Lesson 2 - Solana Theory / Rust

Lesson Plan

- Solana Community
- Solana Architecture
- Consensus
- Solana Network and History
- Introduction to Rust

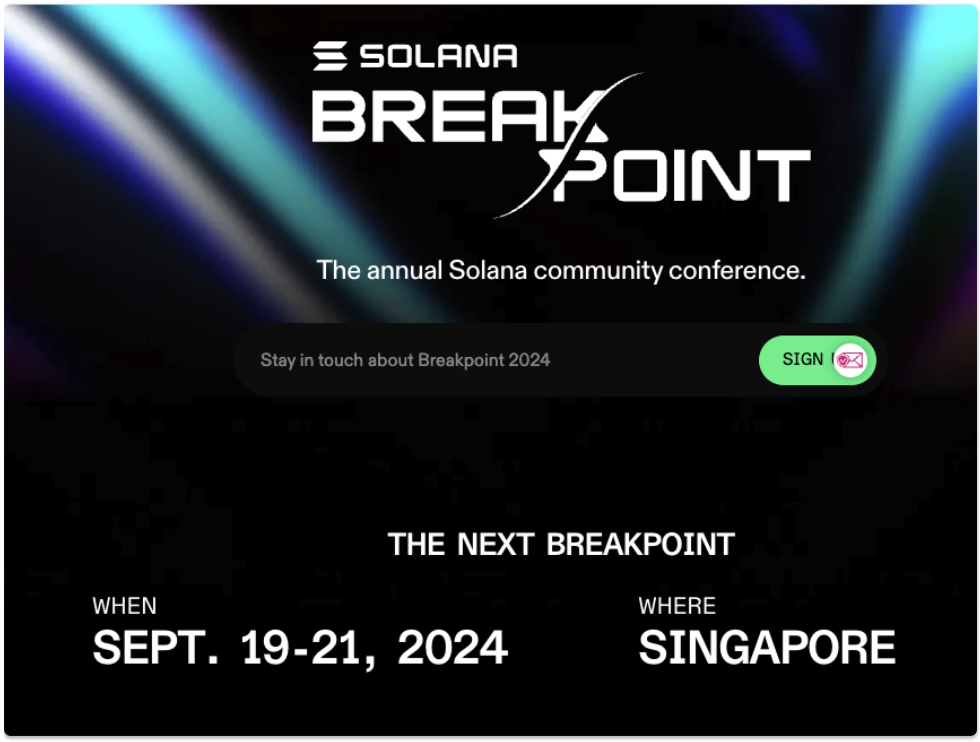
Solana Community

See resources page
This details their telegram / discord channels etc.
There are many meetup groups available worldwide

Hacker House

In 2022 there were hackathons hosted in many cities

Solana Breakpoint - Sept 2024



Solana Collective

See Docs

This is a program to help Solana supporters contribute to the ecosystem and work with core teams.

Solana Grants

Anyone can apply for a grant from the Solana Foundation.
That includes individuals, independent teams, governments, nonprofits, companies, universities, and academics.

Here is the list of initiatives Solana are currently looking to fund.
and categories they are interested in

- Censorship Resistance
- DAO Tooling
- Developer Tooling
- Education
- Payments / Solana Pay
- Financial Inclusion
- Climate Change
- Academic Research

Solana Architecture

Solana Blocks

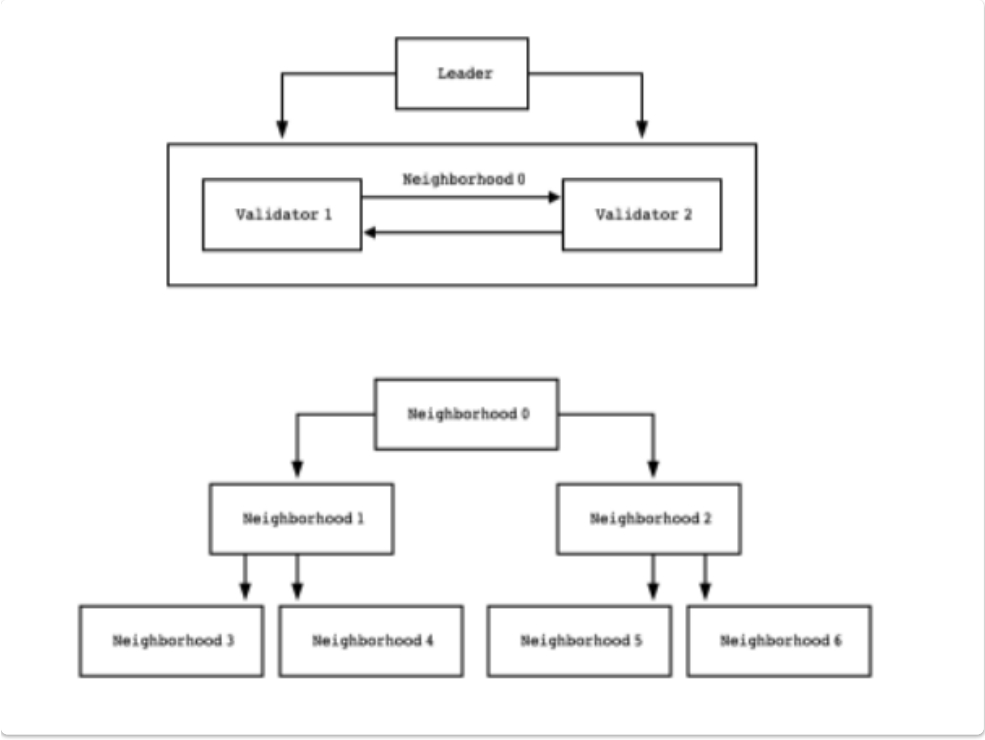
In Solana the concept is a little different, but we still gather transactions together with some meta data.

Overview	
Blockhash	3W9mwZRVvWvnCgwBuGcH4dvTjxp4ZvRceyYkE4aTpSwD
Slot	142,253,865
Timestamp (Local)	Jul 19, 2022 at 18:54:59 GMT+1
Timestamp (UTC)	Jul 19, 2022 at 17:54:59 UTC
Epoch	329
Parent Blockhash	CHJMYUhu7Z4pfZvnvw1EXLZw2B8aYgx3GpvWMfdmygSt
Parent Slot	142,253,864
Child Slot	142,253,866
Processed Transactions	2525
Successful Transactions	1522

Each block is 10MB, blocks are proposed roughly every 800ms,

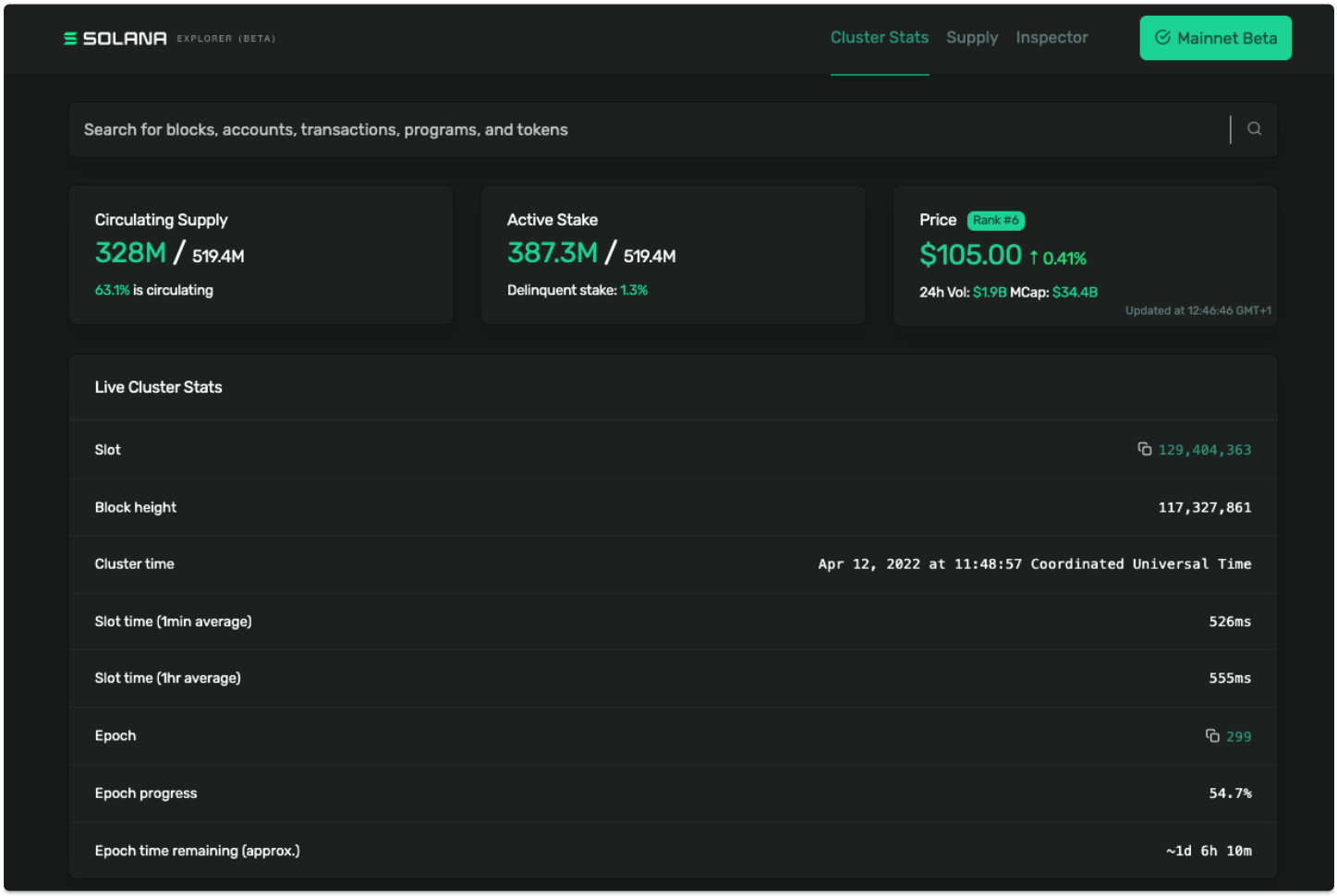
Solana Network

A Solana cluster is a set of validators working together to serve client transactions and maintain the integrity of the ledger. Many clusters may coexist. When two clusters share a common genesis block, they attempt to converge. Otherwise, they simply ignore the existence of the other. Transactions sent to the wrong one are quietly rejected.

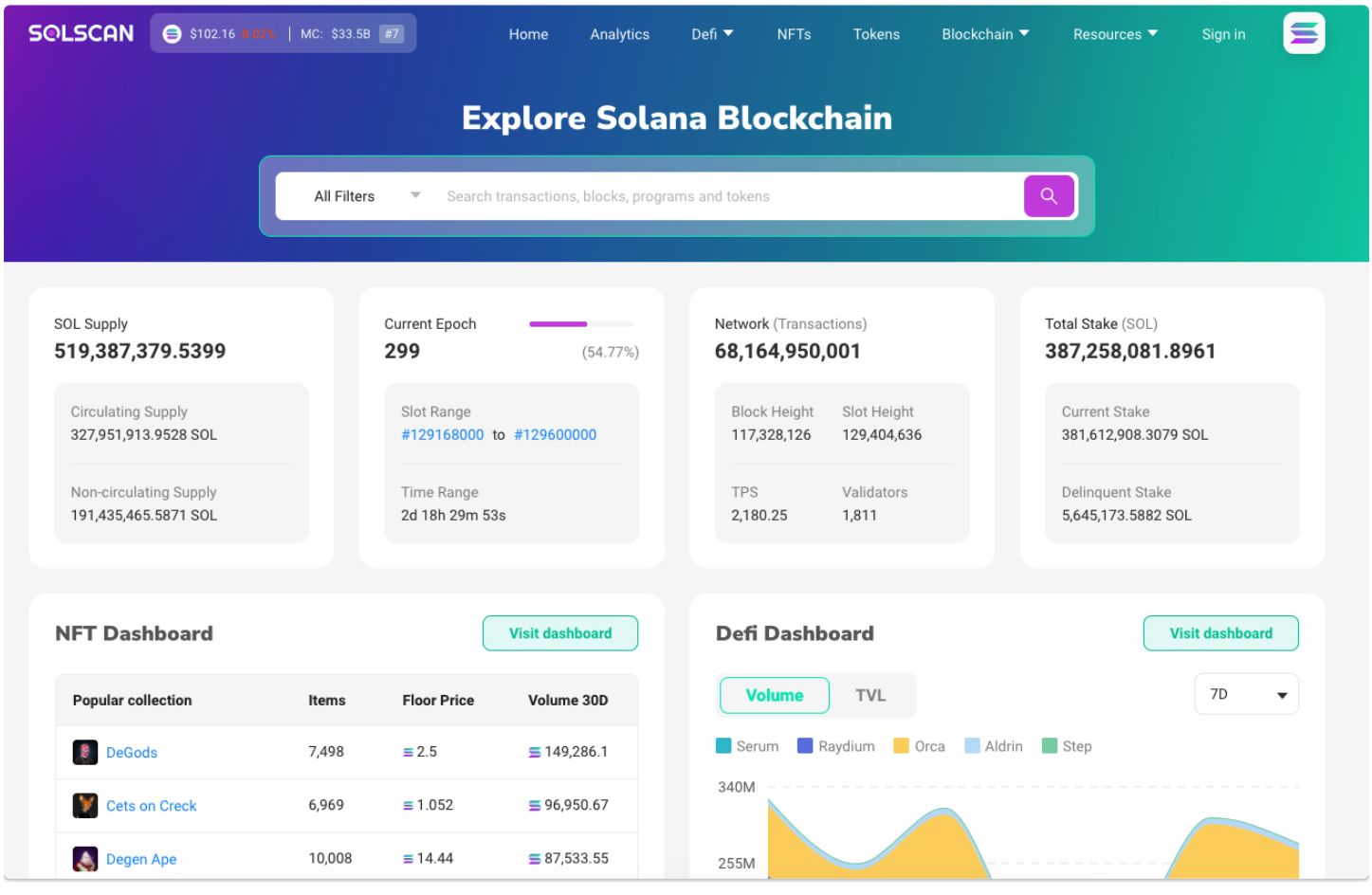


Block Explorers

Solana Explorer



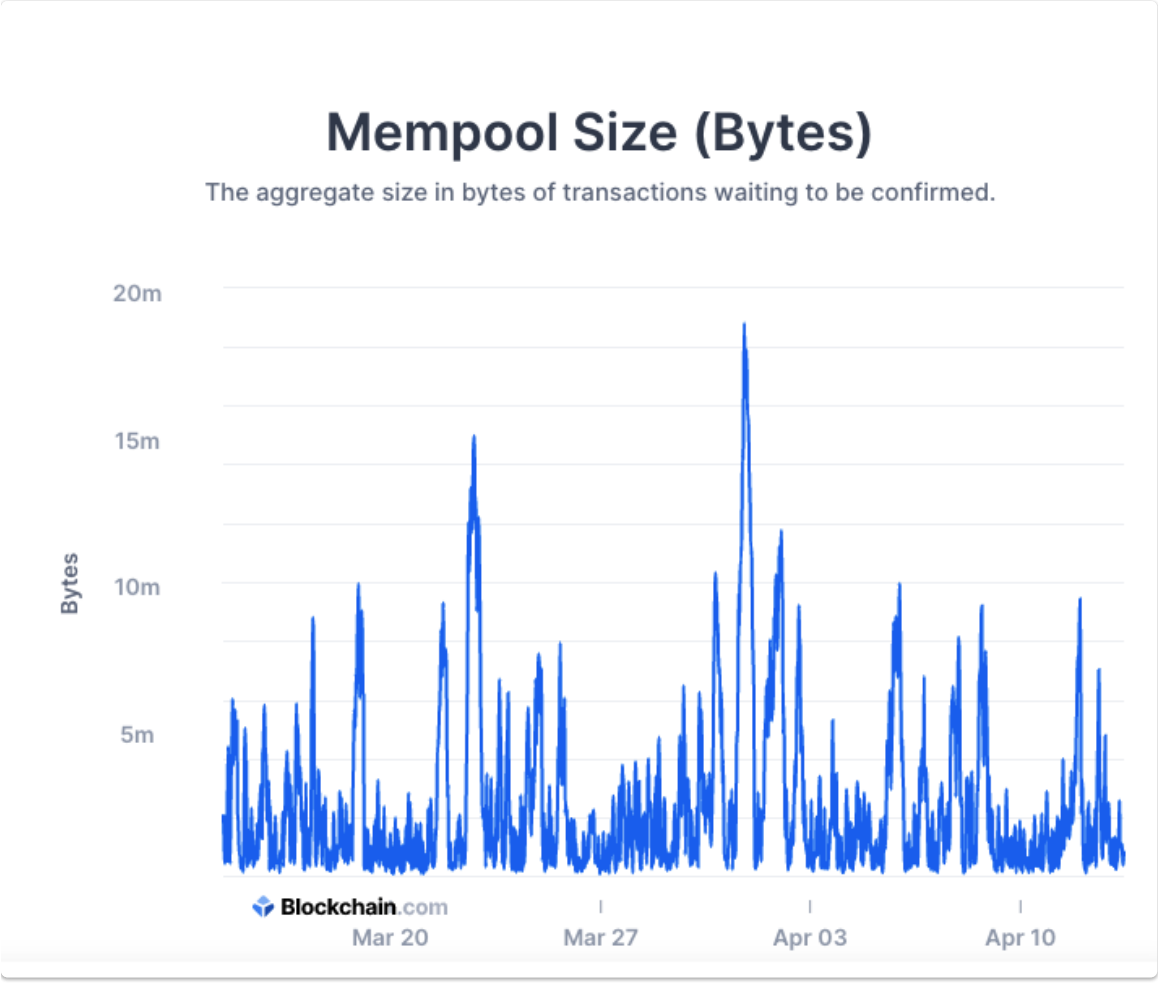
Solscan



[Solana Beach]

The mempool (pending transactions) and Gulfstream

Mempool size in Bitcoin



From Gulfstream docs

"Mempools in Ethereum and Bitcoin are propagated between random nodes in peer-to-peer fashion using a gossip protocol. Nodes in the network periodically construct a bloom filter representing a local mempool and request any transactions that do not match that filter (along with a few others such as a minimal fee) from other nodes on the network. Propagation of a single transaction to the rest of the network will take at least log(N) steps, consumes bandwidth, memory and computational resources required to filter it."

Gulfstream

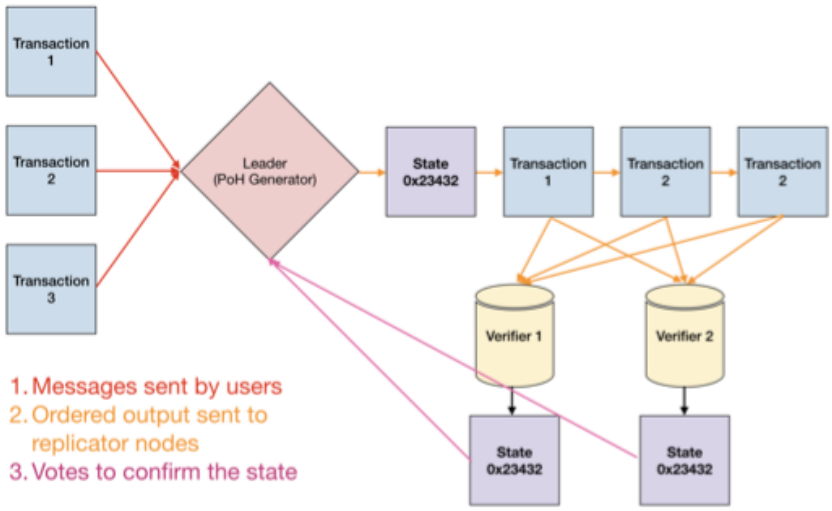
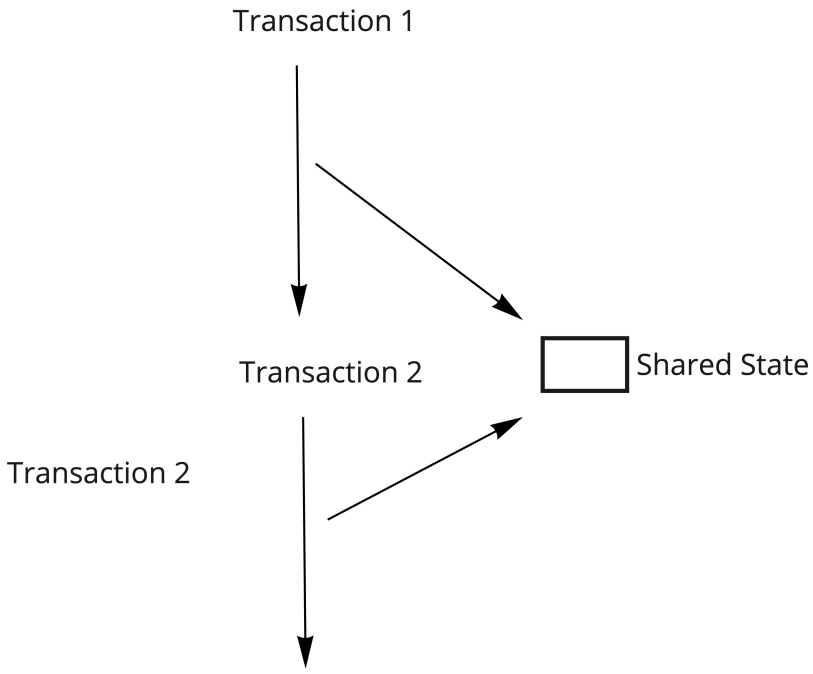


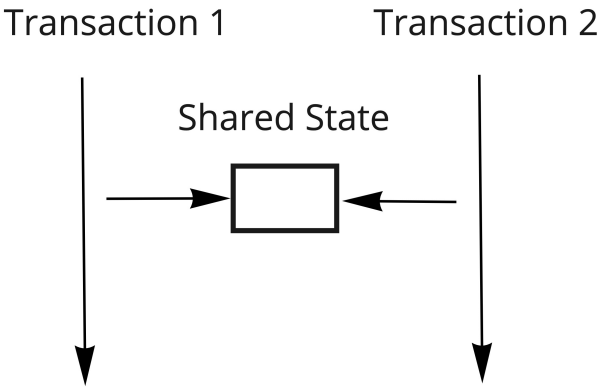
Figure 1: Transaction flow throughout the network.

Since every validator knows the order of upcoming leaders, clients and validators forward transactions to the expected leader ahead of time. This allows validators to execute transactions ahead of time, reduce confirmation times, switch leaders faster, and reduce the memory pressure on validators from the unconfirmed transaction pool. This solution is not possible in networks that have a non-deterministic leader

Transactions reference recent blockhash and the transaction is valid only in the children of the referenced block, and is only valid for about 32 blocks.
Assuming block times of 800 ms, that equates to 24 seconds.

Once a transaction is forwarded to any validator, the validator forwards it to one of the upcoming leaders. Clients can subscribe to transaction confirmations from validators. Clients know that a block-hash expires in a finite period of time, or the transaction is confirmed by the network.
This allows clients to sign transactions that are guaranteed to execute or fail. Once the network moves past the rollback point such that the blockhash the transaction reference has expired, clients have a guarantee that the transaction is now invalid and will never be executed on chain.





How can we improve performance ?

- lets add more threads / cores / do more in parallel

  or
- lets keep everything single threaded

In Ethereum processing of contracts is single threaded

Solana has introduced a way to process programs in parallel

Transaction	Updates
1	Account A
2	Account B
3	Account A
4	Account C

Consensus in distributed systems

- how can participants agree on the state of the system ?

Byzantine fault tolerance (BFT) is the dependability of a fault-tolerant computer system to such conditions where components may fail and there is imperfect information on whether a component has failed.

Why do we need consensus ?

“The double spending problem is a potential flaw in a cryptocurrency or other digital cash scheme whereby the same single digital token can be spent more than once, and this is possible because a digital token consists of a digital file that can be duplicated or falsified.”

Paper

Synchronisation in Distributed Systems

There is a general difficulty with agreeing on time / sequences in distributed systems

In Bitcoin and Ethereum there is no clock available, and participants in the system are given the ability to decide on the sequence of transactions, by mining a block.

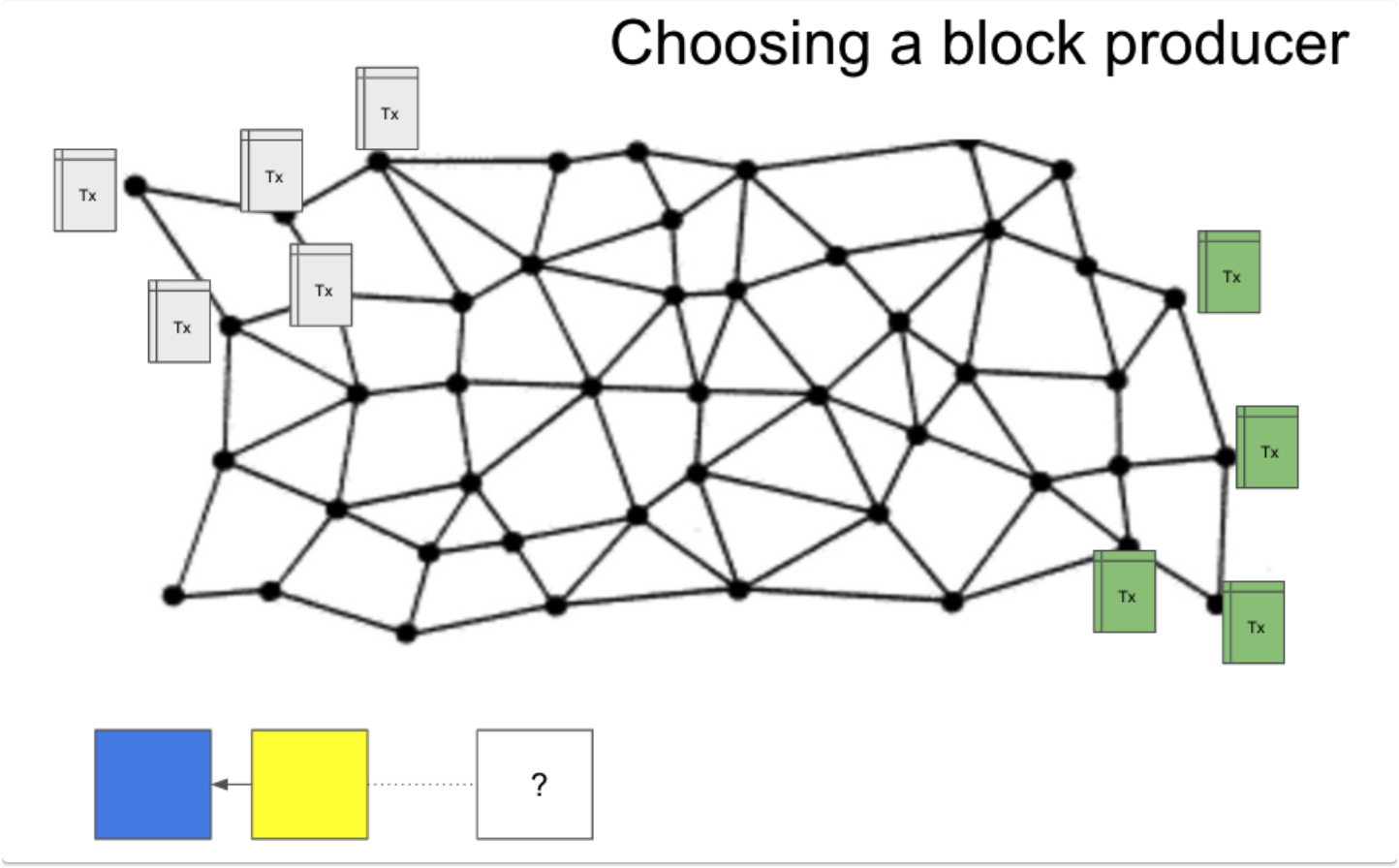
The big innovation from Solana was Proof of History which gives us a verifiable ordering to events

- Proof of Work uses economic incentives to give an ordering of blocks
- Hadera hashgraph uses a median of supplied timestamps

There are 2 Parts to Consensus mechanisms

- Choosing a block producer / leader
- Agreeing on which blocks (transactions) form the canonical chain

For blockchains in general



Typically we allow a block producer to

- choose transactions to be included
- decide on the ordering of transactions

Choosing a leader / block producer

We want this to be 'fair' and difficult to abuse.
PoW uses a race / lottery to solve a puzzle
PoS (DPoS) uses different methods, but often a VRF to assign a time slot to a potential block producer.

One potential problem is liveness

- what if no producer is chosen ?
- what if the chosen block producer fails to produce a block ?

Having a reliable source of time can safely solve timeout issues.

Some Implementations of Sybil / Consensus Mechansisms

- Practical Byzantine Fault Tolerance (pBFT) Castro and Liskov 1999
- Nakamoto Consensus (PoW) 2008

Now there are many “Proof of ….
Stake / Authority / Burn / Elapsed Time / Spacetime ….”
Solana - Proof of History / Tower Consensus (BFT)

Proof of Stake

There are many implementations of PoS - Ethereum , Mina, NXT …

Common features

- Potential block producers have to submit a stake of the native crypto currency to be eligible
- The current block producer is chosen at random, the probability of being chosen will depend on the amount of stake offered.
- If the block producer behaves maliciously they lose some or all of their stake

Ethereum PoS

Ethereum now uses PoS as a means of choosing a block proposer.

- Slots are assigned to producers who have deposited a stake

The consensus mechanism has two parts

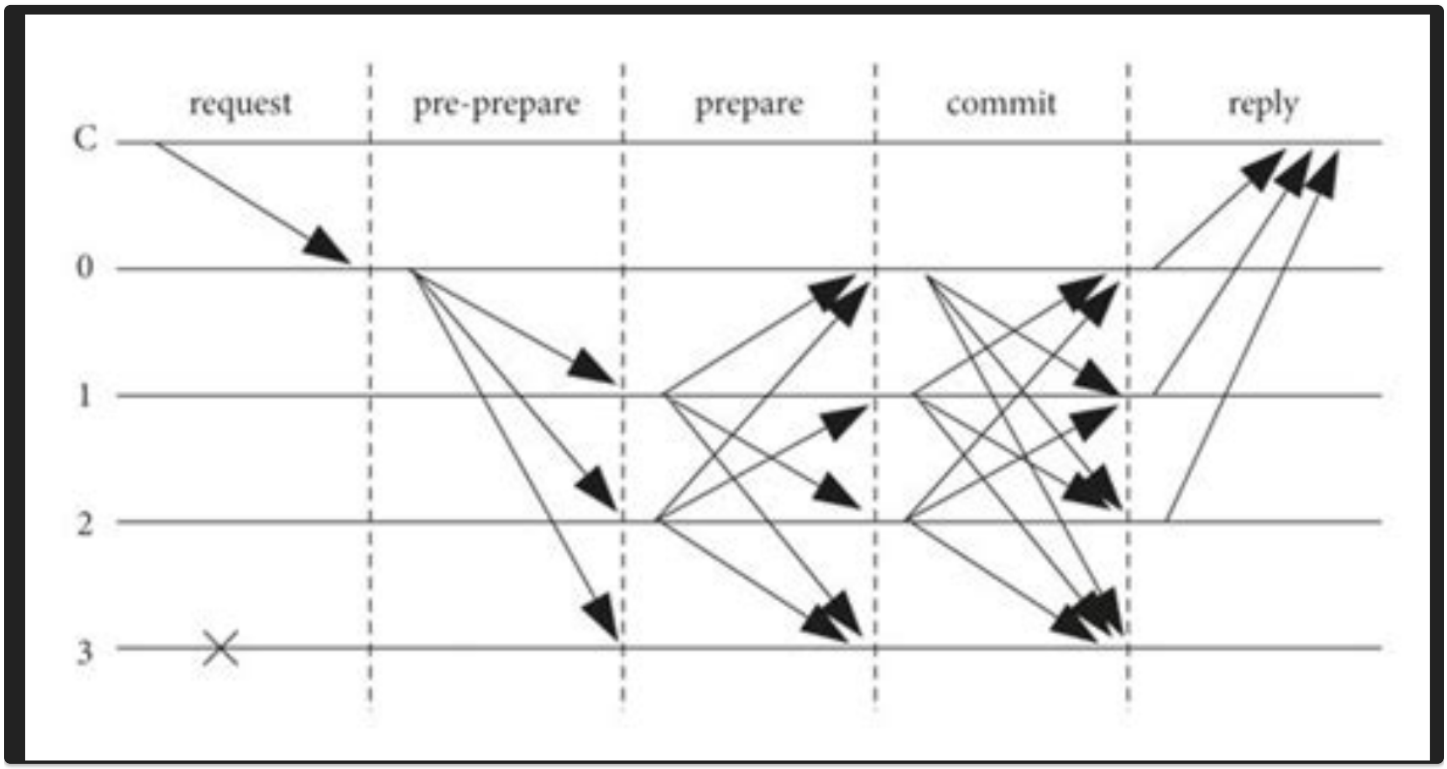
- LMD-GHOST – which adds new blocks and decides what the head of the chain is
- Casper FFG which makes the final decision on which blocks are and are not a part of the chain.

Sui

Narwhal and Bullshark, Sui's Mempool and Consensus Engines
There is a splitting of concerns that you don't get with other mechanisms

- ensuring the availability of data submitted to consensus = Narwhal
- agreeing on a specific ordering of this data = Bullshark

pBFT - Practical Byzantine Fault Tolerance

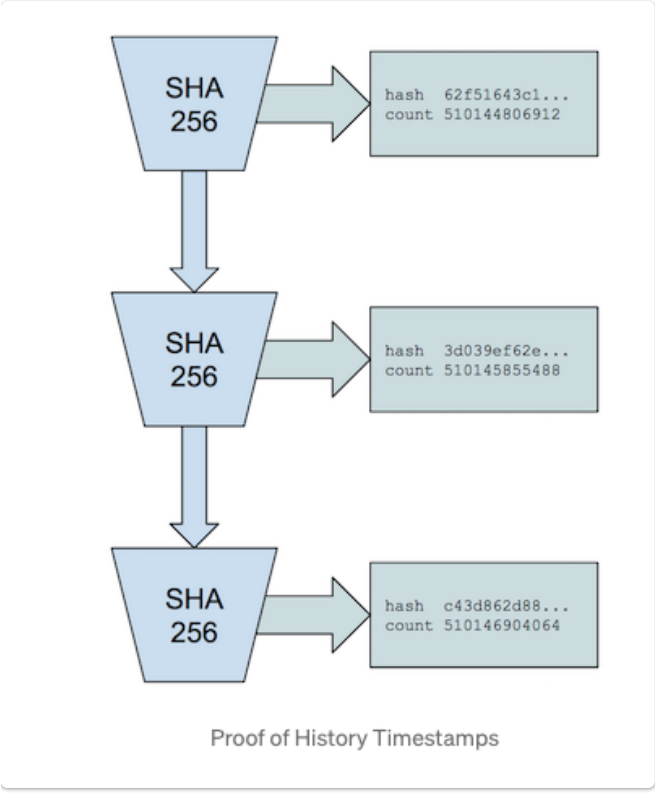


pBFT requires 3f+1 nodes in the system, where f is the maximum number of faulty nodes that the system can tolerate.
Therefore, for the group of nodes to make any decision, approval from 2f+1 nodes is required.

Proof of History

In PoH each node can 'run its own clock' without relying on a central clock.
From Solana docs

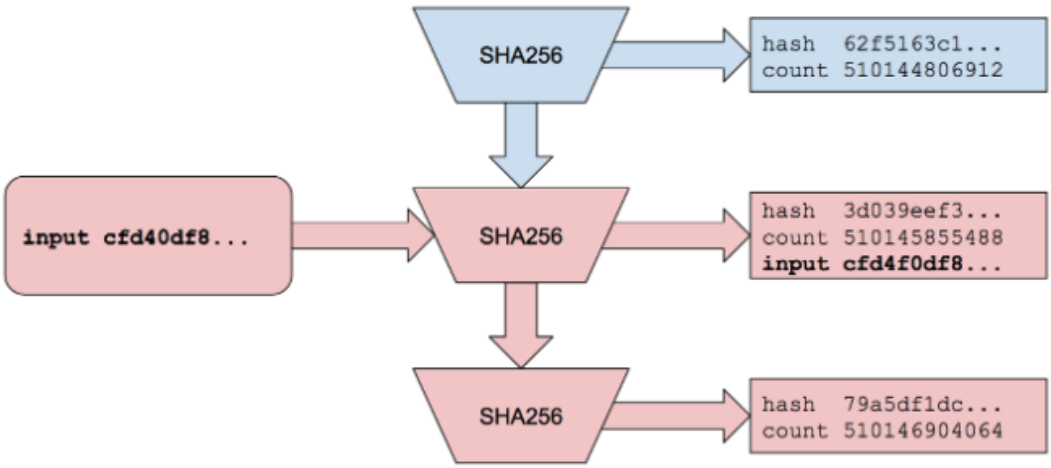
In proof of History we repeatedly hash values, the output from one step forming the input to the next step.



Proof of History Timestamps

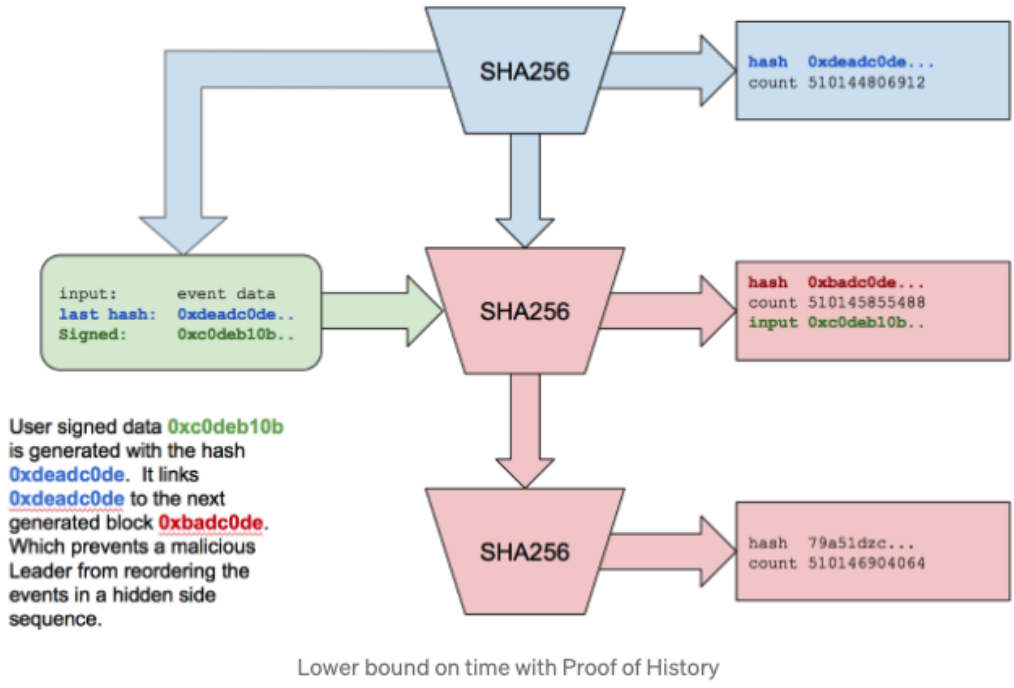
If we were given the hash values and counts out of sequence we would be able to put them into the correct order.

Upper bound on time



Recording messages into a Proof of History sequence

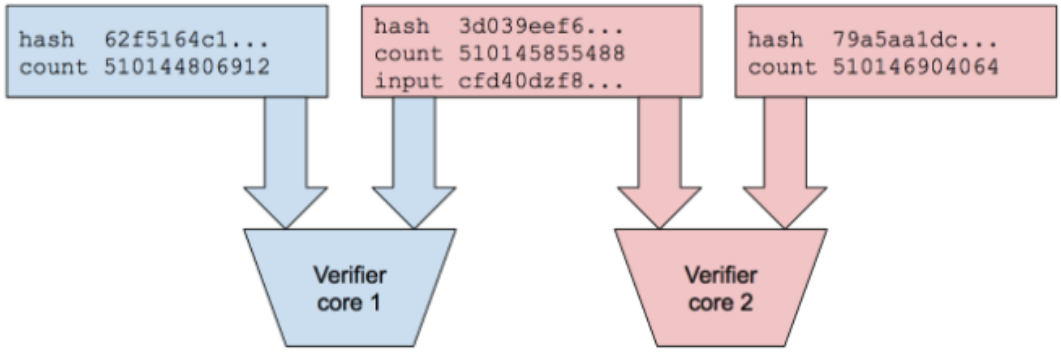
Lower Bound on time



Lower bound on time with Proof of History

Running this process cannot be done in parallel , we need to complete each step in turn.

However, once we have the steps we can verify in parallel.



Parallel Verification

Core 1

Index	Data	Output Hash
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

Core 2

Index	Data	Output Hash
300	sha256(hash299)	hash300
400	sha256(hash399)	hash400

A useful analogy is with a water clock, as in this article

Leader selection

Leader Schedule Generation Algorithm

Leader schedule is generated using a predefined seed. The process is as follows:

1. Periodically use the PoH tick height (a monotonically increasing counter) to seed a stable pseudo-random algorithm.
2. At that height, sample the bank for all the staked accounts with leader identities that have voted within a cluster-configured number of ticks. The sample is called the *active set*.
3. Sort the active set by stake weight.
4. Use the random seed to select nodes weighted by stake to create a stake-weighted ordering.
5. This ordering becomes valid after a cluster-configured number of ticks.

The advantages of deterministic leader selection

Since every validator knows the order of upcoming leaders, clients and validators forward transactions to the expected leader ahead of time. This

allows validators to execute transactions ahead of time, reduce confirmation times, switch leaders faster, and reduce the memory pressure on validators from the unconfirmed transaction pool. This solution is not possible in networks that have a non-deterministic leader

This system lowers latency and increases throughput because slot leaders can stream transactions to the rest of the validators in real-time rather than waiting to fill an entire block and send it at once.

As validators keep the count of time, they can stamp each incoming transaction with a time, or proof-of-history value, so the other nodes can order transactions within a block correctly even if they aren't streamed in chronological order. The other nodes can then verify these transactions as they come in rather than having to review an entire block of transactions at once.

Useful articles

Sol overview
Proof-of-history - Medium (by the founder)

Tower BFT (Proof of Stake)

Solana implements a derivation of pBFT, but with one fundamental difference. Proof of History (PoH) provides a global source of time before consensus. Solana's implementation of pBFT uses the PoH as the network clock of time, and the exponentially-increasing time-outs that replicas use in pBFT can be computed and enforced in the PoH itself.

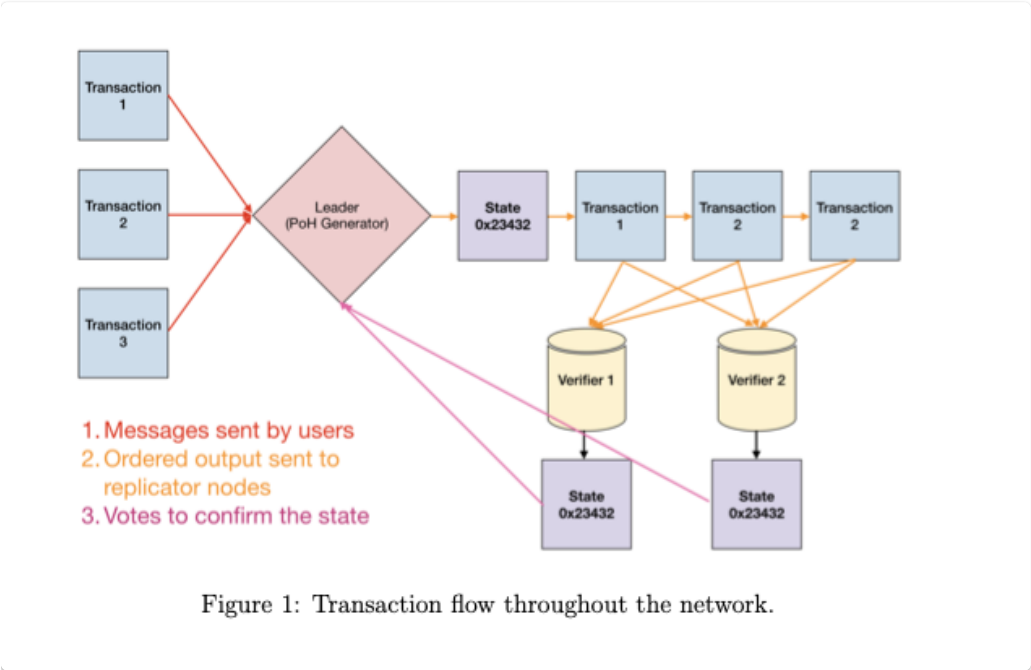


Figure 1: Transaction flow throughout the network.

The leader will be able to publish a signature of the state at a predefined period.
Each bonded validator must confirm that signature by publishing their own signed signature of the state. The vote is a simple yes vote, without a no.

If super majority of the bonded identities have voted within a timeout, then this branch would be accepted as valid.

Every 400ms, the network has a potential rollback point, but every subsequent vote doubles the amount of real time that the network would have to stall before it can unroll that vote.

Once ⅔ of validators have voted on some PoH hash, that PoH hash is canonicalized, and cannot be rolled back. This is distinct from proof of work, in which there is no notion of canonicalization.

Solana History

See Docs

Anatoly Yakovenko published a whitepaper in November 2017 specifying proof of history.
The codebase was moved to Rust and became project Loom.
In Feb 2018 a throughput of > 10K transactions per second was verified.
In March 2018 the project was renamed to Solana to avoid confusion with existing projects.
In July 2018 a testnet of 50 nodes was built which managed up to 250K transactions per second.
In December 2018 the testnet was increased to 150 nodes, and the throughput averaged 200K transactions per second , peaking at 500K.
October 2020 - Wormhole bridge launched (Solana to Ethereum)
November 2022 - Solana affected by collapse of FTX, some stablecoin trading halted.
April 2023 - Solana Saga phone available

Introduction to Rust

Core Features

- Memory safety without garbage collection
- Concurrency without data races
- Abstraction without overhead

Variables

Variable bindings are immutable by default, but this can be overridden using the `mut` modifier

```language-rust
let x = 1;
let mut y = 1;
```

Types

[Data Types -Rust book]

The Rust compiler can infer the types that you are using, given the information you already gave it.

[Scalar Types]

A *scalar* type represents a single value. Rust has four primary scalar types:

- integers
- floating-point numbers
- booleans
- characters

Integers

For example

```language-rust
u8, i32, u64
```

Floating point

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively.

The default type is `f64` because on modern CPUs it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

boolean

The boolean type or bool is a primitive data type that can take on one of two values, called true and false . (size of 1 byte)

char

`char` in Rust is a unique integral value representing a Unicode Scalar value
Note that unlike C, C++ this cannot be treated as a numeric type.

Other scalar types

usize

`usize` is pointer-sized, thus its actual size depends on the architecture your are compiling your program for
As an example, on a 32 bit x86 computer, `usize = u32`, while on x86_64 computers, `usize = u64`.
`usize` gives you the guarantee to be always big enough to hold any pointer or any offset in a data structure, while `u32` can be too small on some architectures.
Rust states the size of a type is not stable in cross compilations except for primitive types.

Compound Types

Compound types can group multiple values into one type.

- tuples
- arrays
- struct

Tuples

Example

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Struct

```rust
struct User {
        name : String,
        age: u32,
        email: String,
}
```

Collections

- Vectors

```rust
let names = vec!["Bob", "Frank", "Ferris"];
```

We will cover these in more detail later

Strings

Based on UTF-8 - Unicode Transformation Format

Two string types:

- `&str` a view of a sequence of UTF8 encoded dynamic bytes, stored in binary, stack or heap. Size is unknown and it points to the first byte of the string
- `String`: growable, mutable, owned, UTF-8 encoded string. Always allocated on the heap. Includes capacity i.e. memory allocated for this string.

A String literal is a string slice stored in the application binary (i.e. there at compile time).

String vs str

`String` - heap allocated, growable UTF-8
`&str` - reference to UTF-8 string slice (could be heap, stack ...)

String vs &str - StackOverflow
Rust overview - presentation
Let's Get Rusty - Strings

Arrays

Rust book definition of an array:

"An array is a collection of objects of the same type `T`, stored in contiguous memory. Arrays are created using brackets `[]`, and their length, which is known at compile time, is part of their type signature `[T; length]`."

Array features:

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

Array declarations

```language-rust
//Syntax1: No type definition
let variable_name = [value1,value2,value3];

let arr = [1,2,3,4,5];

//Syntax2: Data type and size specified
let variable_name:[dataType;size] =
[value1,value2,value3];

let arr:[i32;5] = [1,2,3,4,5];

//Syntax3: Default valued array
let variable_name:[dataType;size] =
[default_value_for_elements,size];
```

```
let arr:[i32;3] = [0;3];

// Mutable array
let mut arr_mut:[i32;5] = [1,2,3,4,5];

// Immutable array
let arr_immut:[i32;5] = [1,2,3,4,5];
```

Rust book definition of a slice:

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.

Arrays - TutorialsPoint
Arrays and Slices - RustBook

Numeric Literals

The compiler can usually infer the type of an integer literal, but you can add a suffix to specify it, e.g.
`42u8`
It usually defaults to `i32` if there is a choice of the type.

Hexadecimal, octal and binary literals are denoted by prefixes `0x` , `0o` , and `0b` respectively

To make your code more readable you can use underscores with numeric literals
e.g.

```language-rust
1_234_567_890
```

ASCII code literals

Byte literals can be used to specify ASCII codes
e.g.
`b'C'`

Conversion between types

Rust is unlike many languages in that it rarely performs implicit conversion between numeric types, if you need to do that, it has to be done explicitly.
To perform casts between types you use the `as` keyword
For example

```language-rust
let a = 12;
let b = a as usize;
```

Enums

See docs

Use the keyword `enum`

```language-rust
enum Fruit {
    Apple,
    Orange,
    Grape,
}
```

You can then reference the enum with for example

```language-rust
Fruit::Orange
```

Functions

Functions are declared with the `fn` keyword, and follow familiar syntax for the parameters and function body.

```language-rust
fn my_func(a: u32) -> bool {
    if a == 0 {
        return false;
    }
    a == 7
}
```

As you can see the final line in the function acts as a return from the function
Typically the return keyword is used where we are leaving the function before the end.

Loops

Range:

- inclusive start, exclusive end
  `for n in 1..101 {}`
- inclusive end, inclusive end
  `for n in 1..=101 {}`
    - inclusive end, inclusive end, every 2nd value

```language-rust
for n in (1..=101).step_by(2){}
```

We have already seen for loops to loop over a range, other ways to loop include
`loop` - to loop until we hit a `break`
`while` which allows an ending condition to be specified
See Rust book for examples.

Control Flow

If expressions

See Docs

The `if` keyword is followed by a condition, which *must evaluate to bool*, note that Rust does not automatically convert numerics to bool.

```language-rust
if x < 4 {
        println!("lower");
} else {
        println!("higher");
}
```

Note that 'if' is an expression rather than a statement, and as such can return a value to a 'let' statement, such as

```language-rust
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

Note that the possible values of `number` here need to be of the same type.

We also have `else if` and `else` as we do in other languages.

Printing

```language-rust
 println!("Hello, world!");

 println!("{:?} tokens", 19);
```

Option

We may need to handle situations where a statement or function doesn't return us the value we are expecting, for this we can use Option.

Option is an enum defined in the standard library.
The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

It is useful in avoiding inadvertently handling null values.

Another useful enum is `Result`

```language-rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Matching

A powerful and flexible way to handle different conditions is via the `match` keyword
This is more flexible than an `if` expression in that the condition does not have to be a boolean, and pattern matching is possible.

Match Syntax

```rust
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Example

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

The keyword `match` is followed by an expression, in this case `coin`
The value of this is matched against the 'arms' in the expression.
Each `arm` is made of a pattern and some code
If the value matches the pattern, then the code is executed, each arm is an

expression, so the return value of the whole match expression, is the value of the code in the arm that matched.

Matching with `Option`

```language-rust
fn main() {
    fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

Installing Rust

The easiest way is via rustup

See Docs

Mac / Linux

```bash
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf
| sh
```

Windows

See details here

download and run `rustup-init.exe`.

Other methods

Cargo

See the docs

Cargo is the rust package manager, it will

- download and manage your dependencies,
- compile and build your code
- make distributable packages and upload them to public registries.

Some common cargo commands are (see all commands with --list):

```
 build, b      Compile the current package
check, c      Analyse the current package and report errors,
but don't build
                          object files
clean         Remove the target directory
doc, d        Build this package's and its dependencies'
documentation
new           Create a new cargo package
init          Create a new cargo package in an existing
directory
add           Add dependencies to a manifest file
run, r        Run a binary or example of the local package
test, t       Run the tests
bench         Run the benchmarks
update        Update dependencies listed in Cargo.lock
search        Search registry for crates
publish       Package and upload this package to the
registry
install       Install a Rust binary. Default location is
$HOME/.cargo/bin
uninstall     Uninstall a Rust binary
```

See `cargo help` for more information on a specific command.

Useful Resources

Rustlings

Rust by example

Rust Lang Docs

Rust Playground

Rust Forum

Rust Discord

Running a validator

Node requirements

1. Validator node:

- CPU
  - 12 cores / 24 threads, or more
  - 2.8GHz, or faster
  - AVX2 instruction support (to use official release binaries, self-compile otherwise)
  - Support for AVX512f and/or SHA-NI instructions is helpful
  - The AMD Zen3 series is popular with the validator community
- RAM
  - 128GB, or more
  - Motherboard with 256GB capacity suggested
- Disk
  - PCIe Gen3 x4 NVME SSD, or better
  - Accounts: 500GB, or larger. High TBW (Total Bytes Written)
  - Ledger: 1TB or larger. High TBW suggested
  - OS: (Optional) 500GB, or larger. SATA OK
  - The OS may be installed on the ledger disk, though testing has shown better performance with the ledger on its own disk
  - Accounts and ledger *can* be stored on the same disk, however due to high IOPS, this is not recommended
  - The Samsung 970 and 980 Pro series SSDs are popular with the validator community
- GPUs
  - Not strictly necessary at this time
  - Motherboard and power supply speced to add one or more high-end GPUs in the future suggested

2. RPC node:
    - CPU
        - 16 cores / 32 threads, or more
    - RAM
        - 256 GB, or more
    - Disk
        - Consider a larger ledger disk if longer transaction history is required
        - Accounts and ledger should not be stored on the same disk

Internet service should be at least 300Mbit/s symmetric, commercial. 1GBit/s preferred.

Price of operation

Solana validators must pay to be eligible to vote. This means a fixed cost of roughly 3 SOL every epoch (2-3 days), which at the time of writing equals costs of ~$100 every single day. This is for transactions which happen on chain and are part of the consensus mechanism.
The validator profits by charging commission on the accounts that delegate to it.

Right now it is required to have combined delegated and staked 50,000 SOL in order to run a node at a profit.

Rewards

Validators earn SOL in two ways:

- staking reward
- commission on 3rd party stake
- 50% of the transaction fee

Rewards are fixed per epoch and are divided amongst the nodes the support the network. Rewards are divided according to:

- stake weight
- participation

As time goes on inflation rate will decrease and with it associated epoch rewards.