The Solana Programming Model: An Introduction to Developing on Solana

What's this Article About?

Solana's approach to decentralized computing is rooted in a simple principle: everything is stored in its own memory region, known as an account. Solana operates as a global key/value store where public keys serve as unique identifiers for their corresponding account. Accounts are the backbone of Solana because they store state; they hold everything from programs to token balances. Transactions are used to update accounts and reflect changes in state.

In this article, we explore the complexities of Solana's architecture. We start with an overview of clusters and the concept of state before discussing the role of accounts and programs as Solana's foundational components. Then, we examine how transactions enable dynamic interactions between accounts and programs.

By the end of this article, you will have a thorough understanding of Solana's programming model. You will be familiar with the architecture of clusters, the crucial role of accounts in data storage, and the process by which transactions update account data. Additionally, you will explore features unique to Solana such as its rent system and versioned transactions.

What are Solana Clusters?

At the heart of Solana's architecture areclusters - a set of validators working together to process transactions and maintain a single ledger. Solana has several distinct clusters, each serving a specific purpose:

- Localhost: a local development cluster found at the default port 8899. TheSolana Command-line Interface (CLI) comes with a built-in test validator that can be customized according to an individual developer's needs without requiring any airdrops or experiencing rate limits
- Devnet: a consequence-free sandbox environment for testing and experimenting on Solana
- Testnet: a testing ground for Solana's core contributors to trial new updates and features before they reach mainnet. It is also used as a testing environment for developers wanting to run performance tests
- Mainnet Beta: the live, permissionless cluster where real-world transactions occur. This is the "real" Solana where users, developers, token holders, and validators interact daily

Each cluster operates independently, completely unaware of one another. Transactions sent to the wrong cluster are rejected to ensure the integrity of each operational environment.

Imagine clusters as a monolithic heap of data. In computer science, a heap refers to a memory region where data can be stored and modified dynamically. It is important to note, however, that clusters do not literally use a heap data structure

. This analogy serves as a conceptual tool to aid in the understanding that clusters consist of various memory regions that can be allocated and deallocated when needed. Understanding clusters as a dynamic heap is key to understanding how data is managed, accessed, and secured within the network.

You can also think of this monolithic heap of data as a digital warehouse of sorts. Here, data is like boxes on shelves, each with unique labels and specific rules for moving them and changing their contents. This ensures a secure, orderly system where only authorized movements or changes are permitted.

Smart contracts, known as programs on Solana, are allocated their own part of the warehouse, or the heap, that they can manage. While a program can read from any part of the space in this warehouse, they need certain permissions to alter the contents of a space they do not own. The only action that is universally permitted is transferring lamports, Solana's native cryptocurrency, to any space within the warehouse.

All state lives in this heap, even programs. Each region has a program that owns it and manages it accordingly. Programs, for example, are owned by the BPFLoader, a program responsible for loading, deploying, and upgrading on-chain programs. We refer to these memory regions, our digital warehouse's boxes, as accounts.

What are Accounts?

Everything on Solana is an account. Think of accounts as containers that hold data persistently, much like files on a computer. They are the building blocks of Solana's program model used to store state (i.e., the account's balance, ownership information, whether the account holds a program, and rent information).

There are three types of accounts on Solana:

- Accounts that store data

- Accounts that store executable programs
- Accounts that store native programs

These types of accounts can be distinguished further based on their capabilities into:

- Executable accounts
- accounts that are capable of running code
- Non-executable accounts
- accounts used for data storage without the ability to execute code (because they don't hold any code!)

In the picture above, we have a few examples of executable and non-executable accounts. For executable accounts, Bubblegum is an example of a Program Account. It is a program by Metaplex used to create and manage compressed NFTs. The Vote Program is an example of a Native Program Account. It is used to create and manage accounts that track validator voting state and rewards. We'll cover the difference between Program and Native Program Accounts in the What are Programs? section. For now, it is important to know that there are different types of executable accounts on Solana.

Moreover, every non-executable account can be categorized as a Data Account. Examples of Data Accounts include:

- An Associated Token Account - an account that holds information about a specific token, its balance, and its owner (e.g., Alice has 10 USDC)
- A System Account - an account created and owned by theSystem Program
- A Stake Account - an account used to delegate tokens to validators to potentially earn rewards

Accounts Structure

Accounts are structured according to the AccountInfo

struct:

Accounts are identified by their address (key

), which is a unique 32-byte public key.

The lamports

field holds the number oflamports owned by this account. One lamport is one-billionth of a SOL, thenative token of Solana.

data

refers to the raw data byte array that is stored by this account. It can store anything from a digital asset's metadata to token balances and is modifiable by programs.

The owner

field contains the owner of this account, represented by the address of a program account. There are a few rules on account ownership:

- Only an account's owner can alter its data and withdraw lamports
- Anyone can deposit lamports into an account
- The owner of an account can transfer ownership to a new owner, provided the account's data is reset to zero

The is_signer

field is a boolean indicating if a transaction has been signed by the owner of the account in question. In other words, it tells programs involved in the transaction if the account is a signer. Being a signer means that the account holds the public key's corresponding private key and has the authority to approve the proposed transaction.

The is_writable

field is a boolean indicating whether the account's data can be modified. Solana allows transactions to specify accounts as read-only to facilitate parallel processing. While the runtime allows read-only accounts to be accessed simultaneously by different programs, it handles potential write conflicts to writable accounts using a transaction processing order. This ensures that only non-conflicting transactions are processed in parallel.

The executable

field is a boolean indicating whether an account can process instructions. Yes, this does mean that programs are stored in accounts, and we will dive into that in the next section. First, we need to cover the concept of rent.

The rent_epoch

field indicates the next epoch at which this account will owe rent. Anepoch is the number of slots for which a leader schedule is valid. Unlike traditional files in an operating system, accounts on Solana have a lifetime expressed in an amount of lamports. This idea that an account's continued existence depends on its lamport balance brings us to the concept of rent.

Rent

Rent is a storage cost incurred to keep accounts alive on Solana and ensure accounts are held in validator memory. Rent collection is assessed based on epochs, a unit of time defined by slots during which a leader schedule is valid. Here's how rent operates:

- Rent Collection
- rent is collected once an epoch. It can also be collected when an account is referenced by a transaction
- Rent Distribution
- some of the collected rent is burned, meaning it is permanently removed from circulation. The remainder is distributed to vote accounts after every slot
- Rent Payment
- if an account does not have enough lamports to pay rent then its data is removed and the account is de-allocated in a process known as garbage collection
- Rent-Exemption
- accounts can become rent-exempt if they maintain a minimum balance equivalent to two years' worth of rent payments. All new accounts must meet this rent exemption threshold, which depends on an account's size
- Rent Retrieval
- users can close an account to reclaim its remaining lamports. This allows for users to retrieve the rent stored in an account

Rent can be estimated using the getMinimumBalanceForRentExemption

RPC endpoint for a particular account size. The Test Drive simplifies this by accepting an account's data length in usize

. The Solana rent CLI sub-command can also be used to estimate the minimum amount of SOL for an account to be rent-exempt. For example, at the time of writing this article, running the command solana rent 20000

will return Rent-exempt minimum: 0.14009088 SOL

.

Addresses on Solana

There are actually two "types" of addresses on Solana. Solana uses ed25519

, an EdDSA signature scheme using SHA-512 (SHA-2) and the Curve22519 elliptic curve, for address creation. This leads to 32-byte public keys, which act as the primary address format. They can be used directly as they are not hashed.

For an address to be valid, it must be a point on the ed25519

curve. However, not all addresses need to be derived from this curve. Program Derived Addresses (PDAs) are generated off-curve, meaning they do not have a corresponding private key and cannot be used for signing. PDAs are created via the System Program

and are used when programs need to manage accounts. This is only meant as an aside to make you, the reader, aware of the different types of addresses on Solana. We'll cover PDAs in a future article.

How Do Accounts on Solana Differ From Accounts on Ethereum?

Ethereum has two primary account types: externally owned accounts (EOAs) and contract accounts. EOAs are controlled by private keys whereas contract accounts are governed by their contract code and cannot initiate transactions by themselves.

Both EOAs and contract accounts follow the same account structure:

- Balance
- every account has a balance measured in Ether
- Nonce
- for EOAs, this is the count of transactions sent from the account. For contracts, this is the number of contracts created by the account
- Storage Root
- a 256-bit hash of the root node of a Merkle Patricia Trie, which is an encoding of the storage contents of the account
- CodeHash
- the hash of the Ethereum Virtual Machine (EVM) code of the contract. This is immutable, meaning that its code doesn't change once created, although its state can. It is important to note that there are exceptions to upgrading contracts on Ethereum, such as the use of proxy patterns, however this is out of the scope of this article. For EOAs, this is a hash of an empty string since EOAs don't contain code

Solana adopts a more uniform account model where any account has the potential to be a program. The separation of code and data fosters a more efficient and flexible environment. Solana programs are stateless, interfacing with various data accounts without redundant deployments. This is particularly advantageous for decentralized finance (DeFi) applications where a user would want to interact with multiple protocols without moving assets across different programs. In contrast, Ethereum's programming model combines code and state into a single entity. This makes interactions more complex and potentially higher costs due to the gas requirements for state change.

Solana accounts used to pay rent, requiring them to hold a minimum balance to remain active. This ensures unused or underfunded accounts are eventually reclaimed by the network, thereby reducing state bloat. Recent updates have made it so there are no longer any rent-paying accounts on mainnet - accounts must be rent-exempt. In comparison, Ethereum employs gas to manage resource allocation. Under this model contract storage persists indefinitely unless it is explicitly cleared. Solana's approach offers a more predictable cost structure for state storage whereas Ethereum's costs can vary and become prohibitive during periods of network congestion.

In the following section, we will examine how Solana separates its program logic from state. In comparison to Ethereum's programming model, you'll see how this modular approach facilitates more efficient on-chain operations while providing a transparent and predictable cost structure for developers.

What are Programs?

Programs are executable accounts owned by the BPF Loader. They are executed by the Solana Runtime, which is designed to process transactions and program logic.

One of the distinctive features of Solana's programming model is the separation of code and data. Programs are stateless, meaning they do not store any state internally. Instead, all the data they need to operate on is stored in separate accounts, which are passed in to programs by reference via transactions. This design allows for a single, generic deployment of a program to interact with different accounts.

Programs on Solana have the capacity to:

- Own additional accounts
- Read from or credit other accounts
- Modify data or debit the accounts they own

There are two types of programs:

- On-chain Programs
- these are user-written programs that are deployed on Solana. They can be upgraded by their upgrade authority, which is typically the account that deployed the program
- Native Programs
- these are programs integrated into the core of Solana. They provide the fundamental functionality required for validators to operate. Native programs can only be upgraded through network-wide software updates. Common examples include the System Program, the BPF Loader Program, and the Vote Program.

Both on-chain and native programs are callable by users and other programs. The primary distinction lies in their upgrade mechanisms: on-chain programs can be upgraded by their upgrade authority whereas native programs can only be upgraded as part of cluster updates.

Solana Labs curates a select group of on-chain programs known as theSolana Program Library. This library facilitates a variety of on-chain operations, including token lending and stake pool creation. The Associated Token Account Program, for instance, sets a standard and mechanism for linking a user's wallet to their respective token accounts. Moreover, the SPL is dynamic. Programs such as Token-2022 building upon and extending the functionalities provided by theToken Program.

Program development on Solana is typically conducted in Rust with the help of Anchor, an opinionated framework that simplifies the creation of programs by reducing boilerplate and streamlining serialization and deserialization. While Rust is preferred, developers are not restricted to it - C, C++, and any language that targets the LLVM's BPF backend (i.e., a component of LLVM that allows the compilation of programs into BPF bytecode) can be used. Recent developments from Solang and Neon Labs have allowed developers to use Solidity in program development.

Programs are typically developed and tested against Localhost and Devnet before being deployed to Testnet or Mainnet Beta. Developers can deploy their program via the Solana CLI with the command solana program deploy

. The program, once compiled into an ELF shared object containing the BPF bytecode, is uploaded to the designated Solana cluster. Deployed programs live in accounts marked as executable

, with the account's address serving as the program_id

.

Initially, programs on Solana were deployed with accounts that were two times the size of the program,Solana's 1.16 update introduces support for resizable accounts to provide more flexibility and resource allocation for developers. Now, a developer is able to deploy their program with a smaller sized account, and expand its size later on.

As mentioned above, programs are considered stateless because any data they interact with is stored in separate accounts that are passed as reference. All programs have a single entry point where instruction processing takes place, which takes in a program_id

, an array of accounts, and the instruction data as a byte array. Programs are executed by the Solana Runtime once invoked by a transaction.

What are Transactions?

Transactions are the backbone of on-chain activity. They serve as the mechanism through which programs are invoked and state changes are enacted. A transaction on Solana is a bundle of instructions that tell validators what actions they should perform, on what accounts, and whether they have the necessary permissions to do so.

A transaction consists of three main parts:

- An array of accounts to read or write from
- One or more instructions
- One or more signatures

Transactions on Solana follow the Transaction struct. This provides the necessary information for the network to process and validate actions. It is defined as follows:

The signatures

field contains a set of signatures that correspond to the serialized Message

. Each signature is associated with an account key from the Message

's account_keys

list, starting with the fee payer. The fee payer is the account responsible for covering the transaction fees incurred when a transaction is processed. This is typically the account initiating the transaction. The number of required signatures is equal to the num_required_signatures

, which is defined in the message's MessageHeader

.

The message

itself is a struct of type Message

. It is defined as:

The header

of the message contains three unsigned 8-bit integers: the number of required signatures (i.e., num_required_signatures

), the number of read-only signers, and the number of read-only non-signers.

The account_keys

field lists all the account addresses involved in the transaction. Accounts requesting read-write access are first followed by read-only accounts.

recent_blockhash

is a recent blockhash, containing a 32-byte SHA-256 hash. This is required to indicate when a client last observed the ledger and acts as a lifetime for recent transactions. Validators will reject transactions with an old blockhash. Moreover, the inclusion of a recent blockhash helps to prevent duplicate transactions since any transaction that is completely identical to a previous one is rejected. If, for whatever reason, a transaction needs to be signed long before it is submitted to the network, a durable transaction nonce can be used in place of a recent blockhash to ensure that it is a unique transaction.

The instructions

field contains one or more CompiledInstruction

structs, each dictating a specific action to be taken by the network's validators.

Instructions

An instruction is a directive for a single invocation of a Solana program. It is the smallest unit of execution logic in a program and acts as the most basic operational unit on Solana. Programs interpret the data passed from an instruction and operate on the accounts specified. The Instruction

struct is defined as:

The program_id

field specifies the public key of the program to be executed. This is the address of the program that will process the instruction. The owner of the program's account, indicated by this public key, specifies the loader responsible for initializing and executing the program. The loader marks the on-chain Solana Bytecode Format (SBF) programs as executable once deployed. Solana's runtime will reject any transactions that attempt to invoke accounts that are not marked as executable.

The accounts

field lists the accounts that the instruction may read from or write to. These accounts must be supplied as AccountMeta

values. Any account whose data may be mutated by the instruction must be specified as writable or else the transaction will fail. This is because programs cannot write to accounts they do not own or have the requisite permissions to. This also applies to mutating the lamports of an account: subtracting lamports from an account not owned by the program will cause the transaction to fail while adding lamports to any account is permissible. The accounts

field can also specify accounts that are not read from or written to by the program. This is done to affect the scheduling of program execution by the runtime, but these accounts will be ignored otherwise.

data

is a general purpose vector of 8-bit unsigned integers that serves as the input that is being passed to the program. This field is crucial as it contains the encoded instructions that the program will execute.

Solana is agnostic to the format of instruction data. However, it has built-in support for serialization through bincode

and borsh

(Binary Object Representation Serializer for Hashing). Serialization is the process of converting complex data structures into a flat series of bytes that can be transmitted or stored. The choice of how data is encoded should consider the overhead of decoding since it all occurs on-chain. Borsh serialization is often preferred over bincode as it has a stable specification, a JavaScript implementation, and is generally more efficient.

Programs use helper functions to streamline the construction of supported instructions. For example, the System Program provides a helper function to construct the SystemInstruction::Assign

instruction:

This function constructs an instruction that, when processed, will change the owner of the specified account to the new owner provided.

A single transaction can contain multiple instructions, which are executed sequentially and atomically in the order they are listed. This means that either all instructions succeed or none do. This also means that the order of instructions can be critical. Programs must be hardened to safely process any possible sequence of instructions to prevent any potential exploits.

For example, during deinitialization, a program may attempt to deinitialize an account by setting its lamport balance to zero. This assumes that the Solana runtime will delete the account. This assumption is valid between transactions, however, it is invalid between instructions or Cross-Program Invocations (We will cover Cross-Program Invocations in a future article). The program should explicitly zero out the account's data to harden against this potential flaw in the deinitialization process. Otherwise, an attacker could issue a subsequent instruction to exploit the presumed deletion, such as reusing the account before the transaction completes.

What are Versioned Transactions?

Transactions on Solana use IPv6 Maximum Transmission Unit (MTU) standards to guarantee the fast and reliable transmission of data across a cluster. Solana's networking stack uses a conservative MTU size of 1280 bytes. After setting aside space for headers, 1232 bytes are available for packet data. Consequently, Solana transactions are limited to this size.

This size constraint facilitates a range of network enhancements but also restricts the complexity of operations that can be performed in a single transaction. Given that each account address occupies 32 bytes of storage, a transaction can store up to 35 accounts without any instructions. Such a restriction poses challenges for use cases requiring more than 35 signature-free accounts within a single transaction.

To address this, a new transaction format that enables support for multiple versions of transaction formats was introduced. The Solana runtime currently supports two transaction versions:

- legacy
- the original transaction format
- 0

(Version 0) - the latest transaction format that includes support for Address Lookup Tables

Version 0 was released to supportAddress Lookup Tables (ALTs). Essentially, they store account addresses in a table-like data structure on-chain. These tables are separate accounts that store account addresses and allow them to be referenced in a transaction using a 1 byte u8

index. This significantly decreases the size of a transaction as each account included only needs to use 1 byte instead of 32 bytes. ALTs are particularly useful for complex operations that involve many accounts, such as those common in DeFi applications.

The term "versioned transactions" refers to the way Solana supports both legacy and Version 0 transaction formats. This approach ensures composability while embracing runtime enhancements.

Structure of a Versioned Transaction

A VersionedTransaction

is defined as:

The signatures

field is a list of signatures from the signers of the transaction. They serve to authenticate and maintain the integrity of the transaction. The message

is the actual content of the transaction. This is encapsulated by the VersionedMessage

type, a thin enum wrapper that handles both legacy and Version 0 messages:

The message version is determined by the first bit in the serialization process. If the first bit is set, the remaining 7 bits are used to determine which Message

version is serialized starting from Version 0

. If the first bit isn't set, then all bytes are used to encode the legacy Message

format. This is due to the fact that there are two Message structs identically named, however, they are separated into different modules - legacy and v0.

A Message

represents a transaction's condensed internal format. This is used for network transmission and manipulation by the runtime. It encompasses a linear list of all the accounts used by the transaction's instructions, a MessageHeader

detailing the structure of the account array, a recent blockhash, and a compact encoding of the message's instructions. This is the structure of the v0 Message

struct:

The difference between a legacy message and a v0 message is the inclusion of the address_table_lookups

field.

Integrating the Programming Model with Solana's Transaction Flow

Solana's programming model is deeply integrated with its account and transaction systems. Here's how the concepts tie together:

- Accounts as State
- Accounts on Solana act as state containers for programs. The programming model revolves around modifying the data stored in these containers in response to instructions
- Instructions
- Programs define the logic for processing instructions that are contained within a transaction. These instructions are the actionable components that interact with account data
- Serialization and Processing
- When a transaction is serialized, the program's instructions dictate the changes to account states. The serialization process respects the program's design, whether it uses the legacy or Version 0 transaction format
- Atomicity
- Solana's programming model ensures atomic instruction processing. Programs must be designed to handle concurrent transactions safely and efficiently
- Scalability
- Solana's programming model supports scalability through features such as Address Lookup Tables (ALTs). These tables reduce a transaction's size and increases the number of accounts a transaction can reference

Solana's programming model is not just about writing code - it is about understanding how that code interacts within the broader ecosystem. Accounts are essential to this model, serving as the primary means by which data is stored and modified on the network. Transactions enable on-chain activity by telling validators what data needs to be created, updated, or deleted. A thorough understanding of these aspects is vital for developers to build applications that are optimized for performance and synergy within Solana's ecosystem.

Conclusion

Congratulations! In this article, we've navigated the complexities of Solana's system architecture, delving into the concept of clusters as monolithic heaps of data. We've discovered how this heap is organized into distinct memory regions known as accounts, forming the backbone of Solana's programming model. Accounts store everything from user tokens to the very programs that define the network's behavior, all of which are modified via transactions.

For developers, understanding Solana's approach to decentralized computing is crucial. Grasping the intricacies of accounts, programs, and transactions is necessary for building applications that fully leverage Solana's capabilities. It is about understanding a system where code is decoupled from state. This results in stateless programs that interact with data through accounts on an unprecedented scale of composability and upgradability.

For investors and casual users alike, understanding how Solana's design creates a robust, flexible, and efficient ecosystem is crucial for appreciating the platform's viability, and its capacity to foster innovative applications that are only possible on Solana.

If you've read this far anon, thank you! Ready to dive deeper? Join ourDiscord to get started programming on Solana, today.

Additional Resources / Further Reading

- Solana Command-line Guide
- Solana Wiki - Account Model
- An Introduction to Account Abstraction
- Garbage Collection
- Address Lookup Tables (ALTs)
- Versioned Transactions
- Crate solana_sdk