Solana Overview

General Information

- **Currency:** SOL = 10^9 Lamports.
- The current implementation sets **block time** to 800ms.
- **Proof of History (PoH)** is not a consensus mechanism, but it is used to improve the performance of Solana's Proof of Stake consensus. It acts as a decentralized trusted clock.
- Solana is using **PoS** (Proof of Stake) as a consensus algorithm, at any given moment, a cluster expects only one validator to produce ledger entries, which is the one assigned as a leader. The lifetime of a leader schedule is called an **Epoch**, the epoch is split into slots, where each slot has a duration of PoH ticks.

Transactions

Solana Transaction Structure

Field	Description
`signatures`	List of signatures.
`accounts`	List of accounts (read-only / read-write)
`recentBlockhash`	Blockhash of recently produced block used as nonce.
`instructions`	List of instructions which each call an on-chain program.

- The first account in the transaction **accounts** list is roughly the same thing as the sender in an Ethereum transaction. It is the account that will be used to pay transaction fees and Solana will verify that the first signature in the transaction signatures list was produced by that account.
- On Ethereum, you would need to pass signatures inside transaction data and verify them inside a smart contract. On Solana, **signatures** can be appended to the transaction signatures list and, since Solana nodes use a GPU to verify signatures, will be verified much more efficiently than they would inside a program.
- Solana transactions can actually **list multiple smart contracts** to call and so they don't have a single "to" field. Instead, they may list one or more **instructions**

which each represent a smart contract call. Each instruction specifies its own smart contract address and the input parameters for the call.

- The Solana VM doesn't have a dynamic gas model for transactions (like Ethereum). Instead, it has a fixed maximum compute cost which currently cannot be adjusted. It naturally puts pressure on developers to optimize on-chain code to fit within the system limits. Transactions do have fees on Solana, though. Currently, **transaction fee calculation** is very simple, each signature in a transaction costs an additional 5000 lamports.
- Solana **instruction's data is limited in size**. The entire encoded size of a Solana transaction cannot exceed 1232 bytes. This constraint allows Solana to optimize its networking layer for quickly passing transactions between nodes (smaller packets = less delay).
- It is important that high-throughput applications **split up state** into multiple accounts because if each transaction modifies the same account, transactions will have to be processed serially.
- Accounts may be annotated as **read-write or read-only accounts**. If an on-chain program modifies a read-only account, the transaction will be reverted. The first account will always be read-write since it is used to cover transaction fees.

Account Model

- An account is either a "**data account**" which holds data or an immutable "**program account**", they are also called non-executable and executable accounts.
- Each account of both types have the following fields:

Field	Description
`lamports`	The number of lamports owned by this account.
`owner`	The program owner of this account.
`executable`	Whether this account can process instructions.
`data`	The raw data byte array stored by this account.
`rent_epoch`	The next epoch that this account will owe rent.

- Each account is owned by a program. By default, a newly created account is owned by a built-in program called the "**System Program**" Only the owner of an account can modify it.
- *one advantage of the program/account model is you can have one generic program that operates on various data. the best example of this is the SPL token program. to create a new token, you dont need to deploy code like you do on ethereum. you create an account that can mint tokens, and more accounts that can receive them. the mint address uniquely determines the token type, and these are all passed as arguments to one static program instance [1].*
- On Ethereum, only smart contracts have storage and naturally, they have full control over that storage. On Solana, any account can store state but the storage for smart contracts is only used to store the immutable byte code. On Solana, the state of a smart contract is actually completely stored in other accounts. To ensure that contracts can't modify another contract's state, each account assigns an owner contract that has exclusive control over state mutations.
- In Sealevel, any account's data can be read or written to by a contract. However, the runtime enforces that only an account's "owner" is allowed to modify it. Changes by any other programs will be reverted and cause the transaction to fail.
- **Account Storage:** In the Solana Sealevel VM, all accounts can store data. However, executable account data is used exclusively for immutable byte code which is used to process transactions. So where can smart contract developers store their data? They can store the data in non-executable accounts which are owned by the executable account. Developers can create new accounts with an assigned owner equal to the address of their executable account to store data.
- **Balance:** once an account is owned by a program, the private key cannot be used to transfer lamports with the System Program since the System Program no longer has permission to send lamports from the account.
- In Sealevel, executable accounts are created just like normal accounts but their owner must be set to a loader program. The loader program processes transactions to write byte code into account data and only once the program passes the loader's verification process, will it be marked as executable.

Rent

- Rent fees are charged every epoch (~2 days) and are determined by account size.
- Accounts with sufficient balance to cover 2 years of rent are exempt from fees.
- If accounts are not required to be rent-exempt, they may eventually run out of lamports and be deleted by the runtime. Once deleted, accounts can be recreated. For this reason, accounts that rely on certain data storage to be present should enforce that accounts are exempt from rent before writing data.
- Accounts with zero balance will be deleted at the end of transaction processing.

- Temporary accounts with zero balance may be created during a transaction.
- Since executable accounts are immutable, any lamports in the account will be frozen when the account is marked executable. The Lamport balance should therefore be no more than the minimum balance required for rent-free storage.

Solana programs

- **Sealevel,** Solana's parallel smart contracts runtime. one thing to consider is that EVM and EOS's WASM-based runtimes are all single-threaded. That means that one contract at a time modifies the blockchain state. What is built in Solana is a runtime that can process tens of thousands of contracts in parallel, using as many cores as are available to the Validator.
- To prevent a program from abusing **computation resources** each instruction in a transaction is given a computing budget. The budget consists of computation units that are consumed as the program performs various operations and bounds that the program may not exceed. When the program consumes its entire budget or exceeds a bound then the runtime halts the program and returns an error.
- Solana on-chain programs are compiled via the **LLVM compiler** infrastructure to an Executable and Linkable Format (ELF) containing a variation of the **Berkeley Packet Filter (BPF)** bytecode.
  Because Solana uses the LLVM compiler infrastructure, a program may be written in any programming language that can target the LLVM's BPF backend. Solana currently supports writing programs in Rust and C/C++.
- **Stack:** BPF VM uses stack frames instead of a variable stack pointer. Each stack frame is limited to 4KB in size. If a program violates that stack frame size, the compiler will report the overrun as a warning.
- **Call Depth:** Programs are constrained to run quickly and to facilitate this: the program's call stack is limited to a max depth of 64 frames.
- **Heap:** Programs have access to a runtime heap either directly in C or via the Rust "alloc" APIs. To facilitate fast allocations, a simple 32KB bump heap is utilized. The heap does not support "free" or "realloc" so use it wisely.
- Programs support a limited subset of Rust's **float operations**, if a program attempts to use a float operation that is not supported, the runtime will report an unresolved symbol error. Also, the BPF instruction set does not support **signed division**.
- **Deployment:** BPF program deployment is the process of uploading a BPF shared object into a program account's data and marking the account executable.
- **Native programs:** Solana contains a small handful of native programs, which are required to run validator nodes. Unlike third-party programs, the native programs are part of the validator implementation and can be upgraded as part of cluster upgrades. Upgrades may occur to add features, fix bugs, or improve performance.

A list of native programs and their Ids can be found here:
https://docs.solana.com/developing/runtime-facilities/programs

Accounts

- If the program needs to store state between transactions, it does so using accounts. Accounts are similar to files in operating systems such as Linux in that they may hold arbitrary data that persists beyond the lifetime of a program. Transactions can indicate that some of the accounts it references are to be treated as read-only accounts in order to enable **parallel account processing** between transactions. The runtime permits read-only accounts to be read concurrently by multiple programs. If a program attempts to modify a read-only account, the transaction is rejected by the runtime.
- If an account is marked **"executable"** in its metadata, then it is considered a program that can be executed by including the account's public key in an instruction's program id. Accounts are marked as executable during a successful program deployment process by the loader that owns the account.
- If a program is marked as final (non-upgradeable), the runtime enforces that the account's data (the program) is immutable. Through the **upgradeable loader,** it is possible to upload a totally new program to an existing program address.
- The current **maximum size** of an account's data is 10 megabytes.
- At the time of writing, programs cannot increase the data size of accounts they own. They must copy data from one account to a larger account if they need more data. For this reason, most programs do not store dynamically sized maps and arrays in account data. Instead, they store this data in many accounts. For example, each key-value pair of an EVM mapping can be stored in a new account.
- **Rent:** Keeping accounts alive on Solana incurs a storage cost called rent because the blockchain cluster must actively maintain the data to process any future transactions. This is different from Bitcoin and Ethereum, where storing accounts doesn't incur any costs.
  As of writing, the fixed rent fee is 19.055441478439427 lamports per byte-epoch on the testnet and mainnet-beta clusters. An epoch is targeted to be 2 days (For devnet, the rent fee is 0.3608183131797095 lamports per byte-epoch with its 54m36s-long epoch).
  This value is calculated to target 0.01 SOL per mebibyte-day (exactly matching 3.56 SOL per mebibyte-year)
  The rent calculation includes account metadata (address, owner, lamports, etc) in the size of an account. Therefore the smallest an account can be for rent calculations is 128 bytes.
  For example, an account is created with the initial transfer of 10,000 lamports and no additional data. Rent is immediately debited from it on creation, resulting in a balance of 7,561 lamports

Alternatively, an account can be made entirely exempt from rent collection by depositing at least 2 years' worth of rent. This is checked every time an account's balance is reduced, and rent is immediately debited once the balance goes below the minimum amount.
Rent fees can slowly drain an account's balance, programs must consider whether to enforce that accounts they use for storage should be required to be rent-exempt. If accounts are not required to be rent-exempt, they may eventually run out of lamports and be deleted by the runtime. Once deleted, accounts can be recreated. For this reason, accounts that rely on certain data storage to be present should enforce that accounts are exempt from rent before writing data.

- Accounts with **zero balance** will be deleted at the end of transaction processing. Temporary accounts with zero balance may be created during a transaction.

Development

- **Memory map:** The virtual address memory map used by Solana BPF programs is fixed and laid out as follows
    - Program code starts at 0x100000000
    - Stack data starts at 0x200000000
    - Heap data starts at 0x300000000
    - Program input parameters start at 0x400000000

  The above virtual addresses are start addresses but programs are given access to a subset of the memory map. The program will panic if it attempts to read or write to a virtual address that it was not granted access to
- **Static Writable Data:** Programs' shared objects do not support writable shared data. Programs are shared between multiple parallel executions using the same shared read-only code and data. This means that developers should not include any static writable or global variables in programs. In the future, a copy-on-write mechanism could be added to support writable data.
- Solana Rust programs follow the typical Rust project layout: https://doc.rust-lang.org/cargo/guide/project-layout.html

Parameter Serialization and Entrypoint

- **Serialization:** BPF loaders serialize the program input parameters into a byte array that is then passed to the program's entrypoint, where the program is responsible for deserializing it on-chain.
  The latest loader serializes the program input parameters as follows (all encoding is little endian):
    - 8 bytes unsigned number of accounts
    - For each account

    - 1 byte indicating if this is a duplicate account, if not a duplicate then the value is 0xff, otherwise the value is the index of the account it is a duplicate of.
    - If duplicate: 7 bytes of padding
    - If not duplicate:
        - 1 byte boolean, true if account is a signer
        - 1 byte boolean, true if account is writable
        - 1 byte boolean, true if account is executable
        - 4 bytes of padding
        - 32 bytes of the account public key
        - 32 bytes of the account's owner public key
        - 8 bytes unsigned number of lamports owned by the account
        - 8 bytes unsigned number of bytes of account data
        - x bytes of account data
        - 10k bytes of padding, used for realloc
        - enough padding to align the offset to 8 bytes.
        - 8 bytes rent epoch
    - 8 bytes of unsigned number of instruction data
    - x bytes of instruction data
    - 32 bytes of the program id
- **Entrypoint:** Programs export a known entry point symbol which the Solana runtime looks up and calls when invoking a program. Solana supports multiple versions of the BPF loader and the entry points may vary between them. Programs must be written for and deployed to the same loader.
- **Deserialization:** Each loader provides a helper function that deserializes the program's input parameters into Rust types.

Data Types

- The loader's entrypoint macros call the program defined instruction processor function with the following parameters:

  program_id: &Pubkey,
  accounts: &[AccountInfo],
  instruction_data: &[u8]

- The *program id* is the public key of the currently executing program.
- The *accounts* is an ordered slice of the accounts referenced by the instruction An account's place in the array signifies its meaning, for example, when transferring lamports an instruction may define the first account as the source and the second as the destination.

- The members of the *AccountInfo* structure are read-only except for lamports and data. Both of these members are protected by the Rust *RefCell* construct, so they must be borrowed to read or write to them. The reason for this is they both point back to the original input byte array, but there may be multiple entries in the accounts slice that point to the same account. Using RefCell ensures that the program does not accidentally perform overlapping read/writes to the same underlying data via multiple AccountInfo structures. If a program implements its own deserialization function care should be taken to handle duplicate accounts appropriately.
- **Restrictions:**
  On-chain Rust programs support most of Rust's *libstd, libcore,* and *liballoc*, as well as many 3rd party crates.
  There are some limitations since these programs run in a resource-constrained, single-threaded environment, and must be deterministic:

  No access to:
  - Rand
  - Std::fs
  - Std::net
  - Std::os
  - Std::future
  - Std::process
  - Std::sync
  - Std::task
  - Std::thread
  - Std::time

  Limited access to:
  - Std::hash
  - Std::os

  Bincode is extremely computationally expensive in both cycles and call depth and should be avoided.

  String formatting should be avoided since it is also computationally expensive.

  No support for ***println!, print!,*** Instead the helper macro *msg!* is provided.

  Programs support a limited subset of Rust's **float** operations, if a program attempts to use a float operation that is not supported, the runtime will report an unresolved symbol error. Also, the BPF instruction set does not support **signed division**.

- The runtime enforces a limit on the number of instructions a program can execute during the processing of one instruction.
- Programs are constrained to run deterministically, so **random numbers** are not available. Sometimes a program may depend on a crate that depends itself on rand even if the program does not use any of the random number functionality. If a program depends on *rand*, the compilation will fail because there is no get-random support for Solana.
- Rust's *panic!, assert!*, and internal panic results are printed to the program logs by default.
- Use the system call ***sol_log_compute_units()*** to log a message containing the remaining number of computing units the program may consume before execution is halted.

Resources:

1. https://twitter.com/dumbcontract2
2. https://pencilflip.medium.com/solanas-token-program-explained-de0ddce29714
3. https://solana.wiki/zh-cn/docs/
4. https://docs.solana.com/