Chapter 1: Web3 & Solana Basics

Introduction

Welcome to the first chapter of this series where we explore the universe of **web3** and the groundbreaking innovations brought forth by **Solana**! In this chapter, we'll go on a journey through the complex landscape of blockchain technology, demistyfying the complexities that make Solana stand out in the decentralized world.

Web3, often reffered as the next generation of the internet, introduces a paradigm shift in how we interact with digital systems. Unlike its predecessors, web3 aims to create a decentralized and trustless environment, empowering users with greater control over their digital experiences[1]. At the forefront of this evolution stands Solana, a high-performance blockchain designed to revolutionize the way we engage with decentralized applications (DApps) and digital assets.

But what makes Solana so special? Its core lies in its ability to provide high throughput and low transaction costs, overcoming some of the scalability challenges faced by earlier blockchain networks[2]. As we journey through this chapter, we'll explore the complex mechanisms that make Solana stands, from cryptographic key pairs to the decentralized nature of smart contracts and Proof of History (POH).

1. Keys

Cryptographic keys are like passwords for the blockchain. They come in pairs: a **public key**, which is like your username, and a **private key**, your super-secret password. The public key is visible to everyone and helps identify you on the blockchain. Meanwhile, the private key is known only to you and is essential for making secure transactions and proving ownership of your assets.

Generating these keys in Solana is as simple as crafting your digital identity. With the help of Rust and **`solana_sdk`** crate, you can create your own key pair effortlessly. Let's consider the following Rust program:

```rust
use solana_sdk::signer::keypair::Keypair;
use solana_sdk::signature::Signer;

fn main() {
    let keypair = Keypair::new();

    let public_key = keypair.pubkey();
    let private_key = keypair.secret();

    println!("Public Key: {:?}", public_key);
```

[1] https://solana.com/news/proof-of-history
[2] https://docs.solanalabs.com/consensus/turbine-block-propagation

```
    println!("Private Key: {:?}", private_key);
}
```

The above code snippet creates your key pair. Once generated, these keys become your gateway to the decentralized world of Solana. They allow you to interact with the blockchain, sending and receiving assets securely. With your keys in hand, you hold the power to navigate the solana world and unlock its countless possibilities.

Understanding the importance of key pairs is crucial. Public keys are shared openly, allowing others to verify your identity and send assets. On the flip side, the private key must be kept safe, as it grants control over your blockchain assets. Storing it securely, like in a hardware wallet, minimizes the risk of unauthorized access.

Solana ensures security by employing **elliptic curve cryptography (Ed25519)** for key pair generation [3]. This cryptographic approach offers strong security with relatively short key lengths, making it efficient for blockchain operations. Ed25519's use ensures robust key pairs, resistant to various cryptographic threats.

Key pairs are the foundation of secure Solana transactions. Your public key is like a digital address, open for all to see, while your private key is the secret sauce, empowering you to authorize transactions and claim ownership over your assets. As you navigate the decentralized world, understanding and safeguarding key pairs become essential for a secure and seamless blockchain experience.

As we harness the power of these keys, they become the guardians of our digital identities and assets. Just like holding a precious artifact, these keys grant access to a world of decentralization, where transactions are secure, and ownership is absolute.

```
use solana_sdk::signer::keypair::Keypair;
use solana_sdk::signature::Signer;

{
    let keypair = Keypair::new();

    let public_key = keypair.pubkey();
    let private_key = keypair.secret();

    println!("Public Key: {:?}", public_key);
    println!("Private Key: {:?}", private_key);
}
```

```
Public Key: C6kh269tqFPgQnJrcvyQEThRgNxhnTyrGLsDZKoSKp7B
Private Key: SecretKey: [206, 153, 32, 204, 95, 103, 248, 6, 94, 161, 18, 95, 218, 192, 48,
```

[3] https://solana.com/news/proof-of-history

()

---

2. Wallets

In the fascinating world of blockchain, wallets are like digital treasure boxes that help you manage your online money and assets. These wallets, in web3 terms, act as magical guardians for special codes called cryptographic keys. These keys unlock the doors to decentralized wonders on the Solana blockchain, as outlined in the previous section.

Web3 wallets come in two main types: **custodial** and **non-custodial** [4]. Custodial wallets are like having a third party (like an exchange) take care of your keys for you. It's easy, but they have control. Non-custodial wallets, on the other hand, give you the full magic—complete control over your keys. This lines up with the idea that in blockchain, you should be in charge of your stuff.

Creating a wallet on Solana is like creating a special item in a game. With rust and the **Solana's sdk**, we can make **custodial** or **non-custodial** wallets. Let's explore the following Rust program:

```rust
use solana_sdk::{signature::Keypair, pubkey::Pubkey, system_instruction};

fn main() {
    // Creating a hypothetical non-custodial wallet
    let owner_keypair = Keypair::new();

    // Creating a hypothetical custodial wallet (managed by an exchange)
    let exchange_keypair = Keypair::new();

    // Doing a transaction to move money between wallets
    let transfer_instruction = system_instruction::transfer(
        &owner_keypair.pubkey(),
        &exchange_keypair.pubkey(),
        10_000_000, // Amount in lamports (SOL)
    );

    println!("Transaction Instruction: {:?}", transfer_instruction);
}
```

In the code, we see a special action, moving money between two wallets. This isn't just moving money; it's like signing a digital letter saying, "Hey! I'm in

[4] https://solana.com/news/proof-of-history

charge here." Every transaction on Solana is a little digital adventure, where your wallet's magic signature speaks for you.

Web3 wallets act like ships, helping you sail the digital seas. They let you send and receive your digital money, talk to special computer programs (smart contracts), and play in the world of Solana. Custodial wallets are like easy cruise ships, while non-custodial ones give you the wheel to steer.

Just like you keep your real-world treasures safe, you must guard your digital wallet. Tools like **hardware wallets** or having a secret backup plan are like shields against digital pirates. Following simple safety rules ensures your digital journey remains fun and secure.

As technology keeps growing, wallets will continue to be your digital sidekick. They might help you log in securely without passwords or make your digital life even more exciting. Wallets are here to stay, opening doors to new possibilities in the ever-growing world of blockchain.

```rust
use solana_sdk::{signature::Keypair, pubkey::Pubkey, system_instruction};

{
    // Creating a hypothetical non-custodial wallet
    let owner_keypair = Keypair::new();

    // Creating a hypothetical custodial wallet (managed by an exchange)
    let exchange_keypair = Keypair::new();

    // Doing a transaction to move money between wallets
    let transfer_instruction = system_instruction::transfer(
        &owner_keypair.pubkey(),
        &exchange_keypair.pubkey(),
        10_000_000, // Amount in lamports (SOL)
    );

    println!("Transaction Instruction: {:?}", transfer_instruction);
}
```

```
Transaction Instruction: Instruction { program_id: 11111111111111111111111111111111, account
```

```
()
```

3. Accounts

In Solana's world, accounts act as the guardians of digital secrets. Picture them as vaults, each holding a unique piece of information. Everything on the solana network is an account, just like Linux, everything is a file. Solana's accounts come in two different flavors: **executable** and **non-executable**, each with its own purpose and characteristics [5].

Solana's accounts are like a massive filing cabinet where the blockchain stores its secrets. Every account is identified by a special key, and it holds valuable information, including the amount of SOL (**lamports**), the owner's identity, and a data byte array for additional details. Think of it as a well-organized repository where Solana keeps track of various aspects of its decentralized world.

```rust
use solana_sdk::{account_info::AccountInfo, pubkey::Pubkey};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let owner = Pubkey::new_unique();
    let account_key = Pubkey::new_unique();
    let mut lamports = 0u64;
    let mut data = vec![0u8; 64];

    let account = AccountInfo {
        key: &account_key,
        is_signer: false,
        is_writable: false,
        owner: &owner,
        lamports: Rc::new(RefCell::new(&mut lamports)),
        data: Rc::new(RefCell::new(&mut data[..])),
        executable: false,
        rent_epoch: 0,
    };

    println!("Lamports: {:?}", account.lamports());
    println!("Owner: {:?}", account.owner);
    println!("Is Executable: {:?}", account.executable);
    println!("Data: {:?}", account.data);
}
```

In the above code snippet, we inspect the metadata of a Solana account. The `main` function reveals essential details, including the amount of SOL (lamports), the account owner, whether it's executable, and the stored data. This code snippet gives us a glimpse into Solana's filing cabinet.

[5] https://solana.com/news/proof-of-history

Solana's filing cabinet has two special sections: **executable accounts** and **non-executable accounts**. An executable account contains a program, allowing it to execute specific functions on the blockchain. Non-executable accounts, on the other hand, are data keepers. They store information without having the power to run programs. This duality ensures that the blockchain stays organized and efficient.

Solana's executable accounts are where the magic happens. They hold the smart contracts that bring the blockchain to life. Non-executable accounts, store data, ensuring that important information is securely kept. Solana's architecture relies on this organized filing system to maintain the balance between program execution and data storage.

Understanding Solana's accounts is like having a map to navigate the decentralized world. Executable accounts hold the blueprints for decentralized applications, while non-executable accounts store the history and details needed for a seamless blockchain experience. Together, they form the backbone of Solana's organized and efficient structure.

To delve deeper into Solana's account system, refer to the Solana Accounts Documentation.

Key Takeaway: - Accounts are represented with 256-bit addresses - Holds some balance of sol on the solana blockchain - Can store arbitrary data - Data storage is paid with rent - Anyone can read data on an account - Only the owner can debit sol or modify data of an account

```rust
use solana_sdk::{account_info::AccountInfo, pubkey::Pubkey};
use std::rc::Rc;
use std::cell::RefCell;

{
    let owner = Pubkey::new_unique();
    let account_key = Pubkey::new_unique();
    let mut lamports = 0u64;

    let mut data = vec![0u8; 64];
    let account = AccountInfo {
        key: &account_key,
        is_signer: false,
        is_writable: false,
        owner: &owner,
        lamports: Rc::new(RefCell::new(&mut lamports)),
        data: Rc::new(RefCell::new(&mut data[..])),
        executable: false,
        rent_epoch: 0,
    };
```

```
    println!("Lamports: {:?}", account.lamports());
    println!("Owner: {:?}", account.owner);
    println!("Is Executable: {:?}", account.executable);
    println!("Data: {:?}", account.data);
}
```

```
Lamports: 0
Owner: 1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM
Is Executable: false
Data: RefCell { value: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
()
```

---

4. Smart Contracts

Smart contracts on Solana are like the wizards of the digital world. They are self-executing, meaning they can carry out tasks on their own without needing someone to watch over them. These digital wizards are crucial for **decentralized applications (dApps)** and provide the backbone for various services on the Solana blockchain.

On the solana network, smart contracts are referred to as **Programs**. Creating a smart contract on Solana involves using `solana_program` as we have used previously. The following code snippet brings a smart contract to life:

```
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
    system_program,
    msg,
    program_error::ProgramError,
    sysvar::{rent::Rent, Sysvar, clock::Clock},
};

#[entrypoint]
fn process(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
```

```
) -> ProgramResult {
    let account_iter = &mut accounts.iter();

    let account = next_account_info(account_iter)?;

    let rent_sysvar = next_account_info(account_iter)?;
    msg!("Rent Sysvar Key: {}", rent_sysvar.owner);
    msg!("Expected Rent Sysvar ID: {}", solana_program::sysvar::id());

    if rent_sysvar.key != &solana_program::sysvar::rent::id() {
        msg!("Invalid Rent Sysvar Owner");
        return Err(ProgramError::InvalidArgument);
    }

    let rent = &Rent::from_account_info(rent_sysvar)?;

    let mut data = account.data.borrow_mut();
    msg!("Account Data Before: {:?}", data);
    for (i, &byte) in instruction_data.iter().enumerate() {
        data[i] = byte;
    }

    msg!("Account Data After: {:?}", data);

    Ok(())
}
```

Creating a functional smart contract involves several steps, including defining the program structure, handling state and storage, and executing logic. Given the complexity and length of a useful smart contract, let's explore the above simplified example.

This code snippet represents the entrypoint function `process` of a Solana smart contract. The function takes the program ID `program_id`, an array of accounts `accounts`, and instruction data as parameters `instruction_data`. It extracts the required accounts and checks if the account is rent-exempt (meaning it has enough SOL to cover storage costs). The logic inside the smart contract is a simple example that copies instruction data into the account's data.

This is a basic example, and real-world smart contracts on Solana can involve more complex logic, interaction with multiple accounts, and implementation of business-specific rules.

Solana's smart contracts are stateless; they don't remember things. To keep track of information, they use special accounts. These accounts, store data and keep the state of the smart contract. This **statelessness** ensures efficiency and consistency across the decentralized world.

Smart contracts empower us to interact with the blockchain in new and exciting ways. They enable decentralized exchanges, gaming platforms, and much more. The magic lies in their ability to execute actions automatically, removing the need for intermediaries and enhancing the trustworthiness of digital interactions.

Smart contracts are open for all to see. Their transparency ensures that everyone can verify the rules and actions encoded within. This openness aligns with the decentralized philosophy, fostering trust and collaboration in the digital world.

As more smart contracts join the Solana world, the possibilities expand. The blockchain becomes a canvas for us to create, innovate, and explore. With each new smart contract, Solana's decentralized community writes a chapter in the ongoing magical saga of blockchain evolution.

For a deeper look into Solana's smart contract magic, you can refer to the Solana Smart Contracts Documentation and unlock the secrets to creating your smart contracts.

```rust
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
    system_program,
    msg,
    program_error::ProgramError,
    sysvar::{rent::Rent, Sysvar, clock::Clock},
};
use std::rc::Rc;
use std::cell::RefCell;
use borsh::{BorshSerialize, BorshDeserialize, BorshSchema};

#[derive(Debug, Default, Copy, Clone, PartialEq, BorshDeserialize, BorshSerialize, BorshSche
struct RentData {
    pub lamports_per_byte_year: u64,
    pub exemption_threshold: u64,
    pub burn_percent: u8,
}

fn process(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let account_iter = &mut accounts.iter();

    let account = next_account_info(account_iter)?;
```

```
    let rent_sysvar = next_account_info(account_iter)?;
    msg!("Rent Sysvar Key: {}", rent_sysvar.owner);
    msg!("Expected Rent Sysvar ID: {}", solana_program::sysvar::id());

    if rent_sysvar.key != &solana_program::sysvar::rent::id() {
        msg!("Invalid Rent Sysvar Owner");
        return Err(ProgramError::InvalidArgument);
    }

    let rent = &Rent::from_account_info(rent_sysvar)?;

    let mut data = account.data.borrow_mut();
    msg!("Account Data Before: {:?}", data);
    for (i, &byte) in instruction_data.iter().enumerate() {
        data[i] = byte;
    }

    msg!("Account Data After: {:?}", data);

    Ok(())
}

fn main() -> ProgramResult {
    let program_id = solana_program::sysvar::id();
    let account_key = Pubkey::new_unique();
    let mut lamports = 0u64;
    let mut lamports1 = 0u64;
    let mut data = vec![0u8; 64];

    let dummy_rent_data = RentData {
        lamports_per_byte_year: 42,
        exemption_threshold: 1000,
        burn_percent: 5,
    };

    let mut serialized_rent_data = borsh::try_to_vec_with_schema(&dummy_rent_data).expect("

    let rent_sysvar_key = solana_program::sysvar::rent::id();
    let rent_sysvar_account = AccountInfo {
        key: &rent_sysvar_key,
        is_signer: false,
        is_writable: false,
        lamports: Rc::new(RefCell::new(&mut lamports1)),
        data: Rc::new(RefCell::new(&mut serialized_rent_data)),
        owner: &system_program::id(),
```

```
        executable: false,
        rent_epoch: 0,
    };

    let account_info = AccountInfo {
        key: &account_key,
        is_signer: false,
        is_writable: false,
        owner: &program_id,
        lamports: Rc::new(RefCell::new(&mut lamports)),
        data: Rc::new(RefCell::new(&mut data[..])),
        executable: false,
        rent_epoch: 0,
    };

    let accounts = vec![account_info, rent_sysvar_account];
    let instruction_data = [1, 2, 3, 4];

    let result = process(&program_id, &accounts, &instruction_data)?;

    Ok(())
}

main()
```

```
Rent Sysvar Key: 11111111111111111111111111111111
Expected Rent Sysvar ID: Sysvar1111111111111111111111111111111111111
Account Data Before: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
Account Data After: [1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
Ok(())
```

---

5. Instructions & Transactions

```
+-----------------------+
|   Solana Transaction |
| +------------------+ |
| | Instruction 1    | |
| +------------------+ |
| +------------------+ |
```

```
| | Instruction 2    | |
| +------------------+ |
| +------------------+ |
| | Instruction 3    | |
| +------------------+ |
+----------------------+
```

In Solana, the execution of smart contracts and the manipulation of the blockchain's state are carried out through a combination of instructions and transactions.

Transactions in Solana are atomic, meaning they either fully succeed or completely fail. This ensures the integrity of the blockchain's state, preventing incomplete or inconsistent changes. The atomic nature of transactions is crucial for maintaining the reliability of the decentralized system.

Programs in Solana are invoked through instructions, and these instructions are encapsulated within transactions. An instruction can be viewed as a set of operations or tasks that a program is designed to execute. Transactions act as the carriers of these instructions, facilitating the interaction between users and the decentralized applications (DApps) on the Solana blockchain.

Let's illustrate this with a simple code snippet. In this example, we'll create a basic transaction that transfers SOL tokens from one account to another:

```rust
use solana_sdk::{signature::Keypair, system_instruction};
use solana_program::instruction::Instruction;
use solana_client::rpc_client::RpcClient;
use solana_sdk::signature::Signer;
use solana_sdk::transaction::Transaction;

{
    let sender_keypair = Keypair::new();
    let receiver_keypair = Keypair::new();

    let rpc_client = RpcClient::new("https://api.devnet.solana.com".to_string());

    let recent_blockhash = rpc_client.get_latest_blockhash()?;

    let instruction = system_instruction::transfer(&sender_keypair.pubkey(), &receiver_keyp

    let mut transaction = Transaction::new_signed_with_payer(
        &[instruction.clone()],
        Some(&sender_keypair.pubkey()),
        &[&sender_keypair],
        recent_blockhash,
    );
```

```
}
```

This code exemplifies the construction of a transaction that includes a system transfer instruction transferring 100 SOL from the sender to the receiver.

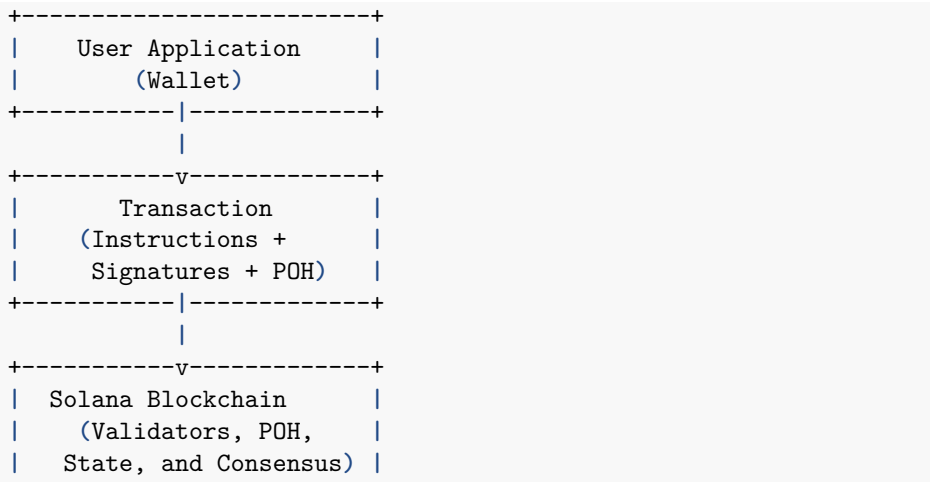
For a transaction to be considered valid, it must be signed by the relevant parties involved. This ensures that only authorized users can initiate changes to the blockchain's state. The signature mechanism adds a layer of security, preventing unauthorized access and malicious activities.

Solana employs a Proof-of-History (POH) mechanism to order transactions and achieve consensus among validators. The POH acts as a historical record, providing a chronological sequence of events. This enables validators to agree on the order of transactions, maintaining the integrity and consistency of the blockchain.

A transaction goes through several stages in its lifecycle, from creation to confirmation. Validators receive transactions, verify signatures, execute instructions, and finally reach a consensus on the state changes. This structured lifecycle ensures that the decentralized network operates smoothly and reliably.

Solana's design promotes parallel execution of transactions, contributing to its remarkable speed. As transactions are stateless and can be processed independently, the blockchain can handle a high throughput of operations simultaneously. This parallelism enhances the efficiency of real-time and high-performance applications built on the Solana blockchain.

Solana's architecture enables low transaction fees due to its efficient handling of transactions. The decentralized nature of the network, coupled with parallel execution, minimizes congestion and optimizes resource utilization. This results in cost-effective transactions, making Solana an attractive platform for developers and users alike.



```
+------------------------+
```

This diagram illustrates how a user application interacts with the Solana blockchain through a transaction containing instructions, signatures, and the Proof-of-History.

Instructions and transactions form the backbone of Solana's functionality, facilitating the seamless execution of smart contracts and state changes. The combination of atomic transactions, program invocation through instructions, and the secure signature mechanism ensures the reliability and security of the Solana blockchain. This robust architecture, coupled with parallel execution and low transaction fees, positions Solana as a high-performance blockchain suitable for real-time applications.

Key Takeaway - Programs, aka smart contracts, are invoked by instructions. - Instructions are sent via transactions. - Transactions are atomic. - Transactions must be signed.

```rust
use solana_sdk::{signature::Keypair, system_instruction};
use solana_program::{instruction::Instruction, system_program};
use solana_client::rpc_client::RpcClient;
use solana_sdk::signature::Signer;
use solana_sdk::transaction::Transaction;

{
    let sender_keypair = Keypair::new();
    let receiver_keypair = Keypair::new();

    let rpc_client = RpcClient::new("https://api.devnet.solana.com".to_string());

    let recent_blockhash = rpc_client.get_latest_blockhash()?;

    let instruction = system_instruction::transfer(&sender_keypair.pubkey(), &receiver_keyp

    // https://docs.rs/solana-sdk/latest/solana_sdk/transaction/struct.Transaction.html#meth
    let mut transaction = Transaction::new_signed_with_payer( // Create a transaction + sig
        &[instruction.clone()],
        Some(&sender_keypair.pubkey()),
        &[&sender_keypair],
        recent_blockhash,
    );

    // TODO: send the transaction
    // rpc_client.send_and_confirm_transaction(&transaction)?;

    println!("Instruction: {:?}\n", instruction.clone());
    println!("Transaction: {:?}", transaction);
```

```
}
```

```
Instruction: Instruction { program_id: 11111111111111111111111111111111, accounts: [AccountM

Transaction: Transaction { signatures: [124C4i3P9YguhSoFsfUxhvjMYNJFy62Lzhn4mXinPM8UFmiLE9Bt
```

```
()
```

```
use solana_program::system_instruction::{create_account, transfer};
use solana_sdk::{signature::Keypair, system_instruction};
use solana_program::{instruction::{AccountMeta, Instruction}, pubkey::Pubkey, message::v0::
    entrypoint::ProgramResult};
use solana_client::rpc_client::RpcClient;
use solana_sdk::{signature::Signer,
     address_lookup_table_account::AddressLookupTableAccount,};
use solana_sdk::transaction::Transaction;
use std::collections::HashMap;
use solana_program::address_lookup_table::{self, state::{AddressLookupTable}};

fn explorer_url(params: HashMap<&str, String>) -> String {
    if let Some(address) = params.get("address") {
        return format!("https://explorer.solana.com/address/{}?cluster={}", address, params
    } else if let Some(tx_signature) = params.get("txSignature") {
        return format!("https://explorer.solana.com/tx/{}?cluster={}", tx_signature, params
    } else {
        return "[unknown]".to_string();
    }
}

{
    let payer = Keypair::new();
    let test_wallet = Keypair::new();

    let connection = RpcClient::new("https://api.testnet.solana.com".to_string());

    println!("Payer address: {:?}", payer.pubkey());
    println!("Test wallet address: {:?}", test_wallet.pubkey());

    let space = 0;

    let balance_for_rent_exemption = connection
        .get_minimum_balance_for_rent_exemption(space)?;
```

```rust
// create an instruction to create an account
let create_test_account_ix = create_account(
    &payer.pubkey(),
    &test_wallet.pubkey(),
    balance_for_rent_exemption + 2_000_000,
    space as u64,
    &system_program::id(),
);

// create an instruction to transfer lamports to the test wallet
let transfer_to_test_wallet_ix = transfer(
    &payer.pubkey(),
    &test_wallet.pubkey(),
    balance_for_rent_exemption + 100_000,
);

// get the latest recent blockhash
let recent_blockhash = connection.get_latest_blockhash()?;

// create a transaction message
let instructions = vec![ // transactions are executed atomically
    create_test_account_ix,
    transfer_to_test_wallet_ix.clone(),
    transfer_to_test_wallet_ix.clone(),
    transfer_to_test_wallet_ix.clone(),
];

// TODO: Use a public wallet address on devnet or mainnet
// let raw_account = connection.get_account(&payer.pubkey())?;
// let address_lookup_table = AddressLookupTable::deserialize(&raw_account.data)?;
// let address_lookup_table_account = AddressLookupTableAccount {
//     key: payer.pubkey(),
//     addresses: address_lookup_table.addresses.to_vec(),
// };

// let message = Message::try_compile(
//         &payer.pubkey(),
//         &instructions,
//         &[address_lookup_table_account],
//         recent_blockhash,
//     )?;

// create a transaction and sign it
let mut transaction = Transaction::new_with_payer(&instructions, Some(&payer.pubkey()));
transaction.partial_sign(&[&payer, &test_wallet], recent_blockhash);
```

```
    // TODO: send the transaction
    // let sig = connection.send_and_confirm_transaction(&transaction)?;

    let mut params = HashMap::new();

    // Example with address
    params.insert("address", payer.pubkey().to_string());
    params.insert("cluster", "testnet".to_string());
    // params.insert("txSignature", sig.to_string());

    println!("Transaction completed.");
    println!("Explorer URL: {}", explorer_url(params));
}
```

```
Payer address: Hq5cefHZut7EnpjnbKCXg6tPGhqUZbyJU7erDZ2cfyFQ
Test wallet address: 8K5AKYzcKqvUaxYL2UE89aWF9pJoANA745VYPsELn5Up
Transaction completed.
Explorer URL: https://explorer.solana.com/address/Hq5cefHZut7EnpjnbKCXg6tPGhqUZbyJU7erDZ2cfy
```

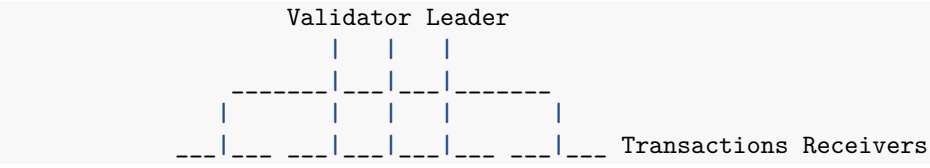
```
()
```

---

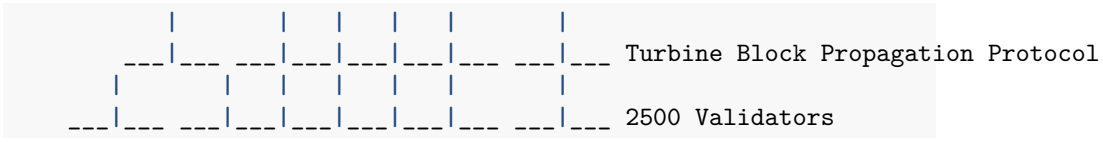
6. Solana Architecture

The Solana architecture represents a groundbreaking design in the blockchain domain, having features that set it apart from traditional platforms. At its core, Solana operates with a validator leader supervising the reception of transactions across the entire blockchain [6]. This pivotal figure efficiently organizes and packs these transactions into blocks. The subsequent distribution of these blocks to all 2500 validators is facilitated by Turbine, a key component responsible for block propagation within the Solana protocol [7].

To better understand this orchestration, envision a centralized hub, the validator leader, managing the influx of transactions from various sources.



[6] https://solana.com/news/proof-of-history
[7] https://docs.solanalabs.com/consensus/turbine-block-propagation



In this visual representation, the validator leader serves as the central point, connecting to numerous sources and efficiently organizing transactions for further processing. The complex web of connections symbolizes the vast network of validators involved in the Solana ecosystem.

One of Solana's standout features lies in its ability to execute transactions in parallel. This is made possible by the stateless nature of each individual transaction.

```
Transaction 1:  |-----|
Transaction 2:        |-----|
Transaction 3:  |-----------|
Transaction 4:            |-----|
Transaction 5:  |---------------|
Transaction 6:              |-----|
```

Each horizontal line represents the execution timeline of a transaction, and the gaps between them signify concurrent processing. This parallel execution contributes significantly to Solana's remarkable speed, a characteristic accentuated by the use of Proof of History (POH).

Proof of History (POH) is a foundational element in Solana's architecture, contributing to its exceptional speed. Think of POH as a synchronized clock, creating a chronological order for transactions. This orderly sequence is crucial for maintaining the integrity of the blockchain while enabling fast confirmation times.

```
POH Clock    :  Tick  Tock  Tick  Tock  Tick  Tock
Transaction 1:  |-----|
Transaction 2:        |-----|
Transaction 3:                |-----|
Transaction 4:                    |-----|
Transaction 5:                        |-----|
Transaction 6:                                |-----|
```

In this representation, the POH clock sets the pace for transaction execution, ensuring an orderly progression. The synchronized nature of transactions enhances security and efficiency.

Devnet is a playground for developers to experiment, **Testnet** is a space for stress testing, and **Mainnet Beta** is where real transactions unfold. Each cluster is like a neighborhood, contributing to the overall functionality and growth of the Solana ecosystem

Summing up, Solana's architecture offers a plenty of advantages: fast confirmation times, low transaction fees, and the ability to execute programs in parallel. This design makes Solana an ideal choice for real-time, high-performance applications. The interconnected validators, parallel transaction execution, and the orchestration by the validator leader, all supported by POH, contribute to Solana's position as a competitor in the blockchain space.

Key Takeaway:

- Fast Confirmation time.
- Low Transaction fees.
- Executes programs in parallel.
- Ideal for high performance applications.

For a deeper exploration of Solana's architecture, refer to the **Solana Clusters Documentation**. This guide provides a detailed map of Solana's districts, offering insights into Devnet, Testnet, and the bustling Mainnet Beta.

---

7. JSON RPC

In the digital conversation between users and the Solana blockchain, **JSON RPC (Remote Procedure Call)** serves as the universal language. It's like a bridge that allows users and developers to communicate with Solana using simple, human-readable messages. JSON RPC acts as the interpreter, translating requests and responses between the crypto city residents and the decentralized infrastructure.

JSON RPC communicates through a set of standard messages, usually in JSON format. It's like chatting with the blockchain in plain English. These messages can be requests for information, transaction instructions, or queries about the state of the blockchain. JSON RPC ensures that users can interact with Solana in a way that feels familiar and accessible.

To engage in conversation with Solana, we can utilize three public endpoints, each serving a specific Solana district:

1. **api.devnet.solana.com:** The developer's hub, where experimentation and testing take place in the Devnet district.
2. **api.testnet.solana.com:** The stress-testing ground of Testnet, where the blockchain proves its resilience.
3. **mainnet-beta.solana.com:** The Mainnet Beta, the live district where real transactions unfold and businesses thrive.

To initiate a conversation with Solana, we send HTTP requests to one of the public endpoints using tools like `curl` or from a rust program. These requests carry specific instructions or queries, and Solana responds in kind. It's like asking questions and receiving answers in the language both humans and the blockchain understand.

```
curl https://api.devnet.solana.com/ \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":1,"method":"getRecentBlockhash","params":[]}'
```

JSON RPC is the gateway to Solana's wealth of information. We can inquire about transaction details, account balances, or the latest block on the blockchain. It's a tool that empowers us to stay informed and to build applications that interact seamlessly with Solana's decentralized infrastructure.

While JSON RPC opens up a world of possibilities, we should be mindful of rate limits imposed on public endpoints. These limits prevent abuse and ensure fair usage. As developers creating production applications, we should consider dedicated infrastructure or validators to avoid potential issues with rate limits on public endpoints.

For more information about the Solana's JSON RPC language, the Solana JSON RPC Documentation serves as a comprehensive guide. Explore the nuances of JSON RPC and uncover the complexities of conversing with Solana in the language of the blockchain.

---

8. POH

In the world of blockchain, achieving **consensus**, ensuring that all participants agree on the state of the network, is crutial. Solana's innovative approach to consensus, known as **Proof of History (POH)**, stands as its secret sauce, revolutionizing how decentralized networks reach agreement without sacrificing speed or security.

At its core, POH is a mechanism for establishing a chronological order of events on the Solana blockchain. It's like a timestamping service that assigns a unique and verifiable timestamp to each transaction or event, ensuring that the sequence of actions is indisputable. POH acts as the backbone of Solana's consensus algorithm, facilitating efficient and trustless validation of transactions.

POH operates by leveraging a **verifiable delay function (VDF)**, a cryptographic primitive that generates a sequence of outputs with a known delay. In Solana's implementation, this VDF produces a stream of timestamps, each linked to the previous one in a cryptographically secure manner [8]. Validators use these timestamps to order transactions and reach consensus on the state of the blockchain.

One of the key advantages of POH is its ability to enhance the efficiency and scalability of the Solana network. By establishing a reliable and immutable ordering of events, POH eliminates the need for costly and time-consuming

[8] https://solana.com/news/proof-of-history

consensus mechanisms like traditional **proof-of-work (PoW)** or **proof-of-stake (PoS)**. This enables Solana to process thousands of transactions per second while maintaining decentralization and security.

POH unlocks a world of possibilities for developers and entrepreneurs seeking to build decentralized applications on Solana. Its efficient and scalable consensus mechanism lays the foundation for a wide range of use cases, from **decentralized finance (DeFi)** and **non-fungible tokens (NFTs)** to gaming and **distributed computing**.

As Solana continues to grow and evolve, POH remains at the forefront of its technological innovation. By harnessing the power of time and cryptography, Solana's consensus mechanism sets new standards for speed, scalability, and security in the blockchain industry. With POH, Solana paves the way for a decentralized future where transactions are fast, reliable, and inclusive.

For more information about Solana's POH consensus mechanism, the **Solana Proof of History Documentation** offers a comprehensive guide. Dive into the inner workings of POH and uncover the secrets behind Solana's groundbreaking approach to consensus.

Conclusion

In our journey through the world of Solana and Web3, we've uncovered the mysteries of blockchain, explored the wonders of decentralized finance, and delved into the innovative technologies driving the future of decentralized applications. As we continue to navigate this digital world, let's embrace the magic of Solana and the boundless possibilities it offers for decentralized innovation and collaboration.