
    function updateData(
        bytes32 newData,
        address user
    ) onlyAllowedUser(user) public {
        // function body
    }
}
```

10.2.3 Special modifiers: `view`, `pure` and `payable`

In Solidity, functions can have special modifiers like pure, view, and payable that indicate their behavior, especially regarding how they interact with the blockchain's state or handle Ether.

- `view`: This modifier indicates that the function does not modify the state of the contract. It can read the state but cannot change it. It is used for functions that only return values based on the current state of the contract.
- `pure`: This modifier indicates that the function does not read or modify the state of the contract. It can only use its input parameters and return values. It is used for functions that perform calculations or operations without relying on the contract's state.
- `payable`: This modifier indicates that the function can accept Ether. It allows the function to receive and process Ether sent along with the transaction. It is used for functions that involve financial transactions or require payment.

Exercise 1 Solidity Basics

1. Explain the difference between `calldata`, `memory` and `storage` in Solidity.
2. Assume we deploy the contract at 10.2.2 on the Ethereum blockchain, without any parameters while deploying. What will be the value of `owner` in the contract (BaseContract)?
3. Write a solidity contract that gives you the N-th Fibonacci number.
4. Differentiate between `pure`, `view` and `payable` functions in Solidity.

10.3 The EIP-721 Standard: Non-Fungible Tokens (NFTs)

Modifier can take in arguments as well and can be borrowed from other contracts. Take for example this:

10.4 Exercise Solutions

Answer of exercise 1

1. Refer to 10.1.1
2. The value would be the zero address. `0x0000000000_0000000000_0000000000_0000000000`
3. Listing 10.5: Fibonacci Contract

```
contract FibonacciContract {
    uint256 f0 = 0;
    uint256 f1 = 1;

    function getNthFibonacci(
        uint256 n
    ) public
    view
    returns (uint256) {
        if (n == 0) {
            return f0;
        } else if (n == 1) {
            return f1;
        }

        uint256 a = f0;
        uint256 b = f1;

        for(uint256 i = 1; i < n; i++) {
            uint256 added = a + b;
            a = b;
            b = added;
        }

        return b;
    }
}
```

4. Refer 10.2.3 for the definitions.

Chapter 11

Binary Search

Binary search is a simple technique of problem solving that splits the solution space in half at each step. But when it comes to implementation, it's rather difficult to write a bug-free code in just a few minutes.

A lot of issues arise when implementing binary search such as:

- Off-by-one errors
- Incorrect bounds
- Incorrect mid calculation
- Incorrect condition to update bounds

11.1 Basic: Minimize k such that $f(k)$ is true

This is the most basic form. Basically becuase the search space is given directly. We would be given a vector $[A0, A1, ..., An]$ and we need to find the minimum k such that $f(k)$ is true.

11.2 Advanced: Search spaces with `feasible(x) -> bool`

Finding the search space could be tricky sometimes[1]. However, one common element in these problems is the aspect of monotonicity. Here, if $f(k)$ is true, then $f(k+1)$ is also true. Following are some good examples that establish the concept.

11.2.1 Problem [SeqShip]: Sequential objects shipping within D days

Problem: Assume a sequential line of objects O with their individual weights described in an array that are situated at port A and want to be shipped to port B. Every day, a ship of some maximum weight capacity m makes trip from A to B transporting some objects in O. Given a maximum days D that shipping company can tolerate to transport all objects in O, find lowest possible m. An example object line could look as follows:

$$O = [6, 7, 1, 2, 3, 4, 5, 8, 9, 10]$$

Given $D = 5$, the minimum m would be 15. Because we can ship items as: $[6, 7], [1, 2, 3, 4, 5], [8], [9], [10]$.

Discussion: This problem does not have a clear search space prima facie, but the thing we realize is it asks us to minimize m. Now let's run this idea: if some m works for shipping in D days, would $m+1$ work also? More

[1] A good description is available on Leetcode: https://leetcode.com/discuss/study-guide/786126/Python-Powerful-Ultimate-Binary-Search-Template.-Solved-many-problems

concretely, assume that a ship of capacity 10 is able to do the work at hand, can a ship of capacity 11 do it too? The answer is yes. We can call this function $f(x)$ more aptly `feasible(x) -> bool`.

But what should be the search space? Well, the minimum we need is atleast $max(O)$ since we need to ship the heaviest object in some trip, even if it is the only item shipped that day. The maximum we need is probably shipping all objects in one day. So, the search space is $[max(O), sum(O)]$.

Solution: The problem can now be reduced to: Find m such that $f(m)$ is true where $f(m)$ is defined as: *Can we simulate shipping O in D days with a ship of capacity m?* in the search space $[max(O), sum(O)]$.

Each simulation is an $O(n)$ operation where $n = |O|$. Maximum simulations is $log_2(sum(O) - max(O))$. Hence runtime is $O(|O| \cdot log_2(sum(O) - max(O)))$.

11.2.2 Problem [KokoBanana]: Monkey eating piles of bananas

Problem: Assume a monkey is hungry and wants to eat some bananas. There are G piles of bananas, each pile i has b_i bananas. The monkey can eat k bananas per hour and has a maximum of D hours to eat all bananas. Each hour, the monkey can decide any one pile and eat k bananas from it, if there are less than k bananas, it eats all but doesn't eat from any other pile that hour. What is the minimum k for which the monkey can eat all bananas in D hours?

Example scenarios:

$$G = 4, \texttt{Piles} = [3, 6, 7, 11], D = 8 \quad k = 4$$

$$G = 5, \texttt{Piles} = [30, 11, 23, 4, 20], D = 5 \quad k = 30$$

$$G = 5, \texttt{Piles} = [30, 11, 23, 4, 20], D = 6 \quad k = 23$$

Discussion: Very similar to problem [SeqShip], this problem also has a monotonicity property. If k bananas per hour works, then $k + 1$ bananas per hour would also work, albeit not the most optimal. This also uses `feasible(x) -> bool` function.

But what is the search space? The minimum k would be 1 as that's the least a monkey has to eat per hour to make progress hour-on-hour. However, we can probably do better than that. The minimum should be no lesser than $m = D/sum(\texttt{Piles})$ as that's the least number of bananas the monkey has to eat per hour to eat all bananas in D hours assuming no inefficiencies, i.e. every hour is spent eating exactly m bananas. This is unlikely since not all b_i may be exactly divisible by m.

The maximum k would be $max(\texttt{Piles})$ as that's the most bananas the monkey can eat in one hour given that in each hour it can only eat from a single pile.

The search space is hence $[D/sum(\texttt{Piles}), max(\texttt{Piles})]$.

Also note an interesting property: a monkey cannot eat any faster than in G hours. Hence any problem where $|Piles| > G$ would have no solution.

11.2.3 Problem [SplitM]: Split array into m subarrays

Problem: Given an array A of non-negative integers, split it into m subarrays such that the maximum sum s of any subarray is minimized. Example:

$$A = [7, 2, 5, 10, 8], M = 2 \quad s = 18(\texttt{split}: [7, 2, 5], [10, 8])$$

Discussion: The monotonicity property still holds. If s is a valid maximum sum of any subarray, then $s + 1$ is also a valid maximum sum of any subarray. This one still makes use of `feasible(x) -> bool` function.

11.3 Advanced: Search spaces with `enough(x) -> bool`

11.3.1 Problem [KSmallMul]: K-th smallest number in multiplication table

Problem: We define a multiplication table as a $m \times n$ matrix where

$$\texttt{table}[i][j] = i \times j$$

Given m, n, k, find the k-th smallest number in the multiplication table. An example may look as follows:

$$m = 3, n = 3, k = 5 \quad \texttt{table} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix} \quad \text{Answer:3}(1, 2, 2, 3, 3)$$

Discussion: For Kth-Smallest problems like this, what comes to our mind first is Heap. Usually we can maintain a Min-Heap and just pop the top of the Heap for k times. However, that doesn't work out in this problem. We don't have every single number in the entire multiplication table. Instead, we only have the height and the length of the table. If we are to apply Heap method, we need to explicitly calculate these $m \times n$ values and save them to a heap. The time complexity and space complexity of this process are both $O(mn)$, which is quite inefficient.

The search space is not clear. But we can define a function `enough(x) -> bool` that tells us if there are atleast k numbers in the multiplication table that are less than or equal to x. Ofcourse, if there x satisfies k numbers, then $x + 1$ would also satisfy k numbers. We exploit this monotonicity property.

Now, we intend to beat time to be better than $O(mn)$. That means `enough(x) -> bool` should be considerably cheaper to compute. We exploit the following algorithm:

Listing 11.1: Rust implementation of enough function

```rust
fn enough(num: i32, m: i32, n: i32, k: i32) -> bool {
    let mut count = 0;
    // We run iteration for each row
    for row in 1..=m {
        // Each row has (num / row) elements less than or equal to num.
        // However, this has to be contained within n since at most there are n columns.
        let add = (num / row).min(n);
        if add == 0 {
            // Exit and do no further iterations since if there are no numbers
            // lower than num in this row, any further rows won't have any as well.
            break;
        }
        count += add;
    }
    count >= k
}
```

The above algorithm runs in $O(m)$ time.

The search space of course is $[1, m \times n]$. For every x in this search space, we can calculate the number of elements in the multiplication table that are less than or equal to x in $O(m)$ time. Hence, the runtime of this solution is $O(m \cdot log_2(m \times n))$.

Chapter 12

Dynamic Programming

Dynamic Programming in it's crudest sense is nothing more than "clever bruteforce". What that means is while the solution eventually a large brute-force computation, the *cleverness* is in finding a sub-problem that helps solve for the larger problem instances. The key idea is "instance*s*", not a single instance.

Take the following relation for example:

$$\begin{aligned} F(x) &= Max(F(x-1)+1, 1) \\ F(0) &= 0 \end{aligned} \tag{12.1}$$

Here, $F(2)$ will only be used for calculating $F(3)$ and nowhere else.

On the other hand, the following relation:

$$\begin{aligned} F(x) &= Max(Max(nF(x-n)+1) \forall n \in [0, x), 1) \\ F(0) &= 0 \end{aligned} \tag{12.2}$$

uses $F(2)$ for all calculations where $x > 2$ such as $F(3)$, $F(4)$, etc.

The cleverness in the relation 12.2 that can potentially help it get to similar asymptotics as the 12.1 is **memoization**.

12.1 Memoization

Memoization is nothing but trade-off of space to gain in time. Naive implementation of 12.2 in code will yield results with almost constant space but exponential time. The time complexity leads to:

$$O(n) = \sum_{i=0}^{n-1} O(i) + 1 \tag{12.3}$$

If however, after every solution of an instance in range $[0, k)$, the solution is kept in some constant-time accessible storage M, the problem instance k dissolves into:

$$
\begin{aligned}
O(k) &= \sum_{i=0}^{k-1} O(i) + 1 \\
&= \sum_{i=0}^{k-1} O(1) + 1 \\
&= k + 1
\end{aligned}
\tag{12.4}
$$

We are going to use this idea quite generously.

12.2 Most important subproblems

Most of the times while we find ourselves looking at different problem statements, many-a-times at their core, they stem from one of the following few recursive relations:

- **O/1 Knapsack**:
- **Unbounded Knapsack**:
- **Longest Common Subsequence (LCS)**:
- **Longest Increasing Subsequence (LIS)**:

Alphabetical Index