LocalSolana

LocalSolana Litepaper V0.1

October 2024

Abstract

LocalSolana is a decentralized peer-to-peer (P2P) protocol for people to exchange crypto assets like Solana and USDT (Tether) for fiat money without centralized counterparty risk through escrow smart contracts. The protocol allows people from any country to exchange assets on the Solana blockchain network and with any fiat money payment network or bank transfer method.

The protocol is governed by a decentralized autonomous organization (DAO) that also arbitrates disputes between parties. Stakeholders coordinate through the protocol token, but LocalSolana users will not need to interact with the protocol token in order to complete an exchange. The dispute mechanism relies on game theory incentives for arbitrators to rule cases truthfully and to prevent bad actors.

Introduction

Crypto adoption has been accelerating throughout the last decade, especially in developing countries experiencing sustained inflation and depreciation of their fiat currencies against the US dollar. This has been seen in the explosive growth of US dollar-pegged stablecoins which have increased 1000% in market capitalization over the past five years[1]. However one of the largest challenges remaining in crypto asset adoption in emerging markets has been onboarding people in developing countries onto blockchain networks. Traditional onramp methods through card networks or bank transfer methods often don't exist or are plagued by high fees and failure rates that severely hinders crypto adoption.

This has led to the majority of crypto users in emerging markets exchanging crypto assets and fiat money through peer-to-peer (P2P) platforms. These platforms are largely offered by centralized exchanges which have always been the subject of heists and fraud leading to the loss of customer funds. Continued devaluation in local currencies has also forced governments to block access to centralized exchanges like Binance and their P2P exchange with it. This has pushed more users to adopt self-custody wallets and onchain protocols as an alternative, with many new users downloading them as their first crypto application. A decentralized P2P protocol needs to be available to onboard users into these applications without the need to go through a centralized exchange.

History of P2P Platforms

Since the emergence of cryptocurrency, some early adopters have been looking for alternative options to centralized exchange trading. Centralized exchanges, which keep records of customer identities and balances in a database and facilitate the transfer of fiat money to and from cryptocurrency, are seen by some as contradictory to the decentralized nature of cryptocurrencies.

One alternative to centralized exchange trading is over-the-counter (OTC) trading. In OTC trading, two parties exchange directly with one another without the oversight of an exchange. This type of trading offers flexibility as there are no limitations on what can be exchanged for cryptocurrency and the transactions are peer-to-peer, without the need for a financial intermediary. Additionally, fees for larger transactions can be lower than exchange fees.

[1] Coingecko - Top Stablecoins by Marketcap. https://www.coingecko.com/en/categories/stablecoins

However, OTC trading also carries a significant risk, as both parties may be uncertain about the other's ability to fulfill their end of the trade and not abscond with the money or cryptocurrency.

An early innovation in OTC trading was the creation of an online "web of trust" platform, like the one known as #bitcoin-otc. On this platform, traders can leave public ratings for other traders, and a website will calculate each user's cumulative trust score. If a user has a high rating, they are more reputable and less likely to scam. While this system was useful, it was not perfect as it had some loopholes, like high-rated traders committing heists and scammers manipulating their ratings.

In 2012, LocalBitcoins introduced an escrow system as a solution. Sellers of Bitcoin could now transfer their BTC to LocalBitcoins first and release it from escrow once the buyer confirms payment. If the buyer fails to send the money, the seller can request LocalBitcoins to return the coins. On the other hand, if the seller refuses to release the escrow, the buyer can request LocalBitcoins to review their proof of payment and force the seller to release the Bitcoins. This system combined with the reputation system worked well but the problem was that LocalBitcoins became the new intermediary. If LocalBitcoins were hacked, went offline or its founders decided to abscond, all the Bitcoins in escrow could be lost. There was a huge amount of trust placed in the centralized escrow system and users no longer held the keys to their Bitcoin; instead, they were trusting LocalBitcoins to hold Bitcoins for them, similar to a bank or an exchange. This moved the counter-party risk but it was not completely eliminated.

The centralization of these platforms has also made them easy to shutdown. Since 2022, LocalBitcoins, LocalCryptos (formerly LocalEthereum) and LocalMonero have all shutdown operations. Binance P2P and other P2P exchanges offered by centralized exchanges have been blocked by authorities in Nigeria, Venezuela and other countries experiencing sustained currency depreciation and capital flight.

The launch of Ethereum and Solana made it possible to program money and as a result, many ideas not possible with Bitcoin could now be achieved through smart contracts. Building on the concepts of a web of trust and escrow system, LocalSolana is developing a P2P OTC trading platform that keeps users in control of their cryptocurrency at all times.

Disadvantages of Existing Onramp Solutions

In order to understand the need for LocalSolana, a look at the current state of crypto on/off ramps is necessary. For self-custody wallets the most popular onramp providers are MoonPay, Ramp Network and Transak. Though existing onramps have been a popular addition to self-custody wallets over the past years, their success is largely limited to developed countries for the following reasons.

Limited Availability of Payment Methods

Unlike developed countries, alternative payment networks are largely the norm in developing countries. For example the instant-transfer networks PIX and UPI in Brazil and India, are both the dominant payment methods for consumers in each respective country. Instant cashless transfers are being adopted rapidly in Nigeria. However few onramp providers offer these payment methods, and

when they do the fees are exorbitantly high, often near 3% of the total transactions size[2]. Because LocalSolana connects two parties, any payment method to transfer fiat currency can be used and the fees are a tenth of existing onramp solutions.

Payment Destinations are a Central Point of Failure

Even when alternative payment methods are available, payments to the provider entities from a crypto buyer's bank are often blocked. As local currencies devalue, it is not in the bank's self interest to have their depositors move funds to crypto. In the case of P2P transactions, professional merchants who both buy and sell crypto on exchanges have tens or even hundreds of bank accounts which make it harder for banks to flag the transactions. Only by decentralizing the exchanges between crypto assets and fiat money through a P2P platform are people able to reliably onboard into the crypto economy.

Limited Availability of Countries

Some of the largest crypto adopting countries are not served by leading onramp providers today. For example Pakistan with 220 million people and ranked 6th on Chainalysis' Global Crypto Index[3] isn't available with market leaders like Moonpay, Transak and Ramp Network[4]. Regulatory uncertainty and the need for centralized entities to find banking partners in each country make it difficult for centralized on/off ramp providers to provide services in developing countries where crypto is popular. Because LocalSolana doesn't require banking partners, any country can be supported globally by the protocol. However the front-end served by any legal entities may require geo-blocking if required in the jurisdictions in which we are domiciled.

High Fees

Most on/off ramps today charge transaction fees of 3% on average. There are also minimum fees per transaction on average between $4-5. In developing countries where the average transaction can be $50 for new users, this results in a major barrier to entry as users are required to pay an effective transaction fee of 10% along with any additional taxes and fees required by their government or bank. The high transaction fee removes the economic incentive for buyers to switch their fiat money for crypto assets. LocalSolana doesn't charge any fees to crypto buyers and only takes a 0.5% fee from sellers.

The Disadvantages of P2P Platforms

The leading P2P platforms today are offered by centralized exchanges including but not limited to Binance, Bybit, Kucoin and OKX.

---

[2] Moonpay Help Center - What fees do you charge?
https://support.moonpay.com/hc/en-gb/articles/360011930117-What-fees-do-you-charge-
[3] Chainalysis - 2022 Global Crypto Adoption Index.
https://blog.chainalysis.com/reports/2022-global-crypto-adoption-index/
[4] Moonpay - What are our non-supported countries, states and territories for on-ramp product?
https://support.moonpay.com/hc/en-gb/articles/6557330712721-What-are-our-non-supported-countries-states-and-territories-for-on-ramp-product-

Counterparty Risk

The collapse of FTX in late 2022 showed the risks with centralized exchanges. As a P2P merchant selling crypto assets, you need to keep all your available funds on the centralized exchange in order to exchange with other users. In prior years, centralized exchanges have been the victim of hacks leading to the loss of user funds. Phishing attacks are also another risk when funds are kept on a centralized exchange.

As a user you are also required to receive the funds on a centralized exchange resulting in the same risk that merchants are exposed to. Though you can later move your purchased crypto assets to a self-custody wallet, the extra step is cumbersome for inexperienced users. With LocalSolana users can directly fund their self-custody wallet without interacting with a centralized middleman.

User Flow Location

In 2024, most users that are entering crypto for the first time are downloading a mobile self-custody wallet like Phantom, Backpack or Solflare first rather than Binance. This is due in a large part due to the popularity of meme coins on Solana, however it isn't the sole reason as volumes for stablecoin payments and DeFi protocols are also seeing significant growth. The problem with self-custodial wallets and onchain applications today is that they don't provide a P2P onramp option like Binance and other centralized exchanges do. This leaves users in emerging markets who require this option to still have to access a centralized exchange to buy crypto through their P2P option in order to use these wallets or onchain products. This isn't an obvious step for many users and drives them away from crypto products all together. Even if they aren't driven away, having to navigate numerous steps through an external product in order to use the intended one isn't conducive to mass adoption.

Privacy

Another disadvantage of P2P platforms offered by centralized exchanges is the need to verify your identity. Centralized exchanges holding all customer identity and financial records create a honeypot for hackers looking to exploit this information. In 2018, popular Brazilian crypto investment platform Atlas Quantum was the victim of a hack that resulted in 261,000 customers' records being leaked including emails, names and physical addresses[5]. In late 2022, popular American exchange Gemini leaked 5.7 million customer emails and phone numbers[6]. For centralized exchange P2P platforms, personal details are used to provide a reputation signal to other users that a user is legitimate along with preventing sybil attacks. In LocalSolana, decentralized identity and proof of humanity solutions will be utilized in order to achieve the same results while allowing users to preserve their privacy. Users can verify their identity with a provider who provides an attestation that they're a verified person not under any sanction.

---

[5] Cointelegraph - Brazilian Crypto Platform Atlas Quantum Reveals Data Breach Affecting 260K Customers. https://cointelegraph.com/news/brazilian-crypto-platform-atlas-quantum-reveals-data-breach-affecting-260k-customers

[6] Cointelegraph - 'Third-party incident' impacted Gemini with 5.7 million emails leaked. https://cointelegraph.com/news/gemini-allegedly-suffers-data-breach-5-7-million-emails-leaked

How Trading on LocalSolana Works

Definitions

We need to define names for participants and actions in order to explain how people use LocalSolana.

Seller: Person posting an advertisement that they want to sell crypto and exchanging said crypto assets for fiat money

Buyer: Person responding to advertisement and purchasing crypto assets with fiat money

Trade: Exchange between merchant and buyer

An LocalSolana Trade

As LocalSolana never holds funds, the experience differs from centralized exchanges and escrow providers. LocalSolana doesn't take deposits, process withdrawals or hold fiat or crypto. Exchanging fiat money for crypto assets on LocalSolana looks like the scenarios below:

1. Alice (The Seller) posts an ad to sell USDT, posting a message to a LocalSolana contract accessible and public for anyone else to find. The ad states the available USDT to sell, minimum and maximum trade sizes, acceptable payment methods, and any price rules. For example: Alice is selling 100 USDT, with a minimum sale of 10 USDT and maximum of 100 USDT, she is willing to accept ₦1700 per USDT.

2. Bob (The Buyer) responds to the ad to exchange USDT for Naira ₦170000. When Bob responds to the offer, Alice is sent a notification to escrow the funds for the trade. Bob wants to purchase 100 USDT for ₦170000.

3. Alice accepts the offer and in a single transaction deploys a contract that escrows her funds. Once funds are escrowed, Bob receives a notification revealing Alice's payment details and instructions on how to make the payment. As soon as the crypto is escrowed a payment window set by the seller begins. If payment is not made within that time the crypto escrowed in the contract can be released back to the seller.

4. Bob confirms when he makes the payment and signs a transaction so the escrowed funds cannot be withdrawn back to the seller without arbitration. A notification is sent to Alice that payment has been made and to release the escrowed USDT to Bob.

5. Upon receipt of ₦170000, Alice signs a transaction to release the escrowed funds from the contract and Bob receives 100 USDT.

An LocalSolana Dispute

After crypto has been escrowed either party can raise a dispute and assign a third-party arbitrator to make a resolution. In order to initiate a dispute both parties must pay a small fee to initiate arbitration. At this time they can also submit evidence such as proof of payment to help the arbitrators make a decision.

At launch the arbitrator will be a multisig controlled by the LocalSolana team. As the protocol grows arbitration will be decentralized to the community and arbitration will be done by protocol token stakers.

LocalSolana Architecture

User Accounts

Accounts in LocalSolana are represented by each user's Solana address. Users can login to LocalSolana through a compatible wallet like Phantom, Backpack or Solflare. Users can seamlessly access LocalSolana through the mobile dapp browser feature common in most wallets. LocalSolana will also develop and maintain SDKs so any wallet can integrate natively with the protocol. Users are not required to share any other piece of identity, though proof-of-humanity and other decentralized identity solutions will form an important reputation layer. Protocol token stakers may be able to validate other reputation information like bank account details in return for fees.

LocalSolana Account Creation

Most peer-to-peer trades in emerging markets today happen through US dollar pegged stablecoins, primarily with USDT (Tether). Though LocalSolana is a self-custody platform, it is built to compete with centralized platforms in terms of user experience. LocalSolana is built on Solana as it has proven itself as the fastest and most scalable Layer 1 blockchain live today. Decentralized exchange and transaction volumes have consistently eclipsed Ethereum and all its Layer 2 solutions combined, while maintaining a single execution and settlement layer. With lower gas-fees, LocalSolana utilises relayers to cover gas fees and rent for users so they can buy crypto even if they have no SOL in their wallet.

Users create a LocalSolana account known as a Program-Derived Address through the LocalSolana protocol. The LocalSolana account can store SOL and other SPL tokens for the user that can be automatically moved into Escrow Accounts when a trade is initiated.

Escrow Creation

Whenever a trade is initiated the LocalSolana protocol programmatically creates an individual escrow account where SOL and other SPL tokens are held throughout the trade. Only the buyer and seller can interact with the escrow account unless a dispute is started and then the LocalSolana arbitration account will also be able to interact with the escrow account. After a trade is completed or canceled the LocalSolana protocol closes the account and claims back the SOL paid for rent.

Reputation

At launch reputation will be centralized and based on core metrics including:

Wallet Age: How long the wallet address has been making transactions on Solana

Account Age: How long the account has been created for

Trade Count: How many successful trades have been completed through LocalSolana

Volume: Historical volume traded denominated in US dollars

Positive Feedback Percentage: After every trade each user will have the option to rate their trade as Good or Bad

LocalSolana intends to decentralize reputation over time so any front-end application can interact with all aspects of the protocol. Other metrics may be tracked to give a holistic view for each traders counterparty. These metrics may include a user's activity on decentralized finance (DeFi) protocols and on other popular protocols on the Solana blockchain.

Communication Between Parties

LocalSolana strives to maintain the highest levels of privacy between parties. In the future, a wallet-to-wallet messaging protocol will be adopted so messaging between parties is end-to-end encrypted. Users can opt to receive trade notifications via email or PGP.

LocalSolana Contracts

This following program code is the LocalSolana Protocol. The protocol is still under development and may have changes or new functions in the future.

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::{ program::invoke, system_instruction };
use anchor_spl::token::{ self, Mint, Token, TokenAccount, Transfer };
use anchor_spl::associated_token::AssociatedToken;
use anchor_spl::associated_token::{ create, get_associated_token_address };

declare_id!("1w3ekpHrruiEJPYKpQH6rQssTRNKCKiqUjfQeJXTTrX");

#[program]
pub mod local_solana_migrate {
    use super::*;

    pub const DISPUTE_FEE: u64 = 5_000_000;

    pub fn initialize(
        ctx: Context<Initialize>,
        fee_bps: u64,
```

```
        dispute_fee: u64,
        fee_discount_nft: Pubkey
    ) -> Result<()> {
        let escrow_state = &mut ctx.accounts.escrow_state;
        require!(!escrow_state.is_initialized,
SolanaErrorCode::AlreadyInitialized);
        escrow_state.is_initialized = true;
        escrow_state.seller = *ctx.accounts.seller.key;
        escrow_state.fee_bps = fee_bps;
        escrow_state.arbitrator = *ctx.accounts.arbitrator.key;
        escrow_state.fee_recipient = *ctx.accounts.fee_recipient.key;
        escrow_state.fee_discount_nft = fee_discount_nft;
        escrow_state.dispute_fee = dispute_fee;
        // escrow_state.deployer = *ctx.accounts.deployer.key;
        Ok(())
    }

    pub fn create_escrow_sol(
        ctx: Context<CreateEscrowSOL>,
        order_id: String,
        amount: u64,
        seller_waiting_time: i64,
        automatic_escrow: bool
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
        require!(
            ctx.accounts.buyer.key() != ctx.accounts.seller.key(),
            SolanaErrorCode::InvalidBuyer
        );
        require!(
            seller_waiting_time >= 15 * 60 && seller_waiting_time <= 24 * 60 * 60,
            SolanaErrorCode::InvalidSellerWaitingTime
        );

        let escrow_account = &mut ctx.accounts.escrow;
        require!(!escrow_account.exists, SolanaErrorCode::OrderAlreadyExists);

        escrow_account.exists = true;
        escrow_account.seller_can_cancel_after = Clock::get()?.unix_timestamp +
seller_waiting_time;
        escrow_account.fee = ((amount * ctx.accounts.escrow_state.fee_bps) / 10000)
as u64;
        escrow_account.dispute = false;
        escrow_account.partner = ctx.accounts.partner.key();
        escrow_account.open_peer_fee = ((amount * 30) / 10000) as u64;
        escrow_account.automatic_escrow = automatic_escrow;
        escrow_account.amount = amount;
        escrow_account.token = Pubkey::default();
        escrow_account.seller = *ctx.accounts.seller.key;
        escrow_account.buyer = *ctx.accounts.buyer.key;

        let seller_info = ctx.accounts.seller.to_account_info();
        let escrow_info = ctx.accounts.escrow.to_account_info();
        let system_program_info = ctx.accounts.system_program.to_account_info();
        // // amount  = amount+ &ctx.accounts.escrow.open_peer_fee;
        let fee_amount = ctx.accounts.escrow.amount + ctx.accounts.escrow.fee;
        if automatic_escrow {
            //Transfer lamports from seller to escrow account
            invoke(
```

```
                &system_instruction::transfer(seller_info.key, escrow_info.key,
fee_amount),
                &[seller_info.clone(), escrow_info.clone(),
system_program_info.clone()]
            )?;
        }
        emit!(EscrowCreated { order_id });
        Ok(())
    }

    pub fn create_escrow_sol_buyer(
        ctx: Context<CreateEscrowSOLBuyer>,
        order_id: String,
        amount: u64,
        seller_waiting_time: i64,
        automatic_escrow: bool
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
        require!(
            ctx.accounts.buyer.key() != ctx.accounts.seller.key(),
            SolanaErrorCode::InvalidBuyer
        );
        require!(
            seller_waiting_time >= 15 * 60 && seller_waiting_time <= 24 * 60 * 60,
            SolanaErrorCode::InvalidSellerWaitingTime
        );

        let escrow_account = &mut ctx.accounts.escrow;
        require!(!escrow_account.exists, SolanaErrorCode::OrderAlreadyExists);

        escrow_account.exists = true;
        escrow_account.seller_can_cancel_after = Clock::get()?.unix_timestamp +
seller_waiting_time;
        escrow_account.fee = ((amount * ctx.accounts.escrow_state.fee_bps) / 10000)
as u64;
        escrow_account.dispute = false;
        escrow_account.partner = ctx.accounts.partner.key();
        escrow_account.open_peer_fee = ((amount * 30) / 10000) as u64;
        escrow_account.automatic_escrow = automatic_escrow;
        escrow_account.amount = amount;
        escrow_account.token = Pubkey::default();
        escrow_account.seller = *ctx.accounts.seller.key;
        escrow_account.buyer = *ctx.accounts.buyer.key;
        // // amount  = amount+ &ctx.accounts.escrow.open_peer_fee;
        let fee_amount = ctx.accounts.escrow.amount + ctx.accounts.escrow.fee;

        if automatic_escrow {


**ctx.accounts.escrow_state.to_account_info().try_borrow_mut_lamports()? -=
fee_amount;
            **ctx.accounts.escrow.to_account_info().try_borrow_mut_lamports()? +=
fee_amount;
        }
        emit!(EscrowCreated { order_id });
        Ok(())
    }

    pub fn create_escrow_token(
```

```
        ctx: Context<CreateEscrowToken>,
        order_id: String,
        amount: u64,
        seller_waiting_time: i64,
        automatic_escrow: bool,
        token: Pubkey,
        from_wallet: bool
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
        require!(
            ctx.accounts.buyer.key() != ctx.accounts.seller.key(),
            SolanaErrorCode::InvalidBuyer
        );
        require!(
            seller_waiting_time >= 15 * 60 && seller_waiting_time <= 24 * 60 * 60,
            SolanaErrorCode::InvalidSellerWaitingTime
        );
        let escrow_account = &mut ctx.accounts.escrow;
        require!(!escrow_account.exists, SolanaErrorCode::OrderAlreadyExists);

        escrow_account.exists = true;
        escrow_account.seller_can_cancel_after = Clock::get()?.unix_timestamp +
seller_waiting_time;
        escrow_account.fee = ((amount * ctx.accounts.escrow_state.fee_bps) / 10000)
as u64;
        escrow_account.dispute = false;
        escrow_account.partner = ctx.accounts.partner.key();
        escrow_account.open_peer_fee = ((amount * 30) / 10000) as u64;
        escrow_account.automatic_escrow = automatic_escrow;
        escrow_account.amount = amount;
        escrow_account.token = token;
        escrow_account.seller = *ctx.accounts.seller.key;
        escrow_account.buyer = *ctx.accounts.buyer.key;
        msg!("Automatic Escrow: {}", automatic_escrow);
        if automatic_escrow {
            if
ctx.accounts.escrow_token_account.to_account_info().try_borrow_data()?.is_empty() {
                msg!(
                    "Escrow's token associated token account is not initialized.
Initializing it..."
                );

                // If the token account doesn't exist, create it
                let cpi_accounts = anchor_spl::associated_token::Create {
                    payer: ctx.accounts.fee_payer.to_account_info(),
                    associated_token:
ctx.accounts.escrow_token_account.to_account_info(),
                    authority: escrow_account.to_account_info(),
                    mint: ctx.accounts.mint_account.to_account_info(),
                    system_program: ctx.accounts.system_program.to_account_info(),
                    token_program: ctx.accounts.token_program.to_account_info(),
                };

                let cpi_program =
ctx.accounts.associated_token_program.to_account_info();
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

                create(cpi_ctx)?;
```

```
                msg!("Escrow's USDC associated token account has been successfully
initialized.");
            }

            if !from_wallet {
                if
                    ctx.accounts.escrow_state_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info()
                        .try_borrow_data()?
                        .is_empty()
                {
                    msg!(
                        "Escrow state token associated token account is not
initialized. Initializing it..."
                    );

                    // If the token account doesn't exist, create it
                    let cpi_accounts = anchor_spl::associated_token::Create {
                        payer: ctx.accounts.fee_payer.to_account_info(),
                        associated_token: ctx.accounts.escrow_state_token_account
                            .as_ref()
                            .ok_or(SolanaErrorCode::AccountError)?
                            .to_account_info(),
                        authority: ctx.accounts.escrow_state.to_account_info(),
                        mint: ctx.accounts.mint_account.to_account_info(),
                        system_program:
ctx.accounts.system_program.to_account_info(),
                        token_program:
ctx.accounts.token_program.to_account_info(),
                    };

                    let cpi_program =
ctx.accounts.associated_token_program.to_account_info();
                    let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

                    create(cpi_ctx)?;
                    msg!(
                        "Escrow state token associated token account has been
successfully initialized."
                    );
                }
            }
            // Perform the transfer
            let cpi_accounts = if from_wallet {
                Transfer {
                    from: ctx.accounts.seller_token_account.to_account_info(),
                    to: ctx.accounts.escrow_token_account.to_account_info(),
                    authority: ctx.accounts.seller.to_account_info(),
                }
            } else {
                Transfer {
                    from: ctx.accounts.escrow_state_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
                    to: ctx.accounts.escrow_token_account.to_account_info(),
                    authority: ctx.accounts.escrow_state.to_account_info(),
```

```
                }
            };
            let cpi_program = ctx.accounts.token_program.to_account_info();
            if from_wallet {
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
                token::transfer(cpi_ctx, amount + escrow_account.fee)?;
            } else {
                let seller_key = ctx.accounts.seller.key();
                let escrow_state_seeds = &[
                    b"escrow_state",
                    seller_key.as_ref(),
                    &[ctx.bumps.escrow_state],
                ];
                let seeds: &[&[&[u8]]] = &[escrow_state_seeds];
                let cpi_ctx = CpiContext::new_with_signer(
                    ctx.accounts.token_program.to_account_info(),
                    cpi_accounts,
                    seeds
                );
                msg!("Sending: {}",amount+escrow_account.fee);
                token::transfer(cpi_ctx, amount + escrow_account.fee)?;
            }
        }

        emit!(EscrowCreated { order_id });
        Ok(())
    }

    pub fn create_escrow_token_buyer(
        ctx: Context<CreateEscrowTokenBuyer>,
        order_id: String,
        amount: u64,
        seller_waiting_time: i64,
        automatic_escrow: bool,
        token: Pubkey
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
        require!(
            ctx.accounts.buyer.key() != ctx.accounts.seller.key(),
            SolanaErrorCode::InvalidBuyer
        );
        require!(
            seller_waiting_time >= 15 * 60 && seller_waiting_time <= 24 * 60 * 60,
            SolanaErrorCode::InvalidSellerWaitingTime
        );
        let escrow_account = &mut ctx.accounts.escrow;
        require!(!escrow_account.exists, SolanaErrorCode::OrderAlreadyExists);

        escrow_account.exists = true;
        escrow_account.seller_can_cancel_after = Clock::get()?.unix_timestamp +
seller_waiting_time;
        escrow_account.fee = ((amount * ctx.accounts.escrow_state.fee_bps) / 10000)
as u64;
        escrow_account.dispute = false;
        escrow_account.partner = ctx.accounts.partner.key();
        escrow_account.open_peer_fee = ((amount * 30) / 10000) as u64;
        escrow_account.automatic_escrow = automatic_escrow;
        escrow_account.amount = amount;
        escrow_account.token = token;
```

```
        escrow_account.seller = *ctx.accounts.seller.key;
        escrow_account.buyer = *ctx.accounts.buyer.key;
        msg!("Automatic Escrow: {}", automatic_escrow);
            if
ctx.accounts.escrow_token_account.to_account_info().try_borrow_data()?.is_empty() {
                msg!(
                    "Escrow's token associated token account is not initialized.
Initializing it..."
                );

                // If the token account doesn't exist, create it
                let cpi_accounts = anchor_spl::associated_token::Create {
                    payer: ctx.accounts.fee_payer.to_account_info(),
                    associated_token:
ctx.accounts.escrow_token_account.to_account_info(),
                    authority: escrow_account.to_account_info(),
                    mint: ctx.accounts.mint_account.to_account_info(),
                    system_program: ctx.accounts.system_program.to_account_info(),
                    token_program: ctx.accounts.token_program.to_account_info(),
                };

                let cpi_program =
ctx.accounts.associated_token_program.to_account_info();
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

                create(cpi_ctx)?;
                msg!("Escrow's USDC associated token account has been successfully
initialized.");
            }

            // Perform the transfer
            let cpi_accounts = Transfer {
                from: ctx.accounts.escrow_state_token_account
                    .as_ref()
                    .ok_or(SolanaErrorCode::AccountError)?
                    .to_account_info(),
                to: ctx.accounts.escrow_token_account.to_account_info(),
                authority: ctx.accounts.escrow_state.to_account_info(),
            };
            let seller_key = ctx.accounts.seller.key();
                let escrow_state_seeds = &[
                    b"escrow_state",
                    seller_key.as_ref(),
                    &[ctx.bumps.escrow_state],
                ];
                let seeds: &[&[&[u8]]] = &[escrow_state_seeds];
                let cpi_ctx = CpiContext::new_with_signer(
                    ctx.accounts.token_program.to_account_info(),
                    cpi_accounts,
                    seeds
                );
                msg!("Sending: {}",amount+escrow_account.fee);
                token::transfer(cpi_ctx, amount + escrow_account.fee)?;

        emit!(EscrowCreated { order_id });
        Ok(())
    }

    pub fn mark_as_paid<'info>(
```

```
        ctx: Context<'_, '_, 'info, 'info, MarkAsPaid>,
        order_id: String
    ) -> Result<()> {
        let escrow = &mut ctx.accounts.escrow;
        require!(escrow.exists, SolanaErrorCode::EscrowNotFound);
        if escrow.seller_can_cancel_after != 1 {
            escrow.seller_can_cancel_after = 1;
            emit!(SellerCancelDisabled { order_id: order_id });
        }
        Ok(())
    }

    pub fn release_funds(ctx: Context<ReleaseFunds>, order_id: String) ->
Result<()> {
        require!(ctx.accounts.escrow.exists, SolanaErrorCode::EscrowNotFound);
        require!(
            ctx.accounts.escrow.seller_can_cancel_after == 1,
            SolanaErrorCode::CannotReleaseFundsYet
        );
        require!(
            &ctx.accounts.escrow_state.fee_recipient ==
ctx.accounts.fee_recipient.key,
            SolanaErrorCode::InvalidFeeRecepient
        );
        require!(
            ctx.accounts.escrow.buyer == *ctx.accounts.buyer.key,
            SolanaErrorCode::InvalidBuyer
        );

        require!(
            ctx.accounts.escrow.buyer != *ctx.accounts.seller.key,
            SolanaErrorCode::InvalidBuyer
        );

        if ctx.accounts.escrow.token == Pubkey::default() {
            let fee_amount = ctx.accounts.escrow.fee;
            let total_amount = ctx.accounts.escrow.amount + fee_amount;
            let account_size: usize =
ctx.accounts.escrow.to_account_info().data_len();

            // Fetch the rent sysvar
            let rent = Rent::get()?;

            // Calculate the rent-exempt reserve for this account
            let rent_exempt_balance = rent.minimum_balance(account_size);

            // Get the actual balance in the account
            let actual_balance =
**ctx.accounts.escrow.to_account_info().lamports.borrow();

            // Calculate the total balance (including the rent-exempt amount)
            let total_balance = actual_balance + rent_exempt_balance;
            msg!("Fee Amount:{}", fee_amount);
            msg!("Escrow Lamports: {}", total_balance);
            msg!("Total Amount: {}", total_amount);

            // Check if escrow account has enough lamports
            //require!(total_balance >= total_amount,
SolanaErrorCode::InsufficientFunds);
```

```
            **ctx.accounts.buyer.to_account_info().try_borrow_mut_lamports()? +=
                ctx.accounts.escrow.amount;
            **ctx.accounts.escrow.to_account_info().try_borrow_mut_lamports()? -=
                ctx.accounts.escrow.amount;

**ctx.accounts.fee_recipient.to_account_info().try_borrow_mut_lamports()? +=
fee_amount;
            **ctx.accounts.escrow.to_account_info().try_borrow_mut_lamports()? -=
fee_amount;
        } else {
            msg!("Amount is : {}", ctx.accounts.escrow.amount);
            msg!("Fee is : {}", ctx.accounts.escrow.fee);
            // msg!(
            //     "Escrow Token Account Balance: {}",
            //     let escrow_token_account: TokenAccount =
TokenAccount::try_from(&ctx.accounts.escrow_token_account.to_account_info())?;
            //     escrow_token_account.amount;
            // );
            let (_escrow_pda, _bump) = Pubkey::find_program_address(
                &[b"escrow", order_id.as_bytes()],
                ctx.program_id
            );

            let cpi_accounts = Transfer {
                from: ctx.accounts.escrow_token_account
                    .as_ref()
                    .ok_or(SolanaErrorCode::AccountError)?
                    .to_account_info(),
                to: ctx.accounts.buyer_token_account
                    .as_ref()
                    .ok_or(SolanaErrorCode::AccountError)?
                    .to_account_info(),
                authority: ctx.accounts.escrow.to_account_info(),
            };

            let cpi_program = ctx.accounts.token_program.to_account_info();
            let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

            // Perform the token transfer (with decimals checked)
            token::transfer(
                cpi_ctx.with_signer(&[&[b"escrow", order_id.as_bytes(),
&[_bump]]]),
                ctx.accounts.escrow.amount
            )?;
            msg!("Transferred tokens to buyer");
            // Transfer fee tokens from escrow PDA to the fee recipient's
associated token account
            let cpi_accounts_fee = Transfer {
                from: ctx.accounts.escrow_token_account
                    .as_ref()
                    .ok_or(SolanaErrorCode::AccountError)?
                    .to_account_info(),
                to: ctx.accounts.fee_recipient_token_account
                    .as_ref()
                    .ok_or(SolanaErrorCode::AccountError)?
                    .to_account_info(),
                authority: ctx.accounts.escrow.to_account_info(),
            };
```

```
            let cpi_program2 = ctx.accounts.token_program.to_account_info();
            let cpi_ctx_fee = CpiContext::new(cpi_program2, cpi_accounts_fee);

            // Perform the token transfer for the fee (with decimals checked)
            token::transfer(
                cpi_ctx_fee.with_signer(&[&[b"escrow", order_id.as_bytes(),
&[_bump]]]),
                ctx.accounts.escrow.fee
            )?;
        }

        ctx.accounts.escrow.exists = false;
        emit!(Released { order_id: order_id });
        Ok(())
    }

    pub fn buyer_cancel(ctx: Context<CancelEscrow>, order_id: String) -> Result<()>
{
        let escrow = &ctx.accounts.escrow;
        require!(escrow.exists, SolanaErrorCode::EscrowNotFound);

        if escrow.token == Pubkey::default() {
            **ctx.accounts.seller.to_account_info().try_borrow_mut_lamports()? +=
                escrow.amount + escrow.fee;
            **ctx.accounts.escrow.to_account_info().try_borrow_mut_lamports()? -=
                escrow.amount + escrow.fee;
        } else {
            // if let (Some(seller_token_account), Some(escrow_token_account)) = (
            //     &ctx.accounts.seller_token_account,
            //     &ctx.accounts.escrow_token_account,
            // ) {
            let cpi_accounts = Transfer {
                from: escrow.to_account_info(),
                to: ctx.accounts.seller.to_account_info(),
                authority: ctx.accounts.seller.to_account_info(),
            };
            let cpi_program = ctx.accounts.token_program.to_account_info();
            let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
            token::transfer(cpi_ctx, escrow.amount + escrow.fee)?;
            // }
        }

        ctx.accounts.escrow.exists = false;
        emit!(CancelledByBuyer { order_id: order_id });
        Ok(())
    }

    pub fn seller_cancel(ctx: Context<CancelEscrow>, order_id: String) ->
Result<()> {
        let escrow = &mut ctx.accounts.escrow;
        require!(escrow.exists, SolanaErrorCode::EscrowNotFound);
        require!(
            escrow.seller_can_cancel_after > 1 &&
                escrow.seller_can_cancel_after <= Clock::get()?.unix_timestamp,
            SolanaErrorCode::CannotCancelYet
        );

        if escrow.token == Pubkey::default() {
```

```
            **ctx.accounts.seller.to_account_info().try_borrow_mut_lamports()? +=
                escrow.amount + escrow.fee;
            **escrow.to_account_info().try_borrow_mut_lamports()? -= escrow.amount
+ escrow.fee;
        } else {
            // if let (Some(seller_token_account), Some(escrow_token_account)) = (
            //     &ctx.accounts.seller_token_account,
            //     &ctx.accounts.escrow_token_account,
            // ) {
            let cpi_accounts = Transfer {
                from: escrow.to_account_info(),
                to: ctx.accounts.seller.to_account_info(),
                authority: ctx.accounts.seller.to_account_info(),
            };
            let cpi_program = ctx.accounts.token_program.to_account_info();
            let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
            token::transfer(cpi_ctx, escrow.amount + escrow.fee)?;
            //}
        }

        escrow.exists = false;
        emit!(CancelledBySeller { order_id: order_id });
        Ok(())
    }

    pub fn open_dispute(ctx: Context<OpenDispute>, order_id: String) -> Result<()>
{
        let escrow_state = &ctx.accounts.escrow_state;
        require!(
            ctx.accounts.payer.to_account_info().lamports() >=
escrow_state.dispute_fee,
            SolanaErrorCode::InsufficientFundsForDispute
        );

        require!(
            ctx.accounts.payer.key() == ctx.accounts.escrow.buyer.key() ||
                ctx.accounts.payer.key() == ctx.accounts.escrow.seller.key(),
            SolanaErrorCode::InsufficientFundsForDispute
        );

        let escrow = &mut ctx.accounts.escrow;
        require!(escrow.exists, SolanaErrorCode::EscrowNotFound);
        require!(escrow.seller_can_cancel_after == 1,
SolanaErrorCode::CannotOpenDisputeYet);

        // Mark the party that opened the dispute
        if ctx.accounts.payer.key() == escrow.buyer.key() {
            escrow.buyer_paid_dispute = true;
        } else if ctx.accounts.payer.key() == escrow.seller.key() {
            escrow.seller_paid_dispute = true;
        }

        // Transfer dispute fee from payer to program
        invoke(
            &system_instruction::transfer(
                ctx.accounts.payer.to_account_info().key,
                escrow.to_account_info().key,
                DISPUTE_FEE
            ),
```

```
            &[
                ctx.accounts.payer.to_account_info(),
                ctx.accounts.escrow_state.to_account_info(),
                ctx.accounts.system_program.to_account_info(),
            ]
        )?;

        escrow.dispute = true;
        emit!(DisputeOpened {
            order_id: order_id,
            sender: *ctx.accounts.payer.key,
        });
        Ok(())
    }

    pub fn resolve_dispute(
        ctx: Context<ResolveDispute>,
        order_id: String,
        winner: Pubkey
    ) -> Result<()> {
        let escrow = &mut ctx.accounts.escrow;
        require!(escrow.exists, SolanaErrorCode::EscrowNotFound);
        require!(escrow.dispute, SolanaErrorCode::DisputeNotOpen);
        require!(
            winner == ctx.accounts.seller.key() || winner ==
ctx.accounts.buyer.key(),
            SolanaErrorCode::InvalidWinner
        );

        let winner_account = if winner == ctx.accounts.seller.key() {
            &ctx.accounts.seller
        } else {
            &ctx.accounts.buyer
        };
        let arbitrator = ctx.accounts.arbitrator.to_account_info();

        // Transfer 0.005 SOL from program to winner
        invoke(
            &system_instruction::transfer(
                ctx.accounts.escrow_state.to_account_info().key,
                &winner_account.key(),
                DISPUTE_FEE
            ),
            &[
                ctx.accounts.escrow_state.to_account_info(),
                winner_account.to_account_info(),
                ctx.accounts.system_program.to_account_info(),
            ]
        )?;

        // Transfer 0.005 SOL from program to arbitrator
        invoke(
            &system_instruction::transfer(
                ctx.accounts.escrow_state.to_account_info().key,
                arbitrator.key,
                DISPUTE_FEE
            ),
            &[
                ctx.accounts.escrow_state.to_account_info(),
```

```
                arbitrator,
                ctx.accounts.system_program.to_account_info(),
            ]
        )?;

        if escrow.token == Pubkey::default() {
            if winner == ctx.accounts.buyer.key() {
                **ctx.accounts.buyer.to_account_info().try_borrow_mut_lamports()?
+= escrow.amount;
            } else {
                **ctx.accounts.seller.to_account_info().try_borrow_mut_lamports()?
+= escrow.amount;
            }
            **escrow.to_account_info().try_borrow_mut_lamports()? -= escrow.amount;
        } else {
            let to_account_info = if winner == ctx.accounts.buyer.key() {
                ctx.accounts.buyer.to_account_info()
            } else {
                ctx.accounts.seller.to_account_info()
            };
            let cpi_accounts = Transfer {
                from: escrow.to_account_info(),
                to: to_account_info,
                authority: ctx.accounts.seller.to_account_info(),
            };
            let cpi_program = ctx.accounts.token_program.to_account_info();
            let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
            token::transfer(cpi_ctx, escrow.amount)?;
        }

        escrow.exists = false;
        emit!(DisputeResolved { order_id, winner });
        Ok(())
    }

    pub fn deposit_to_escrow_state(
        ctx: Context<DepositToEscrowState>,
        amount: u64,
        token: Pubkey
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
        let escrow_state = &ctx.accounts.escrow_state;
        let fee = escrow_state.fee_bps;
        let total_amount = amount + fee;
        msg!("Fee Amount:{}", fee);
        msg!("Total Amount: {}", total_amount);
        if token == Pubkey::default() {
            require!(
                ctx.accounts.seller.to_account_info().lamports() >= total_amount,
                SolanaErrorCode::InsufficientFunds
            );
            **ctx.accounts.seller.to_account_info().try_borrow_mut_lamports()? -=
total_amount;

**ctx.accounts.escrow_state.to_account_info().try_borrow_mut_lamports()? +=
                total_amount;
            Ok(())
        } else {
            msg!("Starting Token transfer");
```

```
            // Get the associated token account for the wallet (sender)
            let wallet_token_account = get_associated_token_address(
                &ctx.accounts.seller.key(),
                &ctx.accounts.mint_account.key()
            );

            // Derive the associated token account for the escrow PDA (receiver)
            let escrow_token_account = get_associated_token_address(
                &ctx.accounts.escrow_state.key(),
                &ctx.accounts.mint_account.key()
            );

            // Log the derived token accounts for debugging
            msg!("Wallet USDC Token Account: {}", wallet_token_account);
            msg!("Escrow USDC Token Account: {}", escrow_token_account);

            // Check if the escrow_state's associated token account already exists
            if
                ctx.accounts.escrow_state_token_account
                    .to_account_info()
                    .try_borrow_data()?
                    .is_empty()
            {
                msg!(
                    "Escrow's USDC associated token account is not initialized.
Initializing it..."
                );

                // If the token account doesn't exist, create it
                let cpi_accounts = anchor_spl::associated_token::Create {
                    payer: ctx.accounts.fee_payer.to_account_info(),
                    associated_token:
ctx.accounts.escrow_state_token_account.to_account_info(),
                    authority: ctx.accounts.escrow_state.to_account_info(),
                    mint: ctx.accounts.mint_account.to_account_info(),
                    system_program: ctx.accounts.system_program.to_account_info(),
                    token_program: ctx.accounts.token_program.to_account_info(),
                };

                let cpi_program =
ctx.accounts.associated_token_program.to_account_info();
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

                create(cpi_ctx)?;
                msg!("Escrow's USDC associated token account has been successfully
initialized.");
            }

            let cpi_accounts = Transfer {
                from: ctx.accounts.wallet_token_account.to_account_info(),
                to: ctx.accounts.escrow_state_token_account.to_account_info(),
                authority: ctx.accounts.seller.to_account_info(),
            };
            msg!("After CPI accounts: {}", ctx.accounts.token_program.key());
            let cpi_program = ctx.accounts.token_program.to_account_info();
            msg!("After cpi program");
            let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
            msg!("After cpi ctx");
```

```
            token::transfer(cpi_ctx, total_amount)?;

            Ok(())
        }
    }

    pub fn deposit_to_escrow(
        ctx: Context<DepositToEscrow>,
        order_id: String,
        amount: u64,
        token: Pubkey,
        instant_escrow: bool
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
        let escrow_account = &ctx.accounts.escrow;
        let amount = escrow_account.amount;
        let fee = escrow_account.fee;
        let total_amount = amount + fee;
        msg!("Fee Amount:{}", fee);
        msg!("Total Amount: {}", total_amount);
        if token == Pubkey::default() {
            require!(
                ctx.accounts.escrow_state.to_account_info().lamports() >=
total_amount,
                SolanaErrorCode::InsufficientFunds
            );
            if instant_escrow {

**ctx.accounts.escrow_state.to_account_info().try_borrow_mut_lamports()? -=
                    total_amount;
                **ctx.accounts.escrow.to_account_info().try_borrow_mut_lamports()?
+= total_amount;
            } else {
                **ctx.accounts.seller.to_account_info().try_borrow_mut_lamports()?
-= total_amount;
                **ctx.accounts.escrow.to_account_info().try_borrow_mut_lamports()?
+= total_amount;
            }
        } else {
            if
                ctx.accounts.escrow_token_account
                    .as_ref()
                    .ok_or(SolanaErrorCode::AccountError)?
                    .to_account_info()
                    .try_borrow_data()?
                    .is_empty()
            {
                msg!(
                    "Escrow's token associated token account is not initialized.
Initializing it..."
                );

                // If the token account doesn't exist, create it
                let cpi_accounts = anchor_spl::associated_token::Create {
                    payer: ctx.accounts.fee_payer.to_account_info(),
                    associated_token: ctx.accounts.escrow_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
```

```
                    authority: escrow_account.to_account_info(),
                    mint: ctx.accounts.mint_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
                    system_program: ctx.accounts.system_program.to_account_info(),
                    token_program: ctx.accounts.token_program.to_account_info(),
                };

                let cpi_program =
ctx.accounts.associated_token_program.to_account_info();
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);

                create(cpi_ctx)?;
                msg!("Escrow's associated token account has been successfully
initialized.");
            }
            if instant_escrow {
                let cpi_accounts = Transfer {
                    from: ctx.accounts.escrow_state_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
                    to: ctx.accounts.escrow_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
                    authority: ctx.accounts.seller.to_account_info(),
                };
                let cpi_program = ctx.accounts.token_program.to_account_info();
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
                token::transfer(cpi_ctx, total_amount)?;
            } else {
                let cpi_accounts = Transfer {
                    from: ctx.accounts.seller_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
                    to: ctx.accounts.escrow_token_account
                        .as_ref()
                        .ok_or(SolanaErrorCode::AccountError)?
                        .to_account_info(),
                    authority: ctx.accounts.seller.to_account_info(),
                };
                let cpi_program = ctx.accounts.token_program.to_account_info();
                let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
                token::transfer(cpi_ctx, total_amount)?;
            }
        }
        msg!("Transferred: {}", order_id);
        Ok(())
    }

    pub fn withdraw_balance(
        ctx: Context<WithdrawBalance>,
        amount: u64,
        token: Pubkey
    ) -> Result<()> {
        require!(amount > 0, SolanaErrorCode::InvalidAmount);
```

```
        if token == Pubkey::default() {

**ctx.accounts.escrow_state.to_account_info().try_borrow_mut_lamports()? -= amount;
            **ctx.accounts.seller.to_account_info().try_borrow_mut_lamports()? +=
amount;
        } else {
            // if let (Some(seller_token_account), Some(escrow_token_account)) = (
            //     &ctx.accounts.seller_token_account,
            //     &ctx.accounts.escrow_token_account,
            // ) {
            let cpi_accounts = Transfer {
                from: ctx.accounts.escrow_state.to_account_info(),
                to: ctx.accounts.seller.to_account_info(),
                authority: ctx.accounts.seller.to_account_info(),
            };
            let cpi_program = ctx.accounts.token_program.to_account_info();
            let cpi_ctx = CpiContext::new(cpi_program, cpi_accounts);
            token::transfer(cpi_ctx, amount)?;
            //}
        }

        Ok(())
    }
}

#[derive(Accounts)]
//#[instruction(bump: u8)]
pub struct Initialize<'info> {
    #[account(
        init,
        payer = fee_payer,
        space = 8 + 177,
        seeds = [b"escrow_state", seller.key().as_ref()],
        bump
    )]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut)]
    /// CHECK: This is safe because the seller is a trusted party
    pub seller: AccountInfo<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    /// CHECK: This is safe because the arbitrator is a trusted party
    pub arbitrator: AccountInfo<'info>,
    /// CHECK: This is safe because the fee_recipient is a trusted party
    pub fee_recipient: AccountInfo<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
#[instruction(order_id: String, bump: u8)]
pub struct CreateEscrowSOL<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(
        init,
        payer = fee_payer,
        space = 8 + 165 + 8,
        seeds = [b"escrow", order_id.as_bytes()],
```

```
        bump
    )]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    /// CHECK: This is safe because the seller is being checked in program
    pub seller: Signer<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    /// CHECK: This is safe because the buyer is a trusted party
    pub buyer: AccountInfo<'info>,
    pub system_program: Program<'info, System>,
    /// CHECK: This is safe because the partner is a trusted party
    pub partner: AccountInfo<'info>,
}

#[derive(Accounts)]
#[instruction(order_id: String, bump: u8)]
pub struct CreateEscrowSOLBuyer<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(
        init,
        payer = fee_payer,
        space = 8 + 165 + 8,
        seeds = [b"escrow", order_id.as_bytes()],
        bump
    )]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    /// CHECK: This is safe because the seller is being checked in program
    pub seller: AccountInfo<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    /// CHECK: This is safe because the buyer is a trusted party
    #[account(mut)]
    pub buyer: Signer<'info>,
    pub system_program: Program<'info, System>,
    /// CHECK: This is safe because the partner is a trusted party
    pub partner: AccountInfo<'info>,
}

#[derive(Accounts)]
#[instruction(order_id: String, bump: u8)]
pub struct CreateEscrowToken<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(
        init,
        payer = fee_payer,
        space = 8 + 165 + 8,
        seeds = [b"escrow", order_id.as_bytes()],
        bump
    )]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    /// CHECK: This is safe because the seller is being checked in program
    pub seller: Signer<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
```

```
    /// CHECK: This is safe because the buyer is a trusted party
    pub buyer: AccountInfo<'info>,
    #[account(mut)]
    /// CHECK This is safe because the escrow token account is being verified in
program
    pub escrow_token_account: UncheckedAccount<'info>,
    pub token_program: Program<'info, Token>,
    /// CHECK: This is safe because the partner is a trusted party
    pub partner: AccountInfo<'info>,
    pub mint_account: Account<'info, Mint>,
    #[account(
        mut,
        associated_token::mint = mint_account,
        associated_token::authority = seller
    )]
    pub seller_token_account: Account<'info, TokenAccount>,
    #[account(
        mut,
    )]
    /// CHECK: This is safe because the escrow_state_token_account is derived from
the escrow_state
    pub escrow_state_token_account: Option<UncheckedAccount<'info>>,
    /// Associated Token Program for creating token accounts
    pub associated_token_program: Program<'info, AssociatedToken>,
    /// System Program (needed to create token accounts)
    pub system_program: Program<'info, System>,
    /// Rent sysvar (for rent exemption)
    pub rent: Sysvar<'info, Rent>,
}

#[derive(Accounts)]
#[instruction(order_id: String, bump: u8)]
pub struct CreateEscrowTokenBuyer<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(
        init,
        payer = fee_payer,
        space = 8 + 165 + 8,
        seeds = [b"escrow", order_id.as_bytes()],
        bump
    )]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    /// CHECK: This is safe because the seller is being checked in program
    pub seller: AccountInfo<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    /// CHECK: This is safe because the buyer is a trusted party
    #[account(mut)]
    pub buyer: Signer<'info>,
    #[account(mut)]
    /// CHECK This is safe because the escrow token account is being verified in
program
    pub escrow_token_account: UncheckedAccount<'info>,
    pub token_program: Program<'info, Token>,
    /// CHECK: This is safe because the partner is a trusted party
    pub partner: AccountInfo<'info>,
    pub mint_account: Account<'info, Mint>,
```

```
    #[account(
        mut,
        associated_token::mint = mint_account,
        associated_token::authority = seller
    )]
    pub seller_token_account: Account<'info, TokenAccount>,
    #[account(
        mut,
    )]
    /// CHECK: This is safe because the escrow_state_token_account is derived from
the escrow_state
    pub escrow_state_token_account: Option<UncheckedAccount<'info>>,
    /// Associated Token Program for creating token accounts
    pub associated_token_program: Program<'info, AssociatedToken>,
    /// System Program (needed to create token accounts)
    pub system_program: Program<'info, System>,
    /// Rent sysvar (for rent exemption)
    pub rent: Sysvar<'info, Rent>,
}

#[derive(Accounts)]
#[instruction(order_id: String)]
pub struct MarkAsPaid<'info> {
    #[account(mut, seeds = [b"escrow", order_id.as_bytes()], bump)]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    pub buyer: Signer<'info>,
    /// CHECK: This is safe because the seller is a trusted party
    pub seller: AccountInfo<'info>,
    // System program is required for PDA derivation
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
#[instruction(order_id: String)]
pub struct ReleaseFunds<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut, seeds = [b"escrow", order_id.as_bytes()], bump)]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    pub seller: Signer<'info>,
    /// CHECK: This is safe because the buyer is validated by the program
    #[account(mut)]
    pub buyer: AccountInfo<'info>,
    /// CHECK: This is safe because the Fee Recipient is validated by the program
    #[account(mut)]
    pub fee_recipient: AccountInfo<'info>,
    #[account(mut,)]
    pub fee_recipient_token_account: Option<UncheckedAccount<'info>>,

    pub token_program: Program<'info, Token>,
    /// CHECK: This is safe because the Fee Recipient is validated by the program
    pub mint_account: UncheckedAccount<'info>,
    #[account(
        mut,
    )]
    pub escrow_token_account: Option<UncheckedAccount<'info>>,
    #[account(
```

```
        mut,
    )]
    pub buyer_token_account: Option<UncheckedAccount<'info>>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
}

#[derive(Accounts)]
#[instruction(order_id: String)]
pub struct CancelEscrow<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut, seeds = [b"escrow", order_id.as_bytes()], bump)]
    pub escrow: Account<'info, Escrow>,
    pub seller: Signer<'info>,
    // #[account(mut, constraint = escrow_token_account.owner == TOKEN_PROGRAM_ID)]
    // pub escrow_token_account: Option<Account<'info, TokenAccount>>,
    // #[account(mut, constraint = seller_token_account.owner == TOKEN_PROGRAM_ID)]
    // pub seller_token_account: Option<Account<'info, TokenAccount>>,
    pub token_program: Program<'info, Token>,
}

#[derive(Accounts)]
#[instruction(order_id: String)]
pub struct OpenDispute<'info> {
    #[account(mut, seeds = [b"escrow_state", payer.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut, seeds = [b"escrow", order_id.as_bytes()], bump)]
    pub escrow: Account<'info, Escrow>,
    pub payer: Signer<'info>,
    // System program is required for PDA derivation
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
#[instruction(order_id: String)]
pub struct ResolveDispute<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut, seeds = [b"escrow", order_id.as_bytes()], bump)]
    pub escrow: Account<'info, Escrow>,
    /// CHECK: This is safe because the arbitrator is a trusted party
    pub arbitrator: Signer<'info>,
    /// CHECK: This is safe because the seller is known and validated by the
program
    #[account(mut)]
    pub seller: AccountInfo<'info>,
    /// CHECK: This is safe because the buyer is known and validated by the program
    #[account(mut)]
    pub buyer: AccountInfo<'info>,
    // System program is required for PDA derivation
    pub system_program: Program<'info, System>,
    pub token_program: Program<'info, Token>,
}

#[derive(Accounts)]
#[instruction(order_id: String)]
pub struct DepositToEscrow<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
```

```
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut, seeds = [b"escrow", order_id.as_bytes()], bump)]
    pub escrow: Account<'info, Escrow>,
    #[account(mut)]
    pub seller: Signer<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    pub token_program: Program<'info, Token>,
    pub mint_account: Option<Account<'info, Mint>>,
    #[account(
        mut,
        associated_token::mint = mint_account,
        associated_token::authority = escrow_state
    )]
    pub escrow_state_token_account: Option<Account<'info, TokenAccount>>,
    #[account(
        mut,
        associated_token::mint = mint_account,
        associated_token::authority = seller
    )]
    pub seller_token_account: Option<Account<'info, TokenAccount>>,
    /// Associated Token Program for creating token accounts
    pub associated_token_program: Program<'info, AssociatedToken>,
    /// System Program (needed to create token accounts)
    pub system_program: Program<'info, System>,
    /// Rent sysvar (for rent exemption)
    pub rent: Sysvar<'info, Rent>,
    #[account(
        mut,
    )]
    /// CHECK: This is safe because the escrow_state_token_account is derived from
the escrow_state
    pub escrow_token_account: Option<UncheckedAccount<'info>>,
}

#[derive(Accounts)]
//#[instruction(order_id: String)]
pub struct DepositToEscrowState<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut)]
    pub seller: Signer<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    pub token_program: Program<'info, Token>,
    pub mint_account: Account<'info, Mint>,
    #[account(
        mut,
        associated_token::mint = mint_account,
        associated_token::authority = seller
    )]
    pub wallet_token_account: Account<'info, TokenAccount>,
    #[account(
        mut,
    )]
    /// CHECK: This is safe because the escrow_state_token_account is derived from
the escrow_state
    pub escrow_state_token_account: UncheckedAccount<'info>,
    /// Associated Token Program for creating token accounts
```

```
    pub associated_token_program: Program<'info, AssociatedToken>,
    /// System Program (needed to create token accounts)
    pub system_program: Program<'info, System>,
    /// Rent sysvar (for rent exemption)
    pub rent: Sysvar<'info, Rent>,
}

#[derive(Accounts)]
#[instruction(bump: u8)]
pub struct WithdrawBalance<'info> {
    #[account(mut, seeds = [b"escrow_state", seller.key().as_ref()], bump)]
    pub escrow_state: Account<'info, EscrowState>,
    #[account(mut)]
    pub seller: Signer<'info>,
    // #[account(mut, constraint = escrow_token_account.owner == TOKEN_PROGRAM_ID)]
    // pub escrow_token_account: Option<Account<'info, TokenAccount>>,
    pub token_program: Program<'info, Token>,
    // #[account(mut, constraint = seller_token_account.owner == TOKEN_PROGRAM_ID)]
    // pub seller_token_account: Option<Account<'info, TokenAccount>>,
}

#[account]
pub struct EscrowState {
    pub is_initialized: bool,
    pub seller: Pubkey,
    pub fee_bps: u64,
    pub arbitrator: Pubkey,
    pub fee_recipient: Pubkey,
    pub fee_discount_nft: Pubkey,
    pub dispute_fee: u64,
    pub deployer: Pubkey,
}

#[account]
pub struct Escrow {
    pub exists: bool,
    pub seller_can_cancel_after: i64,
    pub fee: u64,
    pub dispute: bool,
    pub partner: Pubkey,
    pub open_peer_fee: u64,
    pub automatic_escrow: bool,
    pub amount: u64,
    pub token: Pubkey,
    pub seller: Pubkey,
    pub buyer: Pubkey,
    pub seller_paid_dispute: bool,
    pub buyer_paid_dispute: bool,
}

#[error_code]
pub enum SolanaErrorCode {
    #[msg("Invalid amount")]
    InvalidAmount,
    #[msg("Invalid seller waiting time")]
    InvalidSellerWaitingTime,
    #[msg("Escrow not found")]
    EscrowNotFound,
    #[msg("Cannot open dispute yet")]
```

```
    CannotOpenDisputeYet,
    #[msg("Insufficient funds for dispute")]
    InsufficientFundsForDispute,
    #[msg("Dispute not open")]
    DisputeNotOpen,
    #[msg("Invalid winner")]
    InvalidWinner,
    #[msg("Order already exists")]
    OrderAlreadyExists,
    #[msg("Insufficient funds")]
    InsufficientFunds,
    #[msg("Invalid buyer")]
    InvalidBuyer,
    #[msg("Cannot cancel yet")]
    CannotCancelYet,
    #[msg("Serialization error")]
    SerializationError,
    #[msg("Already initialized")]
    AlreadyInitialized,
    #[msg("Cannot release funds as order is not marked as paid")]
    CannotReleaseFundsYet,
    #[msg("Invalid Fee Recepient")]
    InvalidFeeRecepient,
    #[msg("Invalid Dispute Initiator")]
    InvalidDisputeInitiator,
    #[msg("You missed to pass one important account")]
    AccountError,
}

#[event]
pub struct EscrowCreated {
    pub order_id: String,
}

#[event]
pub struct Released {
    pub order_id: String,
}

#[event]
pub struct SellerCancelDisabled {
    pub order_id: String,
}

#[event]
pub struct CancelledByBuyer {
    pub order_id: String,
}

#[event]
pub struct CancelledBySeller {
    pub order_id: String,
}

#[event]
pub struct DisputeOpened {
    pub order_id: String,
    pub sender: Pubkey,
}
```

```
#[event]
pub struct DisputeResolved {
    pub order_id: String,
    pub winner: Pubkey,
}
```

Written in Rust using the Anchor framework, the protocol allows users to securely hold funds (either in SOL or SPL tokens) until a specific condition is met, such as the completion of a trade. The main features and components of the protocol are:

Key Functions:

Initialization (initialize):
This function initializes the escrow state for a seller. It sets up parameters like the fee (in basis points), dispute fee, fee recipient, and arbitrator.

Create Escrow (create_escrow_sol and create_escrow_token):
These functions create an escrow account to hold SOL or tokens, securing the funds until both parties (buyer and seller) complete their transaction. The escrow can be created either by the seller or the buyer, and can be automatically initialized based on the provided flags. A fee is calculated and included in the escrow to cover the platform's service.

Mark as Paid (mark_as_paid):
Once the buyer makes the payment, this function allows them to mark the escrow as paid, preventing the seller from canceling the transaction.

Release Funds (release_funds):
After marking as paid, the funds can be released to the buyer or seller depending on the transaction's outcome. This function ensures that the fee is properly distributed to the fee recipient, and the correct amount is transferred to the buyer or seller.

Cancel Escrow (seller_cancel and buyer_cancel):
These functions allow the buyer or seller to cancel the escrow if specific conditions are met, such as the seller waiting time or when both parties agree to cancel the transaction.

Open Dispute (open_dispute):
If there is a disagreement between the buyer and seller, either party can open a dispute by paying a dispute fee. This action freezes the funds until the arbitrator resolves the dispute.

Resolve Dispute (resolve_dispute):
The arbitrator resolves disputes and decides which party wins, transferring the funds to the winner.

Deposit to Escrow (deposit_to_escrow and deposit_to_escrow_state):
These functions handle depositing additional funds into the escrow. It checks whether the escrow has enough funds and manages both SOL and token-based deposits.

Withdraw Balance (withdraw_balance):

This function allows the seller to withdraw any remaining balance from the escrow state if necessary.

Security and Validation:

The protocol uses several validation checks (e.g., ensuring the amount is valid, that the buyer and seller are different parties, etc.). Each function handles potential errors through the custom error codes (e.g., InvalidAmount, CannotCancelYet, DisputeNotOpen).

Fee Mechanism:

The protocol calculates a fee in basis points (bps) for every transaction, which is sent to a specified fee recipient. There is also a discount mechanism based on the ownership of an NFT (the fee_discount_nft).

Token Support:

The protocol supports both SOL and SPL token transactions, creating associated token accounts if necessary. It manages transfers between different token accounts securely.

Escrow State Management:

Each escrow instance is tied to an EscrowState account, which stores all relevant information like the seller, fees, dispute status, and arbitrator.

LocalSolana Adoption

LocalSolana isn't limited to a simple peer-to-peer trading platform like those offered by centralized exchanges today. In the long term we want to compete with the largest onramps like Moonpay and Ramp, bringing mass adoption to crypto. We believe peer-to-peer trading unlocks onchain crypto adoption in emerging markets where it is needed the most. LocalSolana will look to integrate directly with wallets, DeFi protocols, NFT marketplaces and future platforms with the mission to improve crypto onboarding for users in all emerging markets.

Self-Custody Wallets

LocalSolana will be integrated as an onramp option in self-custody wallets both on desktop and mobile. Many of the popular wallets today like Phantom and Backpack only have onramp options that are compatible with users from western markets. When using these onramps many users have their payments blocked or are unable to pass verification. For most jurisdictions onramp options are simply unavailable. LocalSolana will enable self-custody wallets to onboard users from any country globally through P2P.

Decentralized Applications, Casinos and Games

Decentralized applications, casinos and games today often have large user bases in emerging markets that are still forced to get the protocol's native token through an off platform P2P exchange like

Binance P2P before using the product. By integrating LocalSolana they can build their own native onramping experience and provide an essential P2P option for their users.

Centralized Exchanges

Almost every centralized exchange today has to launch a P2P exchange in order to onramp users in the numerous markets they operate in. Rather than having to acquire P2P merchants and liquidity these centralized exchanges can integrate or whitelabel LocalSolana as their P2P exchange.

Token Mechanisms

The LocalSolana protocol will be powered by the Peer-to-Peer ($PP) utility token following the SPL standard. The token will be used in core mechanisms within the protocol.

Buyback Mechanism

At token launch 50% of all trading fees will be used to buyback tokens from the open market and burn them. This presents an efficient way to incentivize and loyalty across a wide selection of stakeholders including users, token holders and arbitrators.

Arbitration Mechanism

The core mechanism of the LocalSolana token will be for the chance to arbitrate disputes. Token stakers can further stake their tokens for the opportunity to resolve disputes between users. Fees earned from disputes will be paid out to arbitrators in the token used to pay the fee. Currently arbitration fees need to be paid SOL. In the future arbitration fees will be able to be paid in a wider variety of popular tokens. Arbitration is further detailed in the following section.

Reputation Bonding

Money from ransomware, scams and hacks is making its way into P2P markets at an increasing pace. A common interaction is someone who hacks a bank account will trade the hacked fiat on a P2P exchange and get away with the irreversible crypto. An unsuspecting trader is unaware they’ve taken stolen funds and gets their bank account frozen days after releasing the crypto to the other counterparty. To disincentivize this behavior and to improve trust on the platform traders will be given the ability to bond tokens to their account. Bonded tokens can be claimed by another party if they provide evidence that fiat they acquired from the other trader resulted in a chargeback or their account being frozen. Claims will be processed initially by core contributors but in the future will be decentralized to token stakers who are incentivized to keep traders on the platform honest.

Marketplace Visibility

As P2P marketplaces have grown many trades are now executed by professional merchants who act as market makers on the platform. These merchants profit from the spread between buying and selling cryptocurrencies. More visibility on a marketplace means more volume and ultimately more profit for

the professional merchant. LocalSolana will grant higher visibility on the platform to merchants who stake tokens or bond their tokens for reputation purposes.

Governance

Token stakers will be governors of the LocalSolana protocol. Some key governance decisions will be fees, burn rates and which chains LocalSolana will be deployed on in the future. A simple delay mechanism will be implemented to prevent just-in-time voting. Arbitrators who have further staked their tokens can also participate in governance.

LocalSolana Arbitration

In the long term, arbitration is a key element in the LocalSolana protocol and its decentralization. At launch, disputes will be arbitrated by the LocalSolana team through a multisig wallet. However over time the arbitration will be decentralized to the community.

Arbitration Token Mechanics

LocalSolana's arbitration will be powered by the protocol token. Arbitrators have an economic interest to arbitrate decisions as they will collect dispute fees in exchange for their work. By staking tokens, arbitrators can be selected in a pool and vote on how to resolve the dispute. The probability of being selected as an arbitrator on a specific case is proportional to the amount of tokens staked. The higher the amount of tokens staked, the higher chance of being assigned as an arbitrator to a dispute and potentially earning fees. The protocol token will also be used for governance to manage the treasury and set protocol parameters like fees and available tokens.

The protocol token prevents sybil attacks with decentralized arbitration. If arbitrators were simply drawn randomly, someone could create multiple wallets to be drawn multiple times in a dispute and fully control the outcome. The protocol token also incentivizes engagement and honesty. If an arbitrator has tokens staked and doesn't participate in resolving a dispute they could lose part of their stake. Similarly if they vote incoherently with other arbitrators, they could also lose part of their stake to coherent voters.

The token and arbitration mechanism will evolve over time and may allow for delegation and fee-sharing.

Arbitrator Selection

When decentralized arbitration is launched, there will only be a single staking pool that will arbitrate all trades globally. An initial selection of arbitrators will be qualified from frequent buyers and sellers of the protocol. However as the protocol grows, arbitration will be open to all stakers and pools will eventually be divided by countries and payment networks to enable arbitrators to specialize in specific payment networks which they may be more familiar with.

Once arbitrators have staked their protocol tokens in a pool, they can be available to be selected for a dispute. Their votes in the dispute will be weighted based on the number of tokens they have staked

up to a maximum amount. Arbitration selection will also make use of random numbers drawn from blockhashes of recent blocks on Polygon or Ethereum.

Arbitration Voting

Once selected for a dispute, arbitrators will have 24 hours to vote on a resolution otherwise they risk losing part of their stake. Votes will not be public until every arbitrator has cast a vote or the time limit has been reached along with a quorum of votes. Not knowing the other arbitrator's vote is important in finding the truth as detailed in Thomas Schelling's Schelling Point, otherwise described as a focal point. A Schelling Point is each person's expectation of what the other expects him to expect to be expected to do. By incentivizing each arbitrator with fees if they vote with the majority, along with not revealing any votes until every party has voted, we can expect arbitrators to vote for the right and honest resolution.

A minimum number of arbitrators participating will be required in each dispute. If a vote fails to reach quorum new arbitrators will be drawn until enough votes have been cast. The minimum number of arbitrators will be decided by a governance vote and is expected to grow as the protocol is increasingly adopted.

Protecting Against Attacks

There are two possible attacks that could be undertaken by a bad actor. The first is controlling the majority of the liquid token supply. This is unlikely as if the protocol grows popular enough where an attack would be attractive, an attacker would need to purchase a significant amount of protocol token driving up the price and losing any economic benefit. LocalSolana may choose to limit transaction sizes based on protocol token liquidity and price in order to reduce the economic incentive for this type of attack.

Another possible attack is to bribe a voting majority of arbitrators. By bribing 51% of arbitrators a bad actor could dictate the outcome of a vote. LocalSolana may introduce an appeals mechanism where arbitrators will be redrawn and the minimum number of voters will double in size. The bad actor would then need to effectively bribe three times the number of arbitrators drastically reducing their economic incentive.

Conclusion

We have briefly introduced LocalSolana, a decentralized peer-to-peer (P2P) protocol for people to exchange crypto assets like SOL and USDT (Tether) for fiat money without centralized counterparty risk through escrow smart contracts. The protocol is governed by token holders and disputes are arbitrated by a decentralized community of token stakers.

The sustained currency depreciation seen through many developing countries without stable financial systems has fueled significant crypto adoption. As government and banking restrictions on the industry and participants in these countries increase, centralized providers will be unable to provide on and offramp services like they do in developed countries, leaving participants without a reliable bridge between crypto and fiat. As crypto adoption accelerates it's essential that resilient P2P exchanges exist through decentralized protocols.