Conduct Protocol Design

Motivation

We observe the following obstacles to web3's success:

- Most blockchain projects are only partially decentralized. This strategy embraces the disadvantages of centralization while foregoing its benefits. Namely, users must suffer potential resiliency and censorship issues without realizing any benefits to speed or throughput.
- Blockchain projects that may have been designed for decentralization ultimately end up centralized due to the inconvenience or complexity of participating. More specifically, operating a blockchain node is hard, even for technical users.
- Node operation aside, everyday usage of blockchain technology faces major UX challenges, rendering the technology relatively useless to most global users.

Solution

The entirety of our decentralized blockchain protocol runs within a single mobile wallet app. The mobile app will be distributed through the Google Play App Store and Apple App Store to facilitate seamless adoption and upgrades of the protocol and user experience. This mobile application performs the following duties:

- Conduct wallet operations
- Bitcoin wallet operations
- Staking
- Full node operation

A mobile phone, at its core, is a computer. Today's average phone is more powerful than yesterdecade's average desktop. People carry them everywhere they go, and they're always online and connected. They include a variety of sensors, cameras, a screen, a battery, communication radios, and biometric systems. They're the near-perfect medium for decentralization.

On the surface, the application looks like a web3 wallet with a few extra buttons and features. Under the hood, the application is validating, sharing, and extending the blockchain just like today's server-based approaches do.

Architecture

Mobile App

Our primary product is the mobile application which runs on Android and iOS. A majority of its logic is written in Rust, while the application is *presented* using Flutter. The Rust code is loaded into Flutter using a tool called Flutter Rust Bridge. These bindings allow Dart code to interface with the blockchain logic written in Rust. In particular, Dart manages the Rust code and has access to all of the blockchain data.

Internally, the blockchain node handles all of the same operations you would expect from nodes that run on servers: consensus, ledger management, block validation, P2P networking, RPC communication, and block production.

Blockchain data is persisted using the combination of a SQLite database and a distributed hash table. Despite improvements in CPU and memory performance, mobile devices have somewhat stagnated in their disk storage space. A single mobile device is unlikely to have enough disk space to store the entire chain. Instead, the data is partitioned such that each individual node only holds a subset of the overall data. Any missing data is retrieved through the peer-to-peer network.

In addition to acting as a blockchain full node, the app also provides wallet functionality for both Bitcoin and Conduct, offering a seamless experience for the staking and rewards system described later.

Key Management

Decentralization generally involves each participant possessing certain “secret keys”. These keys allow an individual to sign information in a way that others can verify. The typical use-case is transaction signatures. Each device is currently responsible for the following secret keys:

- A Conduct wallet secret key (secp256k1), to sign Conduct transactions
- A Bitcoin wallet secret key (secp256k1), to sign Bitcoin transactions
- A P2P secret key (secp256k1), to sign messages in the network
- A VRF secret key (secp256k1), to sign new Conduct blocks

These keys are stored in a hardware-backed, encrypted storage, namely Android Keystore and Apple Secure Enclave. When a message must be signed, it is read from the encrypted storage, used, and then cleared from memory immediately.

The Conduct wallet secret key, Bitcoin wallet secret key, and P2P secret key are derived from a BIP-39 mnemonic which makes them all recoverable. A VRF key is not derived from this mnemonic and is thus non-recoverable.

While the Bitcoin secret key must use secp256k1, we have the option of using different routines in Conduct should we choose. One alternative is secp256r1 which would enable more direct hardware-based signing without the application accessing secret keys. For increased security, we could allow users to opt-into this process for the Conduct wallet secret key at the risk of making the key non-recoverable.

Server-Based Nodes

The desire to run mobile-based nodes comes primarily from the strategy of having every wallet also be a node, but there's no real harm in allowing nodes to operate in a server-based environment as well. Server-based nodes achieve better uptime, better performance, and more storage, all of which will help improve the resiliency of reliability of the network, especially early on. These nodes **can** operate as stakers but with added complexity. Server-based nodes are generally dreadful to operate and manage, so we don't expect the average user to use them. Rather, Conduct (the business) will operate many of these nodes internally. We will still offer this functionality to users who wish to use it though.

Messenger Server

While the mobile application handles all logic pertaining to the protocol, cell carriers (i.e. AT&T, Verizon, T-Mobile, etc.) generally prohibit inbound TCP or UDP traffic. To overcome this limitation, we provide a separate transport layer which delivers messages between peers, similar to a TURN server. This system operates on cloud-hosted servers to provide a fast and stable communication layer between mobile peers. Each mobile peer has its own Peer ID (a public key). Peers construct signed envelopes which are addressed to other peers. The Messenger Server receives these envelopes and attempts to deliver them to the proper recipients.

P2P communication typically initializes through a messenger server, but shortly after the handshake process, the two peers will attempt direct encrypted UDP communication with hole punching facilitated by messenger servers. If direct communication can't be established, communication will simply flow through the messenger servers by default.

The Messenger Server is a centralized service and is only responsible for delivering messages from one connected peer to another connected peer. It does not have its own separate peer-to-peer layer to communicate and relay messages to other Messenger Servers. Decentralization happens on the client-side, as described below in the P2P section.

Decentralized Protocol

Conduct Protocol is a blockchain. Like other blockchains, it provides immutability and agreement on the order and validity of user transactions. A blockchain uses a transactional

ledger to facilitate financial and data exchanges between users, while a decentralized consensus mechanism keeps participants honest and on the same page.

The “genesis” (first) block of the chain includes an initial token distribution for founders, investors, and business-related expenses. As blocks are produced, new tokens are minted at an ever-decreasing rate.

Consensus

NOTE: The numbers and parameters described below are subject to change as we find the right balance between economic security, protocol security, user experience, and performance.

Our approach to consensus is loosely inspired by Ouroboros, which uses proof-of-stake to grant block production permission. Rather than relying solely on the native token for staking, Conduct incorporates Bitcoin tokens in the stake calculation. This allows us to leverage Bitcoin’s work to help secure our energy efficient sidechain. It also encourages usage of Bitcoin and rewards Bitcoin miners for their efforts.

The blockchain’s internal clock starts at 0 and uses an incrementing unit of time called a “slot” (currently set to 1-second). These slots further make up a larger unit of time called an “epoch”, currently set to 1 day. These epochs allow the protocol to summarize events over a stable window of time, which informs the decision-making for subsequent epochs.

Blockchain blocks are split into three parts:

- A Block Header: Lightweight meta-information about the block
- A Block Body: A list of transaction IDs included in the block
- A Consensus Body: A list of transaction IDs (also included in the block body) of transactions that influence consensus operations, such as staker registrations, staker exits, and misbehavior proofs.

A Block Header includes the following fields:

- **parent_block_id**: The ID of the block previous to this one
- **body_merkle_root**: The merkle root of a tree where each leaf is a Transaction ID within the block body
- **attestation_merkle_root**: The merkle root of a tree where each leaf is the Attestation ID of the corresponding transaction within the block body
- **consensus_merkle_root**: The merkle root of a tree where each leaf is a Transaction ID within the block body, but only transactions that are relevant to consensus
- **timestamp**: The time at which the block was created (unix epoch MS)
- **height**: The number of blocks that have been created up to this point (the Genesis block has height 1)
- **slot**: The protocol-level clock time-step at which the staker was eligible
- **vrf_vk**: The secp256k1 verification key of the staker
- **vrf_proof**: The VRF proof which informs their eligibility at the particular slot

- **block_signature**: The secp256k1 signature of the block's bytes
- **eta**: The current randomness value

Registration and Initialization

The first epoch is bootstrapped using a hardcoded set of genesis stakers. All subsequent epochs follow the normal registration path. Typical registration requires a Bitcoin transaction followed by a Conduct transaction; the former provides the initial BTC fee and the latter provides the initial CNDT stake for the account. Before registering, a staker generates a secret key that is used for block signing and VRF proofs.

Bitcoin Registration Transaction

A staker must submit a transaction to the Bitcoin network that includes a 0-value `OP_RETURN` at output index 0. The rest of the transaction can be for any purpose; the paid transaction fee influences the staker's weight. The OP_RETURN output stores a commitment hash. The data included in the hash's pre-image is:

- The staker's VRF public key
- A Conduct block ID created within the last 6-24 hours
- A Conduct reward address
- A random nonce

Conduct Registration Transaction

Once the Bitcoin Registration Transaction has been confirmed to a depth of at least 6 Bitcoin blocks, the staker submits a Conduct transaction to the Conduct chain. This transaction includes an output which uses the `staking_registration` field and provides the following:

- The Bitcoin transaction ID
- The Bitcoin height which included the transaction
- The Bitcoin merkle proof for the transaction
- The Conduct reward address
- The staker's VRF public key
- The Conduct anchor block ID
- The random nonce

Nodes which validate this transaction will perform simple payment verification of the claimed Bitcoin information.

The staker who produced the block which included this new registration transaction will include its ID in the "Consensus Body" of the block.

Incremental Staking

This process can be repeated by the staker to increase their voting power over time by creating additional Staking Registrations which reference the same VRF key. Fees paid on BTC transactions accumulate in the staker's account, and the CNDT that is used to create the Staking Registrations also accumulates in the account. Spending a Staking Registration will reduce the voting power of the staker.

Staking and Eligibility

Generally, proof-of-stake protocols use a native token to establish staking rights, usually by taking the quantity of staker's tokens relative to other stakers. Conduct's stake is derived from the native CNDT token and BTC transaction fees. An individual staker may add to each value over time. An individual's stake is defined as:

`stake = staked_cndt * (minimum(btc_fees, 20_000_000) + 130_000_000) / 130_000_000`

Our intention is to incentivize regular usage of Bitcoin and some competition in the BTC transaction fee market. To avoid abuse from miners or misalignment between the BTC and CNDT token values, the influence of the BTC fee multiplier ranges from 1x to 1.15x, with the 1.15x boost achieved at a cumulative BTC fee of 0.2 BTC.

Proof-of-Stake consensus eligibility generally favors those with more stake. Rather than granting dominance to the staker with the most stake, the stakers participate in a lottery. A fair lottery should incorporate randomness so that the same lottery ticket doesn't get pulled every time. Sourcing randomness in proof-of-stake consensus is difficult. Rather than relying on the native chain like in Ouroboros, we instead treat the Bitcoin canonical chain as our random beacon. Since future Bitcoin block headers can't be predicted, this prevents biasing the lottery tickets toward favorable outcomes for particular stakers. For simplicity, we still refer to this Bitcoin block ID as the "eta" value for parity with Ouroboros.

At each slot, a staker uses a Verifiable Random Function (VRF) and their proportional stake to determine if they are eligible to produce a block. Each staker has a low likelihood of producing a block at a given slot, but there's a chance that some staker might. There's also the chance that no stakers are eligible for a given slot, so it is left empty. We currently tune the protocol such that someone should be eligible in 1 out of every 20 slots on average (the `f_effective` parameter). This roughly approximates to a new block every 20 seconds.

The following function defines the eligibility for a staker at a particular slot:

```
pub fn is_eligible(
    // The proportion of desired blocks to slots
    f_effective: BigRational,
    // The slot of the parent block
    parent_slot: u64,
    // The ID/hash of a canonical Bitcoin header
    bitcoin_anchor_id: &BitcoinBlockId,
```

```
    // The slot being tested for eligibility
    slot: u64,
    // The staker’s VRF proof for the slot
    rho: &Rho,
    // The staker’s proportion of stake relative to other stakers
    relative_stake: &BigRational,
) -> bool {
    if parent_slot >= slot {
        return false;
    }

    // Establish this staker's "score" at this particular slot, which is a
pseudo-random number
    let message = merge_slices(&[&bitcoin_anchor_id.value, &rho.test_hash()]);
    // The number is represented as a 256-bit hash
    let m = sha256(&message);
    // Convert the hash into a 256-bit integer
    let m_bigint = BigInt::from_bytes_be(Sign::Plus, &m);
    let score = BigRational::new(m_bigint, TWO_256.clone());
    // The number of slots by which the chain has slowed from its target block
production rate
    let delayed_by_slots = slot - parent_slot - f_effective.recip();
    // The longer it has been since a block was produced, the easier it becomes to
make a block
    let scale =
        (f_effective.clone() +
BigRational::one()).pow(delayed_by_slots.try_into().unwrap());
    // This particular staker's score needs to be below this value
    let maximum_score = relative_stake.clone() * f_effective.clone() * scale;
    score < maximum_score
}
```

When all stakers are online, this would theoretically result in slightly faster than 20 seconds per block. Practically speaking, in a mobile-operated network, individual uptimes may be quite low, so the eligibility difficulty decreases if block production appears to slow (the `scale` value in the above function).

Reward Distribution and De-Registration

Registering to stake also initializes a “reward account”. This reward account tracks two values:

- Accumulated block rewards
- Accumulated transaction fees

Producing a new block, even an empty one, yields a block reward. This block reward follows a decay schedule similar to Bitcoin’s. The reward for each block remains fixed over a 4-year period, at which point the reward-per-block is cut in half. We target 33 total reward periods, after

which the block reward is reduced to 0 and no further native tokens will ever be created. The total block reward supply is approximately 14,000,000 CNDT or 1,400,000,000,000,000 condoshis.
The calculation for the reward at each block is defined as:

```
halving_schedule = 4 years = 126,230,400 slots ~= 6,311,520 blocks
total_halvings = 32 (33 periods total)

R0: Reward period=0
reward_supply = R0 * halving_schedule * (2 - (1 / (2 ^ total_halvings)))
R0 = reward_supply / (halving_schedule * (2 - (1 / (2 ^ total_halvings))))
R0 = 110,908,307 condoshis

block_reward(height) = R0 x (1/2) ^ floor(height / halving_schedule)
```

In addition to the base block reward, any fees paid by transactions in their block are also collected by the staker and added to their reward account.

A staker must first de-register in order to collect their rewards. De-registration takes place by spending the original registration UTxO. The de-registration transaction includes a single `output` with a quantity matching the sum total of their reward account and an address matching the reward address committed to in their registration.

The ID of this transaction is included in the “Consensus Body”.

Misbehavior

Like with other proof-of-stake protocols, we have the ability to slash native tokens when a staker misbehaves, but our self-custodial approach to remote staking comes with the consequence of not being able to slash BTC. But by withholding all rewards until the timelock elapses, the protocol maintains the ability to detect and ban a misbehaving or non-participating staker’s VRF key. Once a VRF key is banned, it is no longer able to produce new blocks, thus no longer able to collect its reward.

Managing this ban is not currently implemented. Transactions are able to record evidence of a staker’s misbehavior, but these transactions will be rejected until proper mechanisms exist to enforce them. If sufficient evidence is provided, the misbehaving staker’s reward account will be forfeit and their timelocked CNDT slashed.

While not yet implemented, the currently planned penalizable offenses are:

- Signing an invalid block: If a block contains invalid information or transactions, and the staker signs it, then other nodes can check the invalidity by inspecting the block.
    - Exception: Signing a block with an oversized transaction

- Signing the two different headers with the same slot. The existence of the two signed headers is proof of misbehavior.

Long-Range Attacks

Proof-of-stake protocols are susceptible to a situation where a majority of stakeholders could pick an earlier point in time and rewrite history. The protocol is generally protected from non-extreme long-range attacks by the bi-directional anchoring of the chain histories. Requiring Conduct history commitments in Bitcoin staking registration transactions prevents a staker from attempting to re-use their registrations on an alternative Conduct history. Leveraging the canonical Bitcoin chain as our source of randomness prevents biasing the eligibility system on an alternative history as well.

An extreme long-range attack is when an attacker discretely registers a dominant stake and produces a rival chain without telling anyone. The mobile-oriented nature of our product enables several options with social trust. If one friend trusts another, they can simply scan a QR code to establish a checkpoint. By building a web of trust, users can protect themselves from long-range attacks by only syncing with the chain that is trusted by their friends. Furthermore, our centralized delivery channels allow us to hardcode default lists of trusted peer IDs and checkpoints.

Chain Selection

Our consensus is non-determinstic, meaning multiple stakers may technically be eligible within the same slot. This may result in multiple valid extensions of the chain existing at once. In a globally-distributed P2P network, chain extensions may not immediately be observed by some peers. This leads to natural short-term forks near the tip of the chain. While the non-determinstic forks may contain valid blocks, a deterministic process must be applied so that peers can properly agree on a consistent chain.

Generally speaking, we select the chain which is longest (has the greatest height). If two competing chains have equal length, the chain tip with the earliest slot is selected. If the competing chains both share the same slot, the final tie-breaker compares the `rho + “TEST”` values of the two chains.

Unlike Ouroboros, we do not need a density chain selection rule since Bitcoin acts as our (unbiasable) randomness beacon.

Ledger

Conduct uses an Unspent Transaction Output (UTxO) model for its ledger, similar to Bitcoin. Each UTxO contains a quantity as well as the hash of the script that must be satisfied in order to spend it. A Transaction contains a list of references to UTxOs to spend, authorizations to spend those UTxOs, and a list of new UTxOs to create. Typically, authorization is performed through knowledge of a private key which allows individuals to “own” their UTxOs.

In addition to a simple quantity, a UTxO may embed a label, JSON data, an asset, or references to other UTxOs. To mitigate spam, the minimum quantity for this UTxO scales with the amount of data that is included.

Furthermore, the scripting language which locks these UTxOs can enable complex contracts, even contracts requiring Bitcoin data.

The ability to label and embed data allows dApp developers to store application data on-chain. This data can be searched by dApps, which gives the general experience of a decentralized document database. By also allowing references to other UTxOs, a graph-like data structure manifests which allows for connections and associations between data points, potentially between different dApps or use-cases.

To support asset and NFT use-cases, a UTxO may contain sub-tokens with their own separate quantities. The general workflow is to describe the asset using the label and data mechanism, and then consume the data UTxO to create an asset supply token. The asset supply token allows minting of new instances of the sub-token. Each asset UTxO contains a reference back to the origin data UTxO to support efficient querying of assets.

Due to our tight relationship with Bitcoin, we also support first-party atomic swaps between the two chains. A user can list a SwapOrder to sell CNDT at a desired BTC price. Another user can claim the SwapOrder with a SwapInProgress to prevent duplicate claims. After sending the BTC, the user then claims the CNDT tokens using Conduct's built-in SPV (see below).

Script

A UTxO is encumbered by an "address", which is the sha256 hash of a script. To spend the UTxO, authorization must be provided in the form of a "Witness" which contains the address, the script, and the arguments to provide to the script. Currently, there is only one type of script: Expr. A script has a type prefix, so more scripts could be implemented in the future.

Expr

Expr (short for "expression") is a declarative, functional DSL that can be represented as bytecode. It is not Turing Complete, and a core goal is the ability to statically analyze the worst-case runtime computational cost of an expression.

An expression can be one of:

- Literal: string, integer, float, boolean, byte array, array, or map
- Apply: Invokes a host-defined function with an array of expression arguments
- Let: Evaluates an expression, inserts it into the scope by a provided name, and evaluates another expression with the new scope.

- Reference: Summons a value by name from the current scope

An expression evaluates to a Value. That value can be of one of the following types:

- String
- Integer (signed, 64-bit)
- Float (signed, 64-bit)
- Boolean
- ByteArray
- Array<Value>
- Map<String, Value>
- Host Object (Transaction, Witness, Transaction Output, Transaction Output Reference, Bitcoin Header, Bitcoin Transaction, Bitcoin Transaction Output)

The available host functions are:

- IfThenElse: Evaluates arg0 as a Boolean, and if true, returns the evaluation of arg1, otherwise returns the evaluation of arg2
- Not: Returns the negation of arg0 as a Boolean
- All: Returns true if all provided arguments evaluate to true
- Any: Returns true if any of the provided arguments evaluates to true
- Threshold: Returns true if the threshold number at arg0 is satisfied by the remaining arguments
- SignableBytesOf: Returns a ByteArray containing the signable message of the transaction at arg0
- AttestationOf: Returns an Array of Witness host objects of the transaction at arg0
- Get: Accepts an Array or Map at arg0, and retrieves a value by index or key located at arg1. If arg1 is omitted, the “head” element is returned.
- ArgsOf: Returns a Map representing the user-provided arguments of the witness at arg0
- CurrentTransaction: Returns the current transaction being verified
- CurrentWitness: Returns the current witness being verified
- Equals: Returns true if arg0 equals arg1
- GreaterThan: Returns true if arg0 is greater than arg1
- GreaterThanOrEqual: Returns true if arg0 is greater than or equal to arg1
- LessThan: Returns true if arg0 is less than arg1
- LessThanOrEqual: Returns true if arg0 is less than or equal to arg1
- Add: Returns the sum of arg0 and arg1
- Subtract: Returns arg0 minus arg1
- Multiply: Returns arg0 multiplied by arg1
- Divide: Returns arg0 divided by arg1
- Modulo: Returns arg0 modulus arg1
- Timestamp: Returns the timestamp from the evaluation context (i.e. from the block header)
- Height: Returns the height of the chain from the evaluation context
- VerifySignature: Verifies that the message defined at arg0 aligns with the signature defined at arg1 from the verification key at arg2.

- Base58Decode: Decodes the base58-encoded String at arg0
- Sha256: Returns the ByteArray sha256 hash of the ByteArray at arg0
- FoldCurrentTransactionOutputs: Evaluates an expression at arg1 for each TransactionOutput of the current transaction, with an initial value at arg0. Each time arg1 is evaluated, the scope includes “acc” and “item” values based on the current step of the process.
- OutputsOf: Returns an Array of TransactionOutput host objects from the transaction at arg0
- QuantityOf: Returns the quantity of the TransactionOutput host object at arg0
- AddressOf: Returns the address of the TransactionOutput host object at arg1
- GetBitcoinHeader: Returns the BitcoinHeader host object by height at arg0
- VerifyBtcPayment: Verify that Bitcoin transaction ID at arg0 exists in BitcoinHeader host object at arg1 using merkle proof at arg2
- GetBitcoinTransaction: Returns the BitcoinTransaction host object by ID at arg0
- IdOf: Returns a byte array (length 32) of the current Conduct transaction’s ID
- Concat: Returns the merged contents of byte array at arg0 and byte array at arg1
- AgeOf: Returns the duration in milliseconds since the TransactionOutputReference host object at arg1 was created
- CurrentInput: Returns a TransactionOutputReference host object for the TransactionInput currently being evaluated/spent

In the future, we may also introduce a Zero Knowledge verification function, which enables on-chain verification of an off-chain computation.

Fees

Each transaction must include a fee that is proportional to its size and complexity. This fee is represented by `sum(inputs) - sum(outputs)`. A minimum fee must be satisfied, but premium fees are permitted as a means of “tipping”. The block producer’s reward account accumulates these fees.

Storage and Persistence

Application settings are stored in a SQLite database with a single table: `settings`. It is a table with String keys and String values. The primary purpose is to store information about known messenger servers and the currently selected blockchain/genesis.

Most raw blockchain data is stored using the Distributed Hash Table described below. Specifically, block headers, block bodies, consensus block bodies, and transactions are stored in the DHT. The “Local” Hash Table stores a subset of the total blockchain data and is implemented using Fjall. Data is not stored directly by its key, rather, the key’s distance is measured from the peer’s key, and the distance is stored instead. When the local hash table exceeds a certain capacity, keys with the greatest distance are evicted first.

Beyond raw blockchain data, certain indexed and stateful data is needed for the execution of the protocol. This data is stored in a SQLite database.
The following tables are used:

- `headers`: Stores a subset of header data
- `transactions`: Tracks the height at which each transaction was included in the canonical chain
- `transaction_outputs`: Tracks the spendability of transaction outputs
- `staking_meta`: A single-row table which stores the current state of the staker system
- `staker_stakes`: Tracks the stake of each active staker
- `staker_rewards`: Tracks the cumulative reward of each active staker
- `bss`: Stores the current Block ID corresponding to each “Block Sourced State”

Bitcoin block headers are persisted in a single local file. Bitcoin transactions are retrieved as needed and cached in-memory; they are not currently persisted.

Secret keys, including the wallet key, the P2P key, and the staking VRF key, are all stored in the platform-dependent keyrings. The wallet keys are used infrequently, so they are only pulled into application memory briefly when needed and then erased from memory when the signing operation is complete. P2P and staking VRF keys are used relatively frequently, so they are both held in-memory for the lifetime of the application.

P2P

To support decentralization, blockchain information must be shared between users. This exchange of information takes place using a peer-to-peer network. Each peer connects to several other peers. As new blocks are created or discovered, their IDs are gossiped to other peers. Upon hearing about a new block ID, the peer requests the associated block data and its transactions, and validates the block. A similar process takes place for new transactions. As users create and sign new transactions, they broadcast them to other peers. New transactions are requested and added to the local mempool where they await inclusion in a block.

Messenger Integration

Due to the restrictions on inbound network traffic from mobile carriers, communication takes place using discrete messages rather than data streams. Messages are transported between peers using Messenger Servers. Because Messenger Servers are not P2P networked themselves, it is the responsibility of the clients to share and gossip known Messenger Servers in an effort to establish client-side determinism in Messenger Server selection.

Each Client and Messenger Server has a unique Peer ID (a public key), and for the sake of explanation, this ID can be thought of as a number between 0-100. A client maintains

persistent/long-lived connections to known Messenger Servers with similar Peer IDs. For example, client #42 will always stay connected to servers #43, #40, and #39 so that other peers always know where to find them. If client #26 wishes to send a message to client #42, it should first connect to server #43 and attempt to send a message.

Each client maintains a list of known Messenger Servers (their IP addresses and their peer IDs). Clients gossip their lists with other clients to help facilitate the decentralization effort.

Message Patterns

Messages generally follow a request-response model with the exception of block ID and transaction ID gossip notifications. IDs that are not known locally can then be requested from the remote peer. The following request-response channels are used: ping, get DHT data, and get block ID at height. All of these requests contain a request ID, which allows responses to be delivered out-of-order. They also include a timestamp-based session ID to allow tracking of re-connections.

Distributed Hash Table

Each individual mobile device may only be able to manage a subset of the chain's information, but the combined efforts of multiple devices can handle the total workload. The Peer ID mentioned above helps determine the burden of responsibility for that peer. A transaction ID can also be thought of as a number, and it falls in the same range as a peer ID (i.e. 0-100). By combining them together, we can assign each peer to a range of numbers for which it is responsible. If a peer sees a transaction with an ID that falls within its assigned range, it is expected to validate and save that transaction.

While each person might not validate every transaction, every transaction will be validated by at least someone. This should generally satisfy the security needs for non-stakers.

The implementation plan is to migrate certain data storage responsibilities out of the SQLite database and into a separate key-value persistent storage (the "hash table"). Specifically, the following will be moved into the hash table:

- Block Headers
    - Key: Block ID
    - Value: Consensus-encoded byte representation of the header
    - Note: Some header information will still be stored in SQLite for fast indexed retrieval
- Block Bodies and Consensus Block Bodies
    - Key: hash(Merkle root, "merkle")
    - Value: Consensus-encoded byte representation of the list of transaction IDs
- Transactions
    - Key: Transaction ID
    - Value: Consensus-encoded byte representation of the transaction

All of these values will be stored in the same table/keyspace. Data can be saved or retrieved by key.

Users will configure the maximum size of this table, for example 1GB. As data is added to the table, the table's total size is tracked. When new data is added that would exceed the maximum size, items are deleted in reverse-order by their "key distance" from the peer's ID. To facilitate this process, the keys are transformed into the "key distance" before being saved in the table. When data is retrieved, the requested key is transformed into the "key distance" to find the correct entry.

When the device/app needs access to a particular value from the hash table, the local table is first checked. If the value is not found, the network is searched by selecting the connected peer with an ID closest to the desired data key and requesting the value from them. If not found, the next closest peer is selected. This repeats a third time before the search is abandoned.

Social Trust

Stakers, on the other hand, need to sign off on the transactions included on their blocks. They wouldn't want to risk including an unvalidated transaction. An unfortunate side-effect is some transactions might be held in the mempool for longer periods of time if there are no eligible stakers who are responsible for verifying their particular transaction.

To mitigate this potential risk (among others), we want to leverage "social trust". While "trust" is a scary concept in a decentralized context, it's generally only scary when done anonymously. It's not so scary in the context of known individuals. People already trust their friends and family with their lives, so it's not unrealistic to trust those same people with an app. Our unique position as a mobile-first application could enable easy management of "trusted friends" by simple QR codes. With more friends comes more implicitly trusted validation of transactions. With more friends comes less work on each individual mobile device.

In theory, "social trust" could be taken to the extreme such that each device only connects to other trusted devices. The "6 degrees of separation" concept should allow information to propagate to every point in the network within 6 hops, though in practice, adding a few untrusted connections should speed things up.

UDP Communication

Indirect P2P communication is facilitated by Messenger Servers, but doing so imposes latency and increases the work carried out by each server. While cell carriers may impose restrictions on inbound traffic, many other environments support "UDP hole punching". Each peer binds a local UDP socket which can interact with Messenger Servers for hole punching while also receiving encrypted messages from other peers.

Upon launch and periodically over the lifetime of the application, the node will ping the messenger server to update the socket address it advertises to other peers. New P2P

connections are established via a Messenger Server, and once connected, the two peers perform a handshake which generates ephemeral encryption keys. Using a diffie-helman key exchange, the peers agree on a shared secret which encrypts all UDP traffic for their session. After the handshake, the peers notify each other of their “public” UDP address, at which point the two peers attempt the actual hole punching using UDP messages. If an encrypted UDP message is successfully received, subsequent messages will be sent back over UDP.

Block-Sourced State

Blockchains are considered “immutable” by many which is generally true for a specific moment in time, but certain aspects of a live-running node are necessarily stateful and mutable.

A transaction is immutable; changing its contents would change its identity, thus no longer the same transaction. A transaction output’s spendability may change depending on the context of the chain though. As new blocks are appended, the state of a transaction output might change from “unspent” to “spent”. It is necessary to track this state in order to prevent a subsequent transaction from spending the same thing.

In the case of non-deterministic consensus, there’s a chance that two valid but conflicting extensions of the chain temporarily exist at the same time. Consensus rules allow everyone to eventually agree on the same version, but nodes that initially chose the wrong version may need to “roll back” their state. When this happens, the state of a transaction output might revert back to “unspent”.

A block “chain” refers to one particular sequence of blocks, but alternative histories temporarily form through what are called “uncle” blocks. Temporary branches form off these uncles in what resembles a tree-like structure. The leaves of these branches each have their own deterministic state. This “Block Tree” concept is included in the protocol as a means of providing traversal paths between branches of the tree. This traversal path helps inform the operations that are required to “roll back” to the trunk of the tree, followed by the operations required to apply forward from that point.

The current “stateful” operations of the protocol are:

- Stakers: Tracks the stakers and their stakes for consensus purposes
- Staker Rewards: Trackers staker block reward and accumulated fees for reward distribution
- TxO State: Tracks the spendability of transaction outputs
- Order Book: Tracks the currently open Swap Orders

When needed, the current state can be requested for a particular block ID. For example, when validating for double-spends in a new block, the state of “TxO State” is retrieved at the new block’s parent ID. The new state is calculated by determining the path through the block tree, and then performing the necessary unapply and apply steps based on the transactions of each block.

Bitcoin Light Node

Each Conduct node also acts as a Bitcoin light node. Nodes record and verify the contents of each Bitcoin block header, starting from a hardcoded checkpoint (before which no Conduct data exists).

Bitcoin data is retrieved using JSON-RPC via Esplora, and the app is currently configured to pull data from Sandshrew.

Upon startup, the Conduct node launches a background sync process which fetches and validates Bitcoin block headers. Each header is validated for the expected difficulty. If valid, it is appended to a single persistent file. Each Bitcoin header is exactly 80 bytes, so the underlying file's size will always be a multiple of 80. The node can retrieve a header by height by seeking the file at a specific index.

The syncing process only tracks blocks with a depth of 3+ blocks to avoid the risk of encountering a fork in the Bitcoin network.

Each Conduct header also embeds a particular Bitcoin block ID, so Bitcoin data availability is critical.

When performing Simple Payment Verification, the user provides a Bitcoin block height, a Bitcoin transaction ID, and a Bitcoin merkle proof. The corresponding Bitcoin header is retrieved by height, and the transaction ID and merkle proof are validated against the header's merkle root. Once satisfied, the transaction is retrieved via Esplora and further validation can take place depending on the use-case.