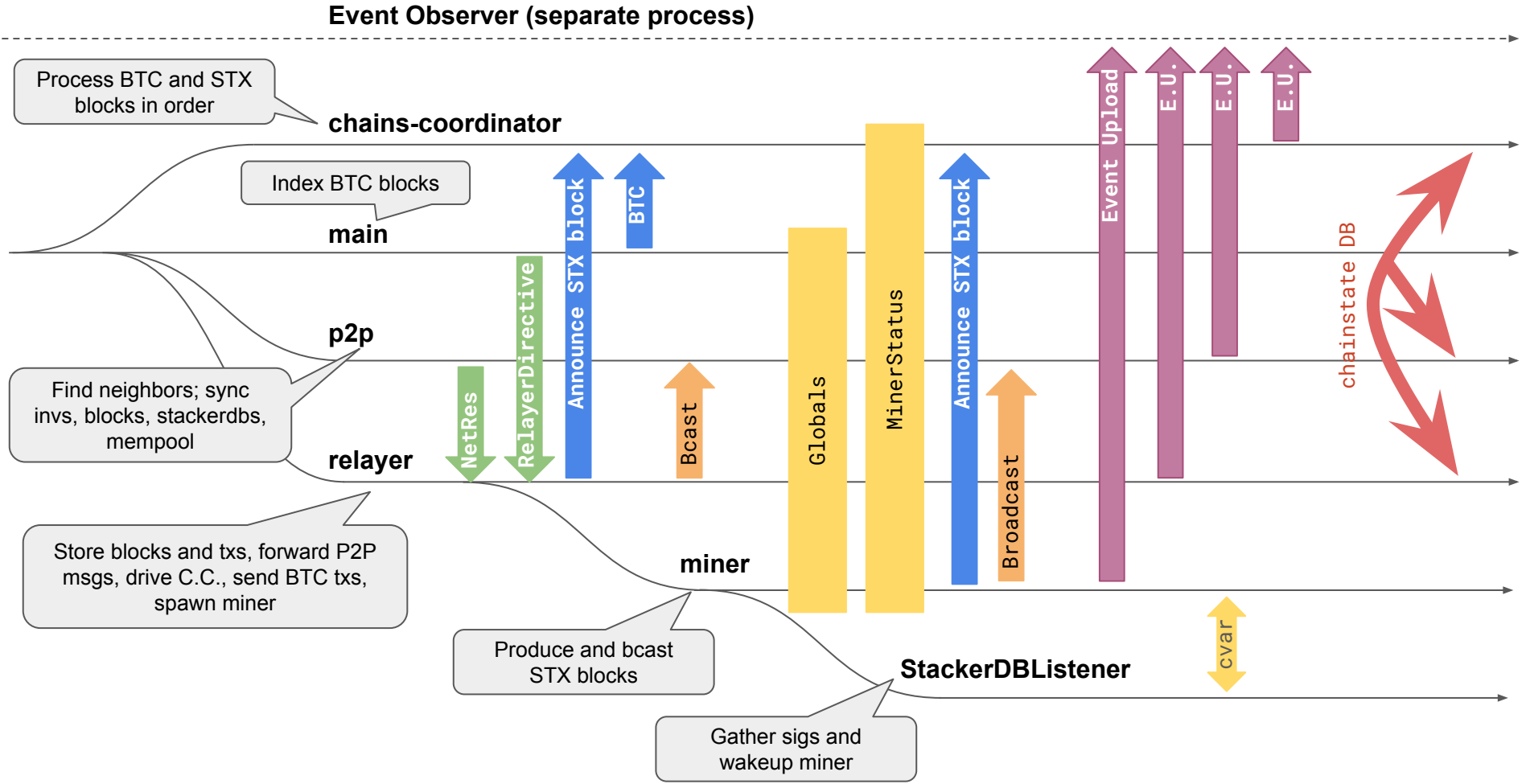
Stacks Academy

How To Succeed

- Platform, not Product
- Initiative and Humility
- Pre-reading
- Specific questions

Q: A high-level overview of the components?

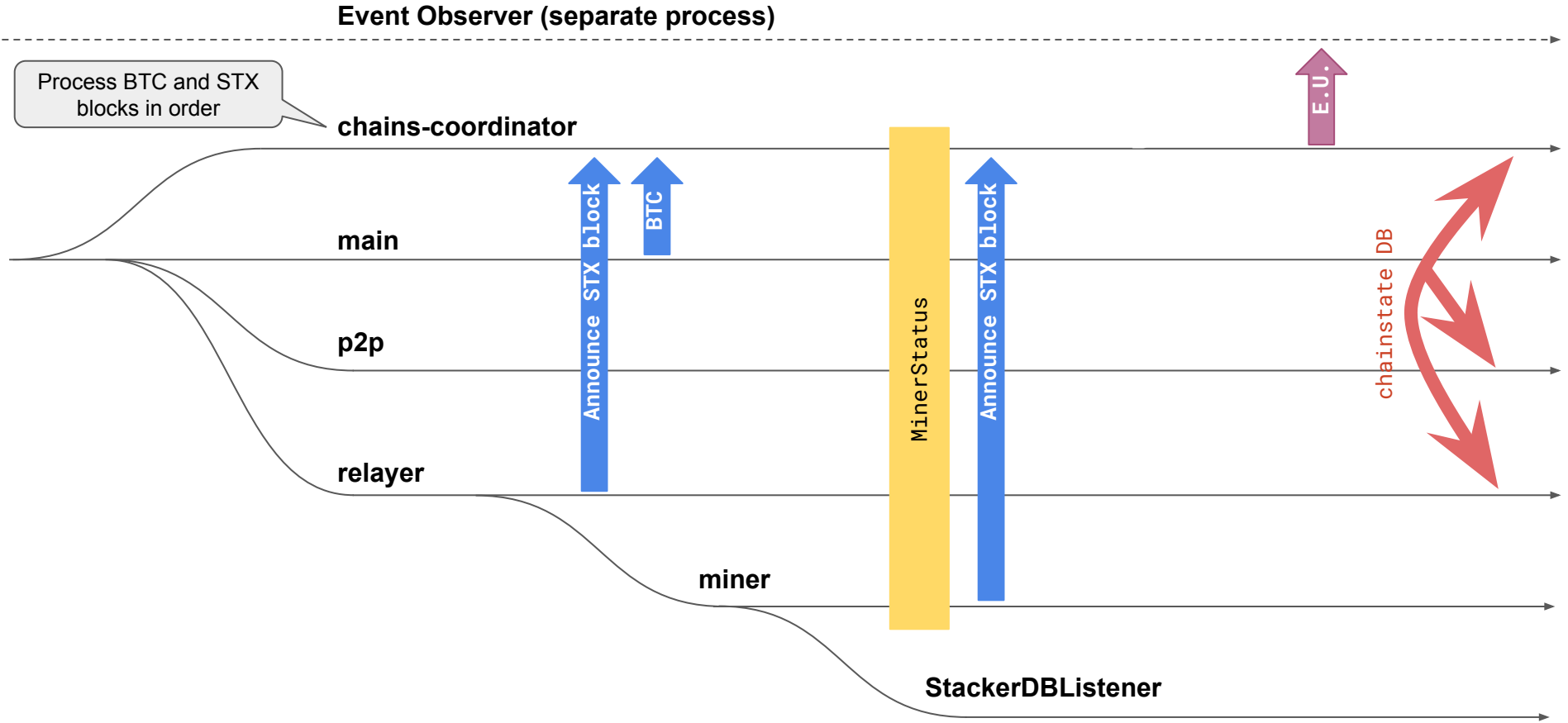
- what they do,
- how they cooperate,
- compete (eg, locks on the same resources),
- how they have evolved (if relevant)


Event Observer (separate process)
Process BTC and STX blocks in order
chains-coordinator
Index BTC blocks
main
p2p
Find neighbors; sync invs, blocks, stackerdbs, mempool
relayer
Store blocks and txs, forward P2P msgs, drive C.C., send BTC txs, spawn miner
miner
Produce and bcast STX blocks
Gather sigs and wakeup miner
StackerDBListener
NetRes
RelayerDirective
Announce STX block
BTC
Bcast
Globals
MinerStatus
Announce STX block
Broadcast
Event Upload
E.U.
E.U.
E.U.
chainstate DB
cvar

Chains Coordinator

- Process pending BTC and STX blocks in order
- Blocks processing for PoX anchor blocks
- Computes PoX reward sets
- Forwards block receipts to event observers
- Maintains Atlas network state (deprecated)
- Handles STX reorgs due to changed affirmations (deprecated)

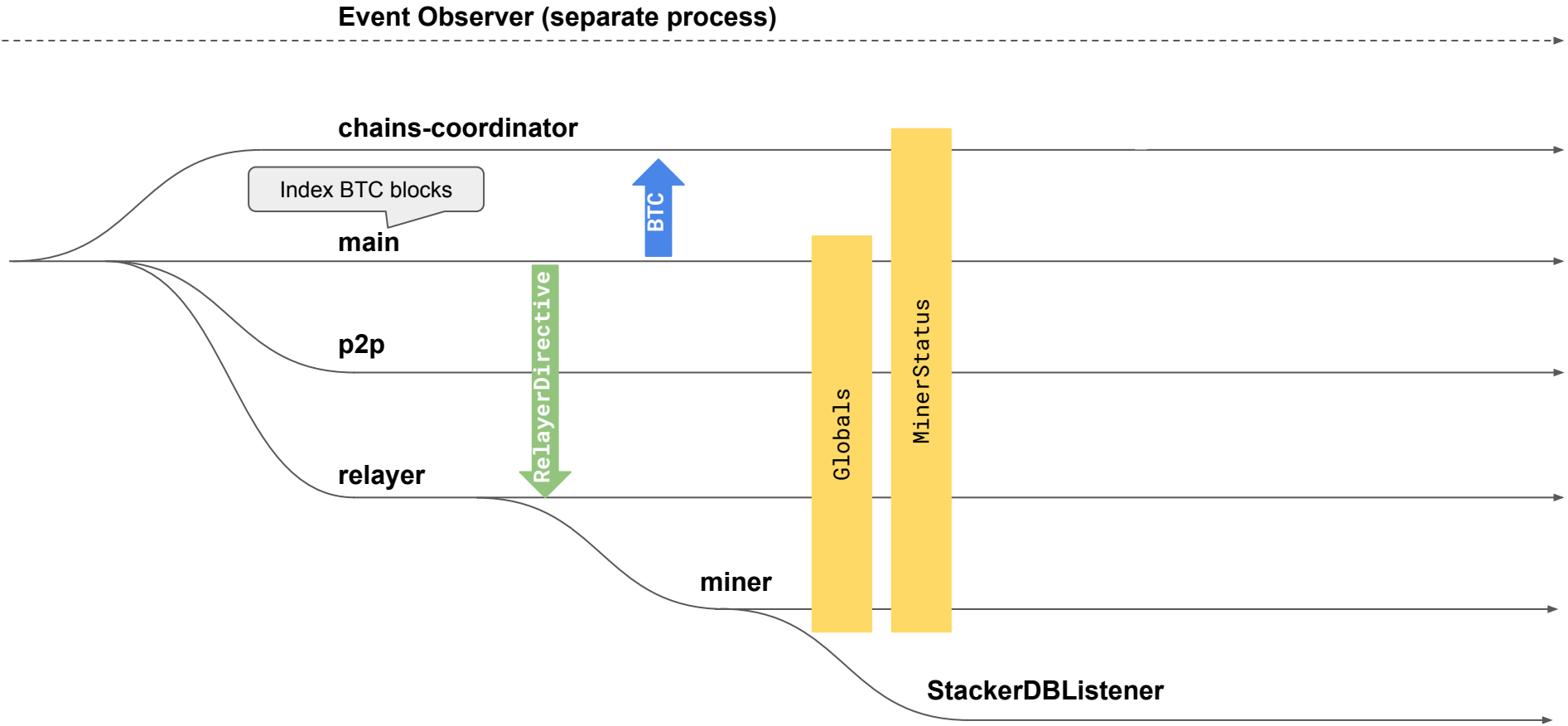
```
stackslib/src/chainstate/coordinator/mod.rs: pub fn run()
stackslib/src/chainstate/nakamoto/coordinator/mod.rs
```


Event Observer (separate process)
Process BTC and STX blocks in order
chains-coordinator
main
p2p
relayer
miner
StackerDBListener
Announce STX block
BTC
MinerStatus
Announce STX block
E.U.
chainstate DB

Main

- Spawns all other components
- Continuously polls for new BTC blocks
- Announces BTC blocks to Chains Coordinator
- Informs relayer when at the chain tip (so mining can start)

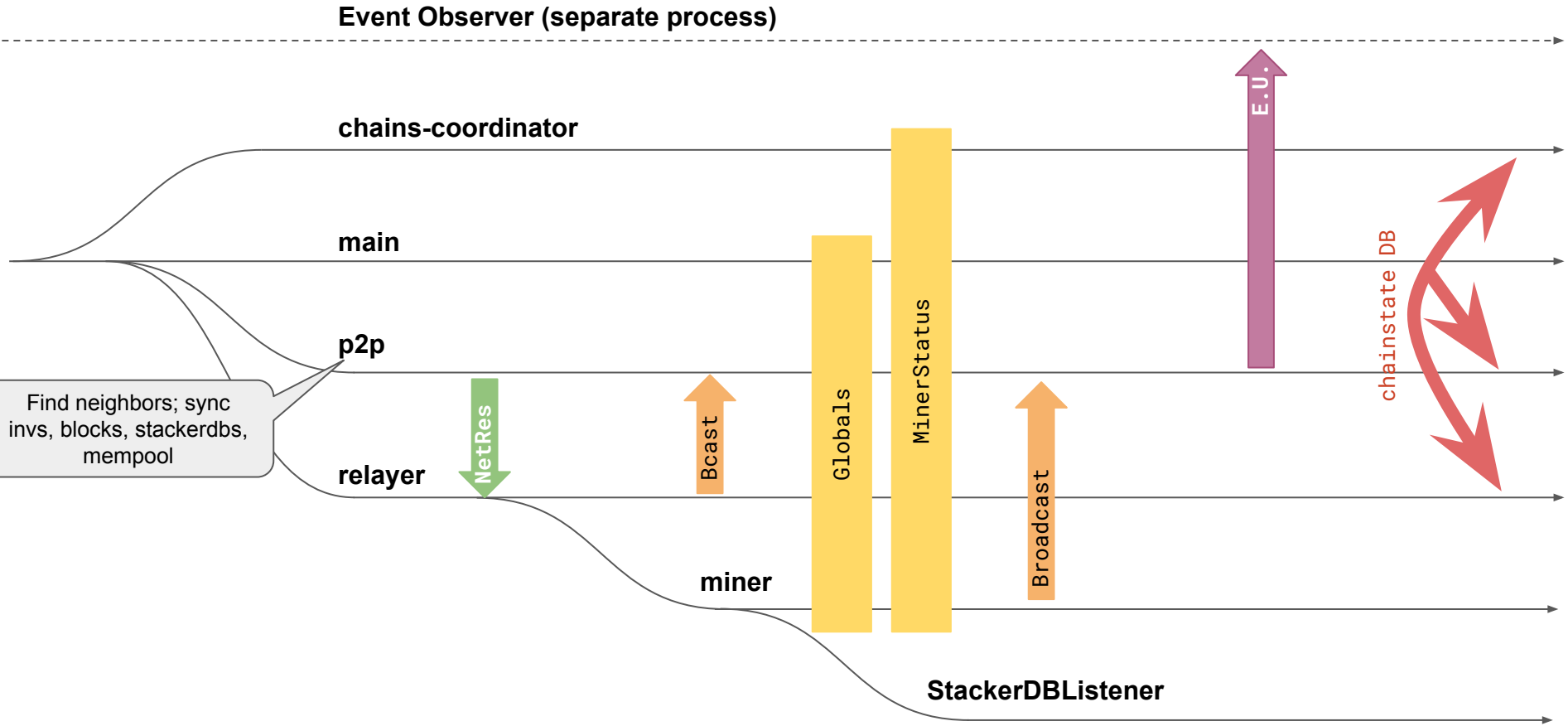
`testnet/stacks-node/src/run_loop/nakamoto.rs`


Event Observer (separate process)
chains-coordinator
Index BTC blocks
main
BTC
p2p
RelayerDirective
relayer
Globals
MinerStatus
miner
StackerDBListener

P2P

- Drives all socket I/O
- Drives P2P network state machines
  - Neighbor walk – discover and prune new p2p neighbors (peers.db)
  - Inv sync – discover which neighbors have which blocks
  - Block download – fetch tenures we're missing
  - StackerDB sync – keep our DBs' chunks consistent with neighbors'
  - Mempool sync – find set difference of remote peer's mempool and get new txs
  - (Deprecated) epoch2.x inv sync
  - (Deprecated) epoch2.x block download
  - (Deprecated) Atlas sync
- Drives HTTP server
  - All data uploads open DB transactions due to API contract

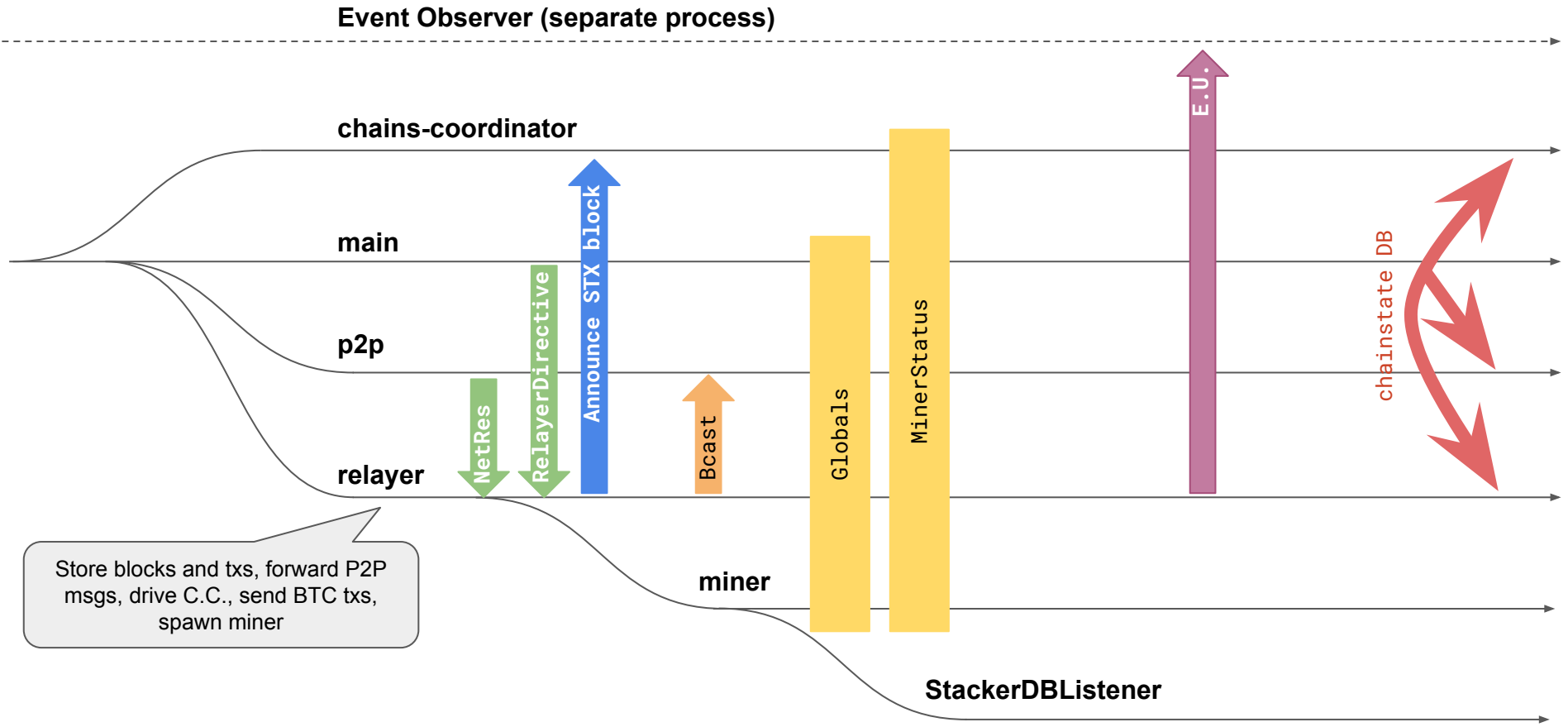
```
stackslib/src/net/p2p.rs: pub fn run()
testnet/stacks-node/src/nakamoto_node/peer.rs
```


Event Observer (separate process)
chains-coordinator
main
p2p
relayer
miner
StackerDBListener
Find neighbors; sync invs, blocks, stackerdbs, mempool
NetRes
Bcast
Globals
MinerStatus
Broadcast
E.U.
chainstate DB

Relayer

- Stores data from P2P thread
- Broadcasts novel data to P2P thread (so it can bcast to neighbors)
- Wakes up Chains Coordinator if novel STX data gets stored
- Drives BTC transaction state machine
  - VRF key registration
  - Block commits
  - Block commit RBF
- Spawn and supervise miner thread

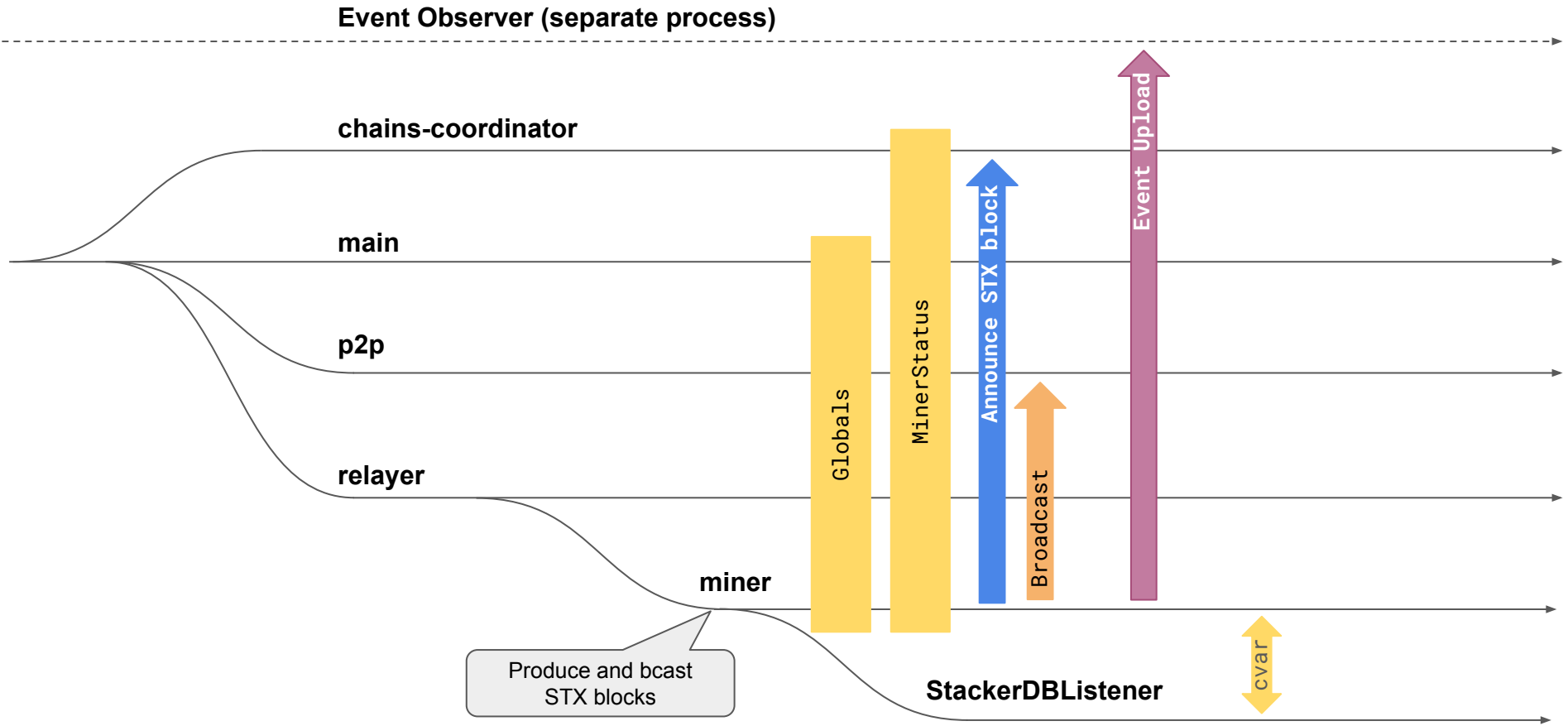
`stackslib/src/net/relayer.rs`
`testnet/stacks-node/src/nakamoto_node/relayer.rs`


Event Observer (separate process)
chains-coordinator
main
p2p
relayer
miner
StackerDBListener
NetRes
RelayerDirective
Announce STX block
Bcast
Globals
MinerStatus
E.U.
chainstate DB
Store blocks and txs, forward P2P msgs, drive C.C., send BTC txs, spawn miner

Miner

- Walk mempool for mineable transactions
- Produce blocks and upload to StackerDB for signers
- Spawn and supervise StackerDBListener thread
- Broadcast signed block (via P2P thread)

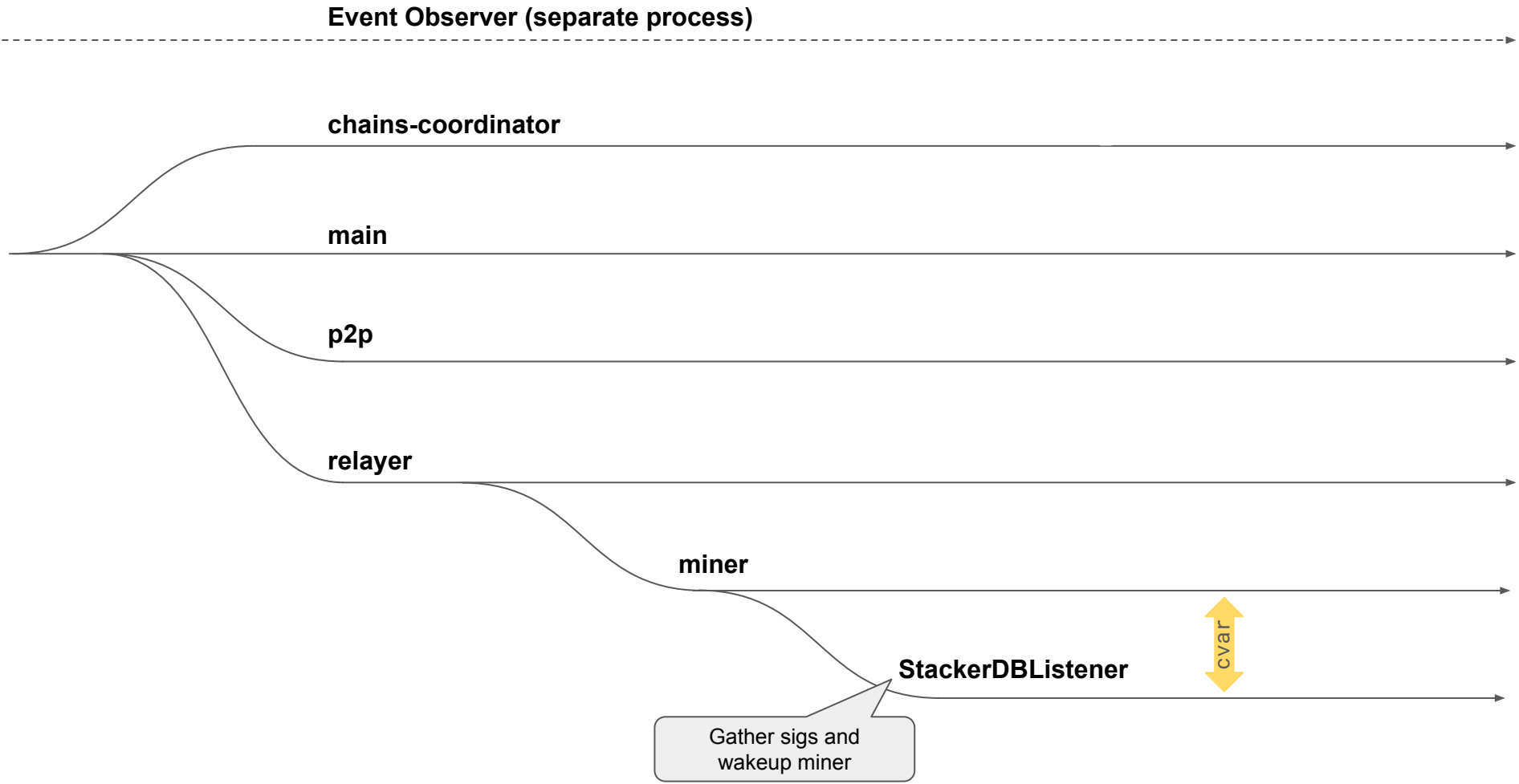
`testnet/stacks-node/src/nakamoto_node/miner.rs`
`stackslib/src/chainstate/nakamoto/miner.rs`


Event Observer (separate process)
chains-coordinator
main
p2p
relayer
miner
StackerDBListener
Globals
MinerStatus
Announce STX block
Broadcast
Event Upload
cvar
Produce and bcast
STX blocks

StackerDBListener

- Waits for StackerDB messages for signers
- Once a signing threshold is met, wake up the miner thread

```
testnet/stacks-node/src/nakamoto_node/signer_coordinator.rs
testnet/stacks-node/src/nakamoto_node/stackerdb_listener.rs
```


Event Observer (separate process)
chains-coordinator
main
p2p
relayer
miner
StackerDBListener
cvar
Gather sigs and
wakeup miner

Globals

- Mutex-guarded global state
- Event Counters
- Communication handles to other threads
- State of Stacks tip
- Shutdown flag

`testnet/stacks-node/src/globals.rs`

MinerStatus

- Mutex-guarded thread waitlist
- Set of threads that want the miner to stop mining
- Miner will interrupt mempool walk if directed to by this struct

`stackslib/src/chainstate/stacks/miner.rs`

Event Observers

- Receives events from the node
  - Mempool accept/drop event
  - New STX block or BTC block (receipts)
  - Mined STX block
  - New StackerDB data
  - (Deprecated) Atlas attachments
  - (Deprecated) epoch 2.x blocks and microblocks
- Different threads send different data
  - Chains Coordinator: Processed STX or BTC block
  - P2P: New mempool tx, new STX block
  - Relayer: New/dropped mempool tx, StackerDB events
  - Miner: mined STX block

DB Contention

- Chains Coordinator, P2P, and Relayer all write to chainstate
  - C.C: writes staging and processed STX blocks and BTC sortitions
  - Relayer: writes staging STX blocks, StackerDBs, and mempool DB
  - P2P: writes STX blocks, StackerDBs, mempool DB (on HTTP upload)
- Event delivery is synchronous
  - Threads hold open a DB tx waiting for event ACK from observer(s) before commit
    - Ensures reliable event synchronization
  - Slow event observer can stall one or more threads

Stacks Academy

Day 2

Q: How does Mining Work?

- How to participate in sortitions?
- How a tenure works?
- Coinbase logic

Cryptographic Sortitions

- “…the selection of public officials or jurors at random, i.e. by lottery, in order to obtain a representative sample.”

  – Wikipedia

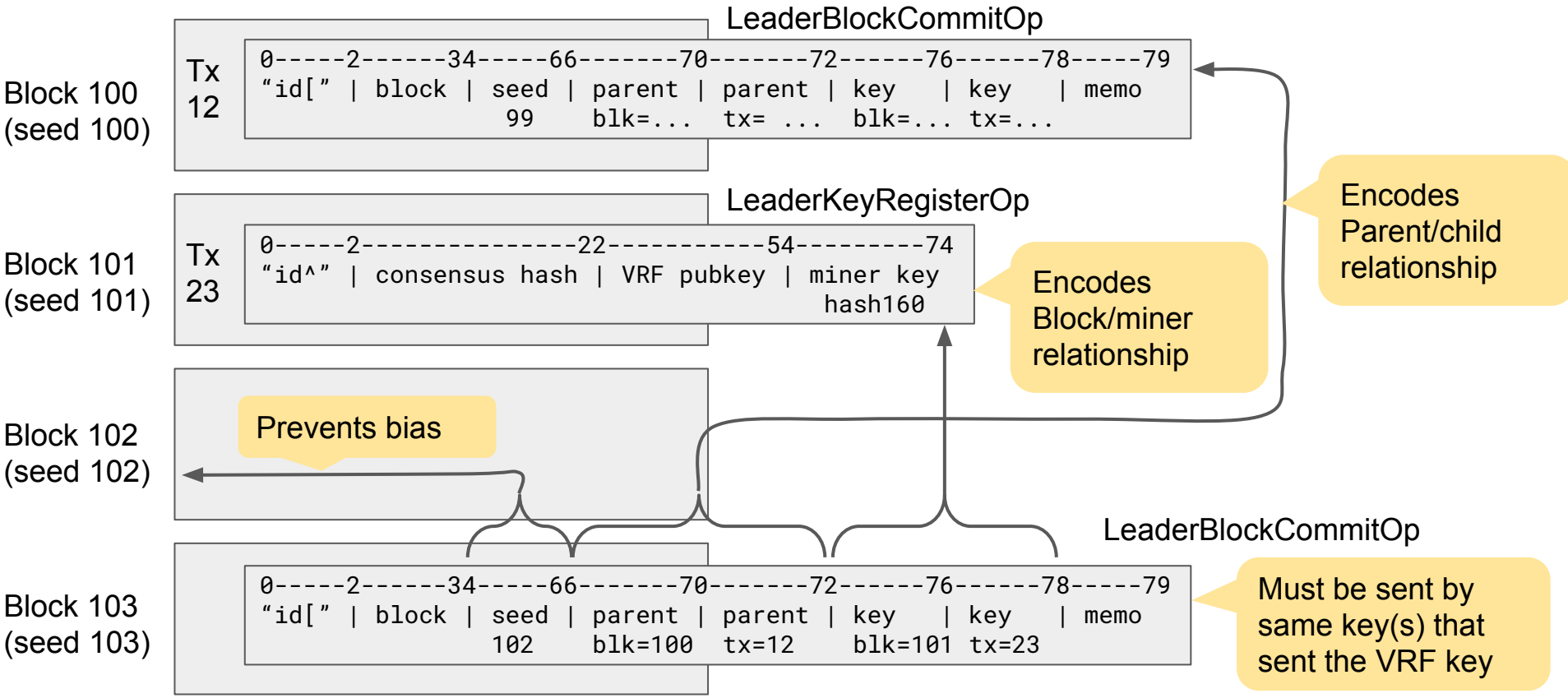
- Miners submit their candidacy to produce the next tenure
  - Each miner pays BTC
- On-chain VRF selects winner
  - Deterministic
  - P[win] is proportional to your fraction of the total block spend
- Single-leader block production
  - Simple and resilient implementation

Cryptographic Sortitions

1. Submit a VRF public key and a block-signing key
    a. `struct LeaderKeyRegisterOp`
    b. Commits to miner public key hash
2. Submit block-commits to mine tenures
    a. `struct LeaderBlockCommitOp`
    b. Points to miner's VRF key (to validate VRF proof in `Coinbase` transaction)
    c. Points to parent tenure's `LeaderBlockCommitOp` (to validate continuity)
    d. Makes next VRF seed (hash of VRF proof)
    e. Commits to hash of the parent tenure's first block (Bitcoin finality)
    f. Can be RBF'ed if miner discovers newer parent tenure

`stackslib/src/burnchains/sortition.rs: pub fn make_snapshot()`

Cryptographic Sortition



Cryptographic Sortition

- Anti-MEV weight function**: Assumed Total Commit with Carryover (ATC-C)**
- You win by being consistent
- Goal: force Bitcoin MEV miners to spend BTC competitively

```
P[tenure] = P[win] * P[sort]
IF all_commits[0] == 0, THEN P[win] == 0
ELSE
                 min(your_commits[0], median(your_commits[-6..0]))
P[win] = ------------------------------------------------------------
                                    all_commits[0]
```

Cryptographic Sortition

```
IF median(total_commits[-6..0]) > 0; THEN
    IF all_commits[0] < median(all_commits[-6..0]) THEN
```

$$P[sort] = ATC\left(\frac{\min(all\_commits[0],\ median(all\_commits[-6..0]))}{median(all\_commits[-6..0])}\right)$$

```
    ELSE P[sort] = 1
ELSE P[sort] = 0
```

Where ATC(x) is a weighted (logistic) probability function. Instead of P[sort] being the ratio,

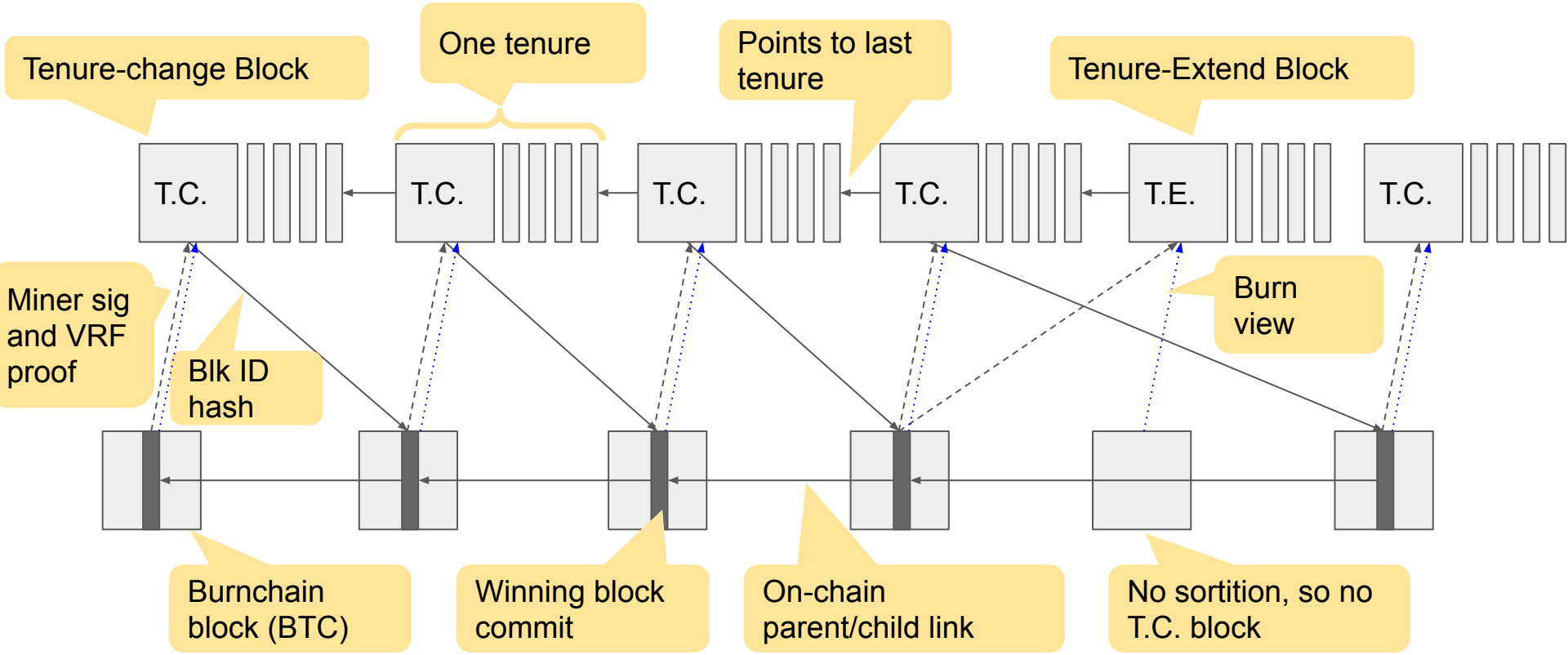
Tenures

- Miner produces blocks as quickly as signers allow
- Tenure cannot exceed a fixed tenure budget
  - `struct ExecutionCost`, tracked by Clarity VM cost functions (see SIP-006)
  - Signers may permit a `TenureExtend` to allot additional tenure budgets
- Signers decide rate of computational resource use
- No “gas” – no “out-of-gas” errors, since Clarity is decidable

Tenures

- Tenures start with a block with a `TenureChange` and `Coinbase`
  - `TenureChange` has a `BlockFound` or `Extended` reason
  - Evaluation of an `Extended` resets the Clarity VM cost tracker
  - Points to winning sortition, parent winning sortition, and current Bitcoin block
- `Coinbase` schedules STX reward to winning miner's address
  - Has a 32-byte memo, a recipient address (could be a contract), and VRF proof
- Validation (non-exhaustive)
  - `LeaderBlockCommitOp` VRF seed must be the hash of the VRF proof
  - Miner must sign with the same signing key as in `LeaderKeyRegisterOp`

Blockchain Structure and Mining



Block Production

- Miner produces a block, and sends it to signers
- Signers each validate block against local chainstate
- Signers vote to accept or reject (70% acceptance for admission)
- Miner builds on accepted blocks, and re-attempts if rejected

Block Production Difficulties

- Miner may have different Bitcoin views than (some) Signers
  - Late-arriving Bitcoin blocks (common)
  - Bitcoin forks (rare)
  - Consequence is partial rejection of valid blocks
- New miner may not have up-to-date Stacks view
  - Signers insist miners build on highest globally-accepted block w/canonical tenure
  - Tenure-change block may be rejected if it orphans such blocks (common)
- Miner may not build on parent block-commit
  - Can happen due to a Bitcoin flash block, in which case, the miner can still mine
  - But if after the “flash block grace period,” the miner will be treated as invalid.

Coinbase

- Allotted to miners according to a Bitcoin-determined schedule
  - SIP-029 revised it from the original whitepaper
- Coinbases from empty sortitions are paid forward to next winner
  - If there are 10 empty sortitions in a row, the winner of the 11th gets them all
  - Ensures that eventually, it will always be profitable to mine
- Miners pay BTC in proportion to win probability * coinbase value
  - Higher STX/BTC price ratio ⇒ more PoX yield
- Coinbase reward is time-locked for 100 Stacks tenures
  - Encourages miners to keep mining after they win

Q: Can a miner orphan another miner's blocks?

- Yes, for now, but only if signers permit it
  - Flash block grace period – miner A and B both submit blocks, and B orphans A
  - Dead miner (no blocks produced) – next miner doesn't need to confirm its commit
- Otherwise, no
  - Can't orphan any prior globally-accepted block outside the grace period
  - Can't orphan a tenure which stopped early (e.g. miner crashed half-way thru)
- Eventually, no orphans ever
  - Requires true BFT agreement between signers, not timeous

Q: When do miners & signers rely on “time”?

- Flash block grace period
- Tenure extends
  - Signer locally permits it if it has witnessed enough “idle time”
- Block timestamps
  - Monotonic increases, but can’t exceed local time by 15s
- Block proposals can be stale
- Block validations can be stale

Q: What's missing from SIP-021 in the code?

- See SIP-025 – no WSTS or aggregate key voting
- PoX anchor block is first confirmed tenure-start in prepare phase
  - Not last confirmed tenure-start block in reward phase
- Missing Stacks-on-BTC delegated Stacking ops
  - `Delegate-stack-stx` and `stack-aggregation-commit` were promised
  - Need to revise SIP-021
- User burn support (SIP-001) was never implemented

Q: Can a miner bid twice on the same block?

- Yes, via separate block-commits (and prior VRF key registers)
- Less cost-effective than sending a single large block-commit

Q: When can an empty sortition occur?

- No valid block-commits
  - Miners' fees weren't high enough
  - Bitcoin miner never saw the commits or RBF'ed commits
  - Old block-commits got mined due to flash block or Bitcoin fork
  - Etc. – there is no exhaustive list of reasons
- ATC-C null miner wins
  - Not enough BTC spent in total, compared to last six blocks
  - Non-linear probability function that sortition winner is defeated by "null miner"
  - Makes it much harder for Bitcoin miners to mine STX at a discount
    - Looking at you, F2Pool

Q: How does signer quorum work?

- Signers make local decision to accept/reject, and announce it
- Signers watch for other signes' decisions to detect global decision
- A block only gets "global accept" if 70% of signing weight accepts
  - Old miner's blocks can keep getting accepted as long as signers never see new miner's blocks, or new miner's sortition
- A new miner takes over once its T.C. block gets "global accept"
  - Can take multiple attempts due to signers having different chain views
- Multiple conflicting miners allowed in grace period

Q: What happens to orphaned Stacks state?

- Stacks follows Bitcoin forks
- If a Bitcoin fork gets dropped, so do the Stacks blocks on it.
- SIP-021 calls for re-mining causally-independent txs (future work)

Q: Why do miners pay to the same two PoX addrs?

- Pick at most two PoX properties:
  - Fixed reward cycle length
  - Everybody gets a guaranteed number of slots
  - The STX per slot is fixed
- Miners can’t know the addrs for Bitcoin block N+1 until N
  - Deterministic random sample without replacement
  - Stops Stacker-miners from blasting BTC to their slots and guaranteeing a win

Q: Why PoX sunsetting?

- This was dropped in Stacks 2.1
- PoX creates warped incentives for Stackers in the long run
  - Stackers would seek to become the only miners who can afford to mine
  - Stackers who mine get a discount
  - Discount is proportional to their fraction of the STX locked if every Stacker mines
- Not a problem in the short-term, since mining is capex-intensive
  - Miners need to sell their STX quickly to remain competitive
- Keeping PoX means yield can be diverted to future projects

Q: Is the signer bitvec validated?

- Not part of consensus
- Signers whose bits are 0 don’t receive PoX rewards
- Signers collectively decide the mandatory bitvec
    - Not yet implemented in code

Q: How to prevent Bitcoin censor attacks?

- ATC-C makes discounted mining hard for BTC miners
  - Motivated from discount mining in the wild by F2Pool
- `stack-stx`, `transfer-stx`, `delegate-stx` have Stacks-on-BTC

Q: Migrate to a new burnchain possible?

- Code structure remains amenable
  - I have ported Stacks to run on Stacks as a burnchain (“app chains”)
  - Hiro has subnets, which use a Stacks contract as a burnchain
  - Efforts in the wild to port Stacks to Dogecoin and BTC forks
- Was motivated by BTC fork wars and original Namecoin focus
- Worthwhile to maintain – could attract more dev talent
  - Ports of Stacks to other chains increases our legitimacy

Q: Other use-cases for StackerDB?

- Docs are scarce due to engineering bandwidth limits
  - Could spend a whole lecture session on how/why it works
- System is generic and largely agnostic to role in signer network
  - `.miners` DB gets special treatment, however
- Lots of use-cases! It lets you leverage Stacks p2p layer for apps
  - Store web3 app data
    - Slots are just binary blobs – store text, images, web pages, whatever
    - Can replace Gaia, our previous decentralized storage layer
  - Store smart contract assets off-chain
    - Pair slots with principals on-chain – storage layer is aware of chainstate
    - E.g. store NFT images, DEX order books, pending multisig tx payloads
  - Comms for other signer networks (e.g. sBTC, bridges, oracles, etc)

Stacks Academy

Day 3

Signers

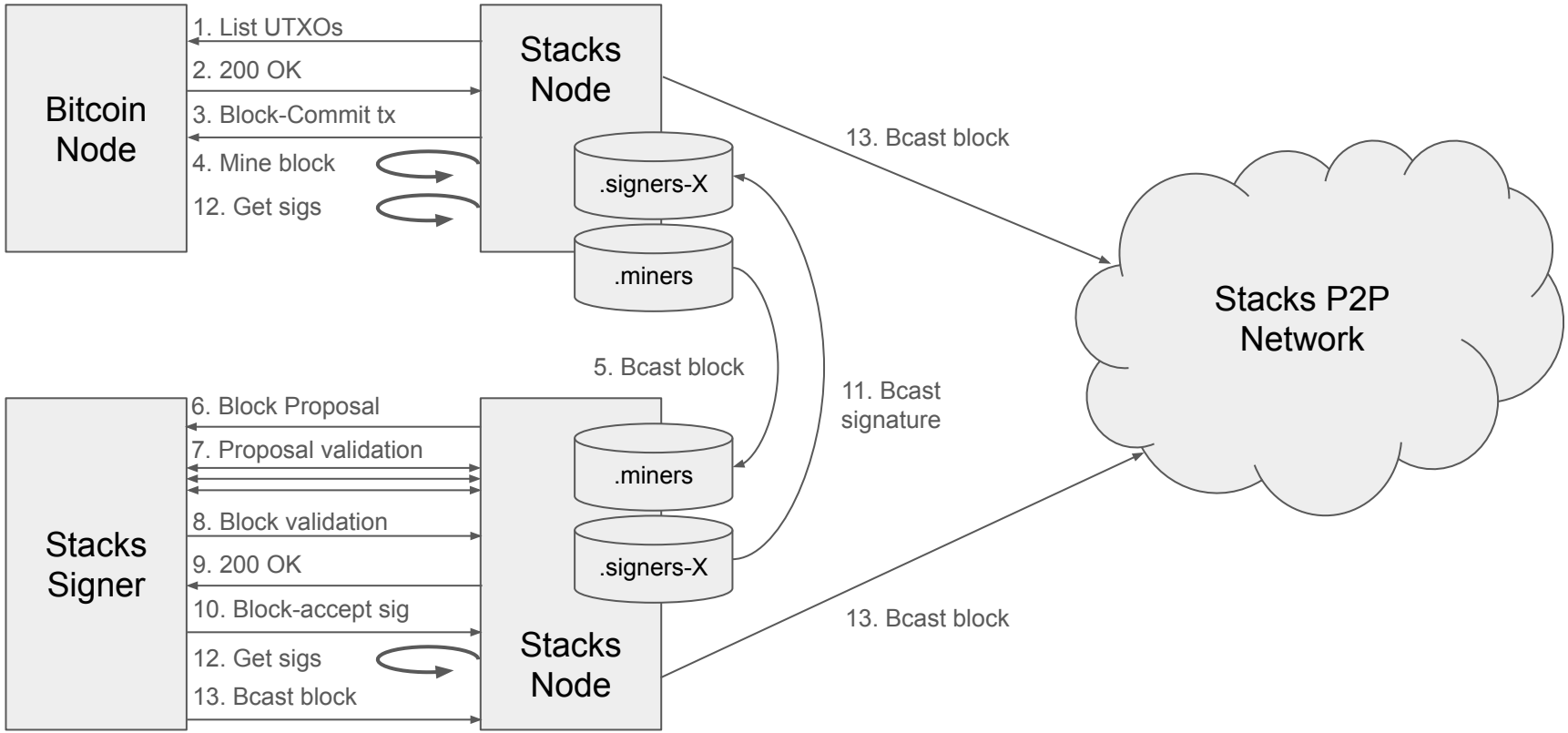
- How to participate in signer election
- How validation and signing of a proposed block works
- BTC earning
- BONUS! StackerDB

Signer Election

- This is PoX – all Stackers are Signers
- Solo Stackers register a signing key on lock-up
- Pool operators register a signing key on aggregation commit
- Block bitvec used to incentivize Stacker participation
    - Signers can withhold PoX yield for lack of participation
    - Signers can go further and prevent you from moving your unlocked STX
    - Punishment code is still TBD

stackslib/src/chainstate/stacks/boot/pox-4.clar

Mining, Validation, Signing



Mining, Validation, Signing

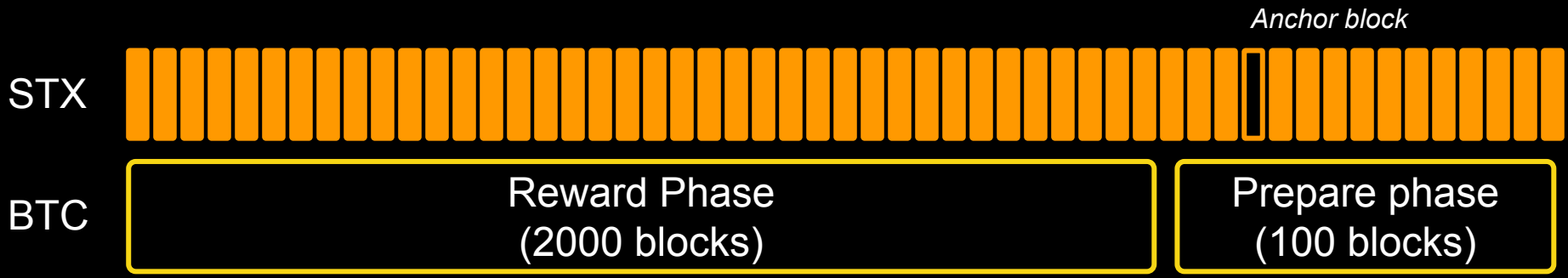
- Miner produces blocks and submits them via `.miners` StackerDB
- Signer validates block and proposal
  - On canonical Bitcoin and Stacks forks
  - Does not orphan any other blocks, unless in grace period
  - Block is semantically valid – asks node to validate against chain tip
- Signer bcasts acceptance signature via `.signers-X-Y` StackerDB
- Miner and Signers listen for enough signatures to reach 70%
- Miner and Signers all broadcast block

Mining, Validation, Signing

- Block signatures are malleable state until confirmed
  - Many combinations and orders of sigs can sum to 70% weight
  - Block hash covers all parent block signatures, but not signer signatures
- Signers enhance block availability and durability prior to bcast
  - Each signer that sees the block as globally accepted applies and bcasts it
  - Helps ensure that the rest of the network (and next miner) have consistent view

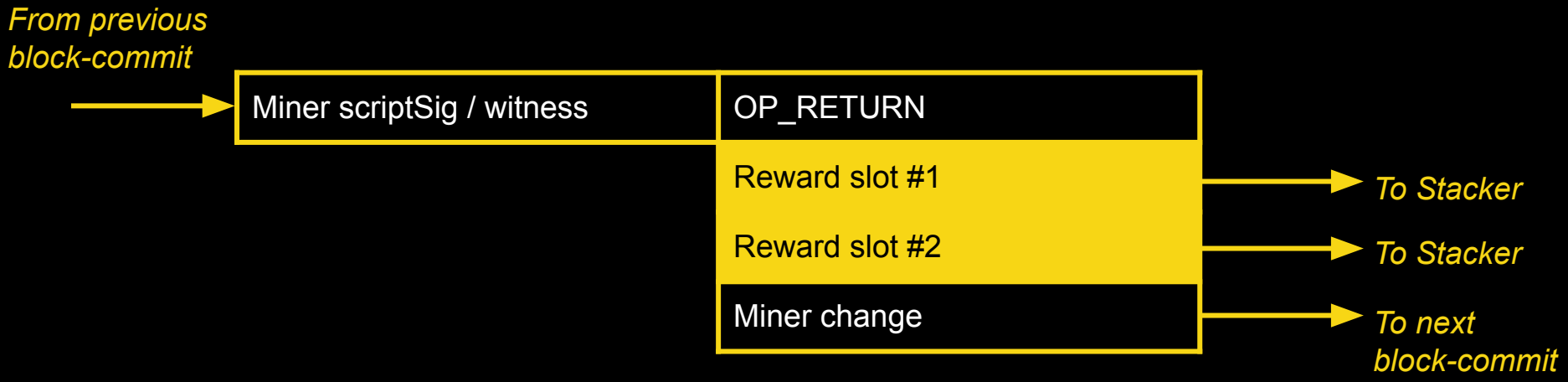
BTC Earning

- **Reward Cycle**: 2100 Bitcoin blocks
- **Reward Phase**: first 2000 Bitcoin blocks
- **Prepare Phase**: last 100 Bitcoin blocks
- **Anchor block**: first Stacks block confirmed in prepare phase
- **Reward Set**: snapshot of locked STX in anchor block
    - Encodes who the next Stackers are, who receives BTC payout, and how often



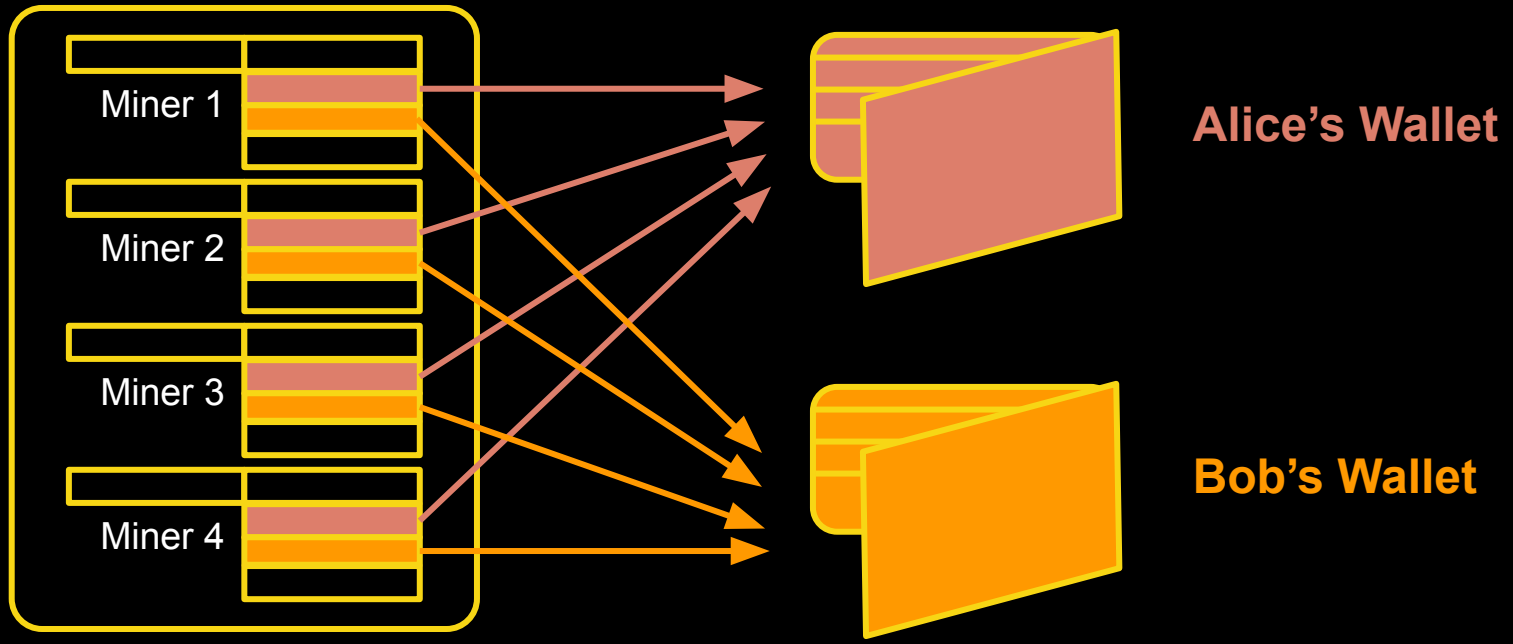
BTC Earning

- **Stacking**: locking up your STX to receive BTC at your BTC address
  - Supports solo and delegated modes
- **Reward Slot**: 1 of 4000 block-commit UTXOs allocated to Stackers
- **Stacking Minimum**: amount of STX required to get a reward slot
  - Usually between 100,000 and 140,000 STX



BTC Earning

- Reward slots sampled from reward set via VRF, without replacement
- All miners pay to the same two reward slot recipients



BTC Earning

- Locked STX **never leave** your account, and unlock automatically
- Locked state in `SP000000000000000000002Q6VF78.pox-4`
  - You lock your STX via a transaction that calls into this contract
- `.pox-4` is queried by the node for reward set
- Not all reward slots are claimed (up to 4000 are)
  - BTC sent to unclaimed reward slots gets burnt
- Miners burn BTC in the prepare phase

StackerDB

- How should signers and miners exchange messages?
- Message Transport Requirements
  - Works across NATs and the hostile Internet
  - All-to-All communication (broadcast, reachability)
  - Message persistence – can query other nodes for delivered messages
  - Fast peer discovery
  - Fast message delivery and broadcast
  - Authorized writers enforced by chainstate (i.e. only signers and miners)
  - Bound message size and upload bandwidth
  - 100% replication – all nodes receive all messages
  - Best-effort anti-entropy (miners and signers implement their own consistency)

StackerDB Peer Discovery

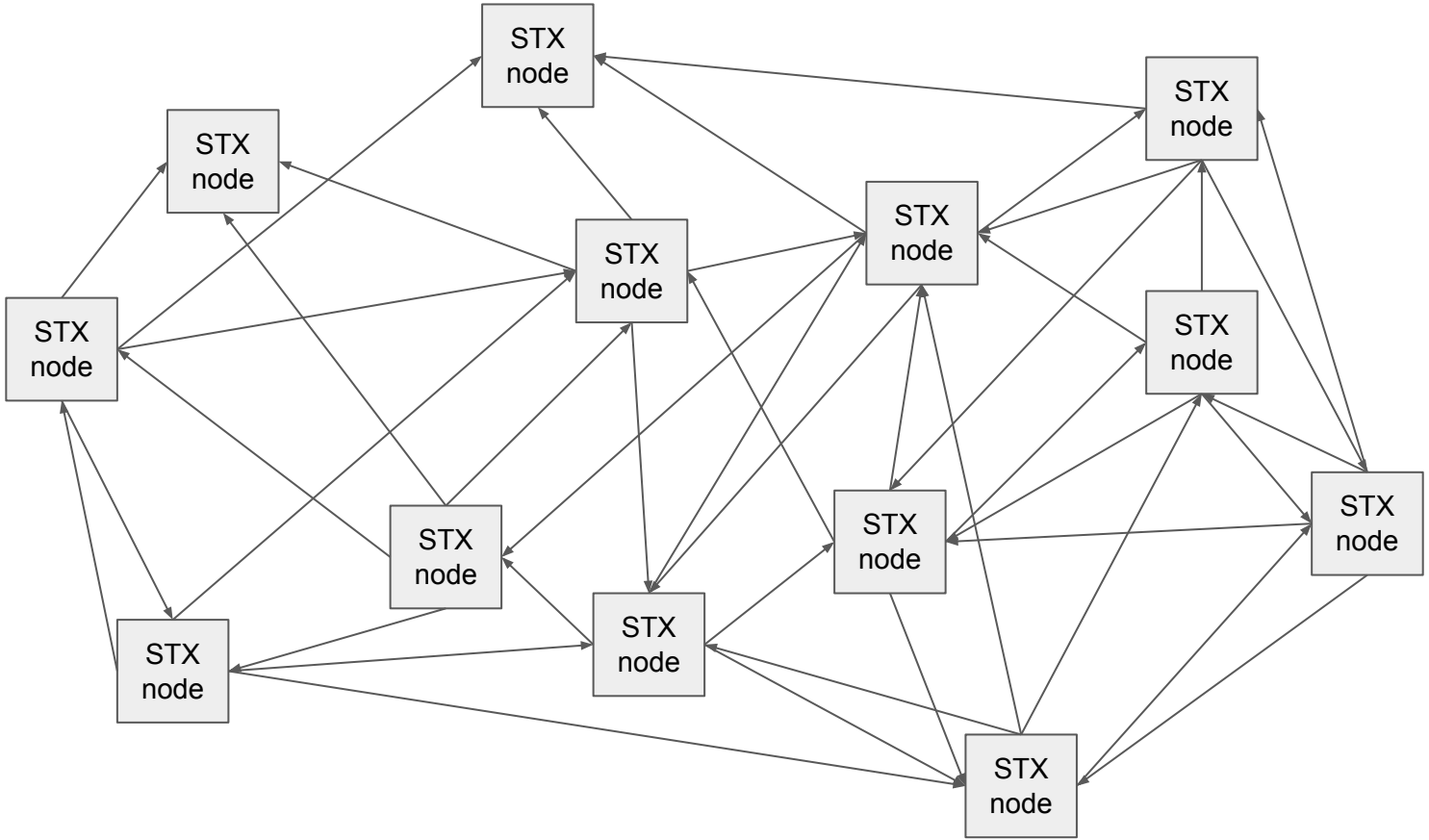
- Peer discovery is the one thing you cannot afford to screw up
- Why not BitTorrent? IPFS? Libp2p? Dat? gunDB?
  - They all rely on distributed hash tables for peer discovery
  - DNS, or the reliance of distinguished peers (e.g. SSB), is a cop-out and a SPOF
- **DO NOT EVER USE AN OPEN-MEMBERSHIP DHT**
  - “2003 called. It wants its routing attacks back.”
  - Structured open-membership p2p overlays are all **trivially easy** to break
  - DHTs are **slow** – both for discovery and propagation

High-level summary pertinent to blockchains:
https://www.nccgroup.com/us/research-blog/how-to-spot-and-prevent-an-eclipse-attack/

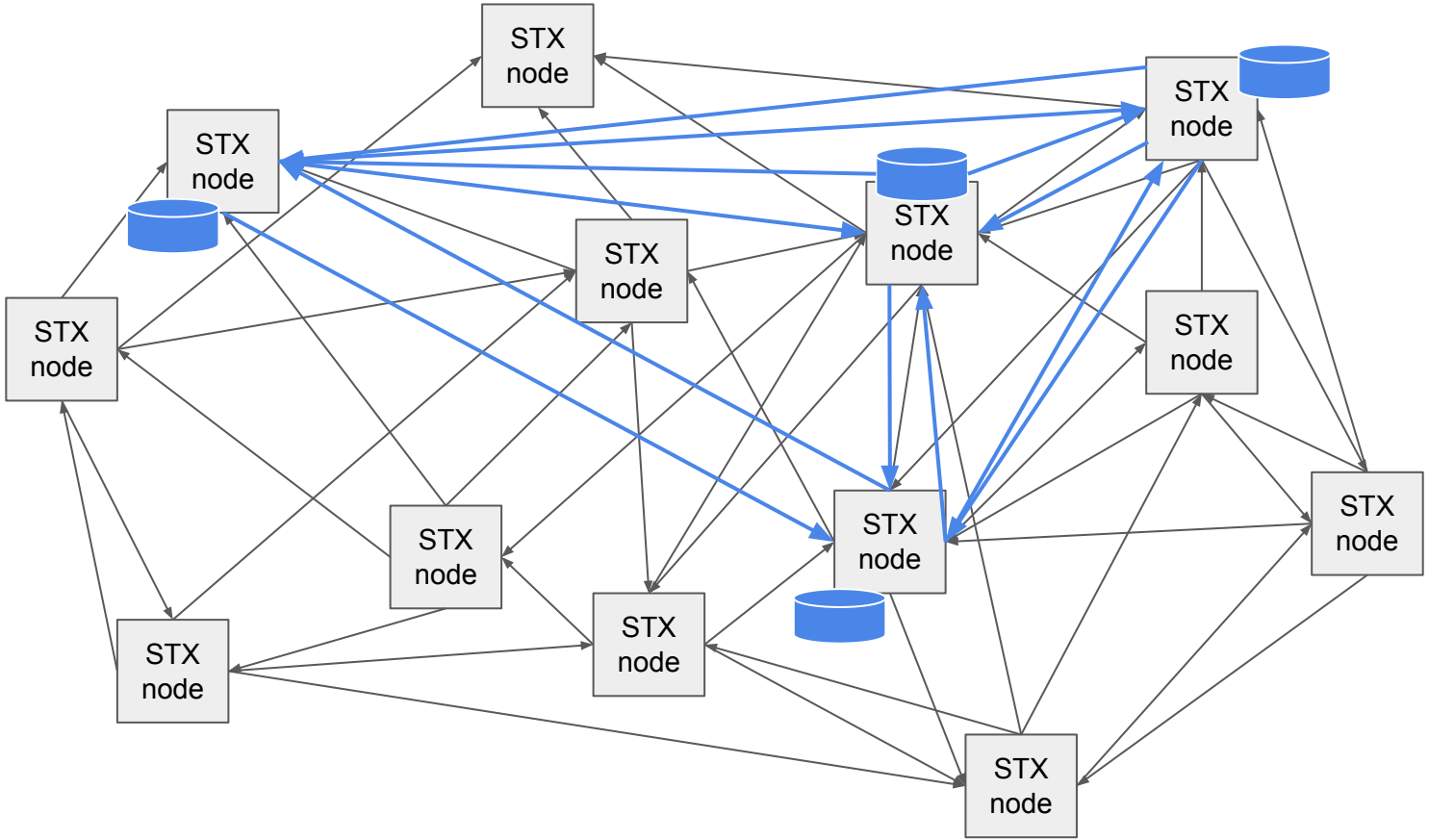
StackerDB Peer Discovery

- Reuse battle-tested Stacks p2p network
  - K-regular random peer graph with fast-convergence MH sampling
  - Already meets all service requirements, but for blocks
- StackerDB builds on learned p2p peer graph
  - Stacks already knows set of reachable peers and their StackerDBs in its peers DB
    - Eager replica discovery through Stacks p2p handshakes
  - Bootstrap K-regular random peer graph from known-subscribed replicas

StackerDB


STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node

StackerDB


STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node
STX node

StackerDB Storage

- **Slot**: a unit of replication, ID'd as 0 up to the DB size (max 4096)
  - A DB is made of one or more slots
- **Chunk**: the contents of a slot
  - A slot may not have a chunk – the slot may be empty
- Each slot has an authorized writer, learned from chainstate
  - A smart contract controls authorized writers and DB config
- Each chunk has a monotonically-increasing version number
  - Used for anti-entropy sync
- Each chunk must be signed by its authorized writer
  - Only chunks whose signatures match the slot writer will be replicated
  - StackerDB replicas only sync if they agree on the slot writers — Tantamount to agreeing on the Stacks chain tip

StackerDB Configuration

```
(define-trait stackerdb-trait

	(define-public (stackerdb-get-signer-slots) (response (list 4096 { signer: principal, num-slots: uint }) uint))

	(define-public (stackerdb-get-config)

	(response {

		chunk-size: uint,

		write-freq: uint,

		max-writes: uint,

		max-neighbors: uint,

		hint-replicas: (list 128 { addr: (list 16 uint), port: uint, public-key-hash: (buff 20) })

	},

	uint)))
```

StackerDB Configuration

- Node calls `stackerdb-get-config` on tip change
  - Subsequent StackerDB sync state machines will use it
  - `write-freq` : frequency of state-machine passes, and rate of new chunk acceptance
  - `hint-replicas` : some super-peers you know of (but nodes can override locally)
- Node calls `stackerdb-get-signer-slots` on tip change
  - If the slot list has changed since the last time, then reconfigure the replica
  - If signer *S* can write to slot *L* before and after, then its chunk will be preserved
  - Otherwise, the chunk will be dropped

StackerDB Replication

1. If `write-freq` seconds have not passed since last sync, then skip
2. Handshake with random replica neighbors pulled from peer DB
   a. If they don't connect or aren't at the chain tip, then disconnect
3. Exchange `[(slot_id, version)]` list for DB
   a. Track which peers have newer slots, so we can download them
   b. Track which peers have older slots, so we can upload them
4. Download remote newer chunks in rarest-first order
5. Upload local newer chunks in rarest-first order
   a. Remote node NACKs if the uploaded chunk is stale, and replies newer version
   b. Goto 4 if we discover this way that the remote replica has newer data than us
6. Sync complete; reset state machine

StackerDB Replication

- All nodes eventually learn of all replicas
- Every replica eventually obtains the latest chunks
- New chunks are broadcast to replica's connected neighbors
  - Epidemic protocol, like block and transaction propagation
  - Even if there are no connected replicas, the node will eventually find some
  - Signers and miner carry out this broadcast when mining
- Authenticate remotely-downloaded chunks with local chainstate
  - You don't need to run a replica *per se*; you just need a reachable one
  - Use `stackerdb-get-signer-slots` to authenticate a remotely-fetched chunk

StackerDB API

- StackerDB chunk broadcast endpoint
  - `POST /v2/stackerdb/{:addr}/{:name}/chunks`
- StackerDB list known replicas
  - `GET /v2/stackerdb/{:addr}/{:name}/replicas`
- StackerDB list chunk metadata
  - `GET /v2/stackerdb/{:addr}/{:name}`
- StackerDB get versioned chunk
  - `GET /v2/stackerdb/{:addr}/{:name}/{:slot_id}/{:version}`
- StackerDB get latest chunk
  - `GET /v2/stackerdb/{:addr}/{:name}/{:slot_id}`

Q: Delta from SIP-021/025? Why not WSTS?

- SIP-025 allows us to use miner multisig instead of WSTS
- “Mock signing” was used in Stacks 2.5 to live-test signers & miners
- WSTS had many operationalization challenges
  - PoX anchor block has to be found at start of prepare phase to learn new signers
  - Need to run DKG at least once per reward cycle (that’s a lot)
  - Coordinator can crash. How to fail-over reliably and quickly?
  - Coordinator can be malicious. How to detect and fail-over reliably and quickly?
  - Vote for aggregate public key. How to guarantee vote delivery? Tx fees?
  - What if vote fails? DKG restart? Chain halt?
- Not many clear answers

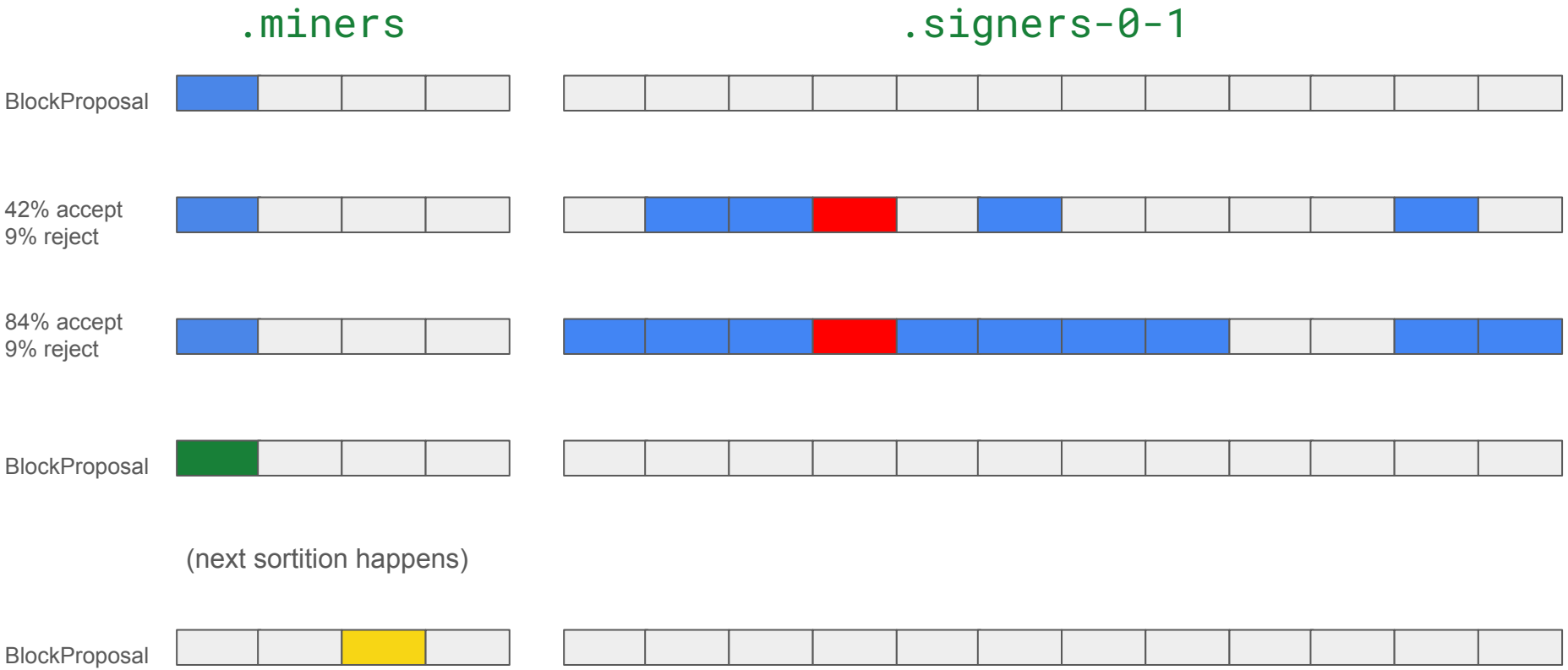
Q: What's up with "shadow blocks?"

- More like "Shadow tenures" – a tenure is fully shadow or not at all
- Shadow blocks are created by a SIP. They're hard-coded.
  - No need to announce or replicate them. They're already known to all nodes.
  - Cannot be mined; they are synthesized out-of-band
  - Added to chainstate through new consensus rule; requires a SIP vote
- Shadow blocks allow a coordinated chain restart on PoX failure
  - A shadow block can be the PoX anchor block
- Shadow blocks allow erasure of corrupted chainstate
  - A shadow block replaces corruption-inducing blocks

Q: Technical overview of Signer/Miner Comms?

- SIP-021 provides protocol diagrams and overview
- Some StackerDB contracts are “boot contracts”
- `.miners` StackerDB
  - Four slots – two for mock-mining, two for block proposals
  - Current and previous sortition winners can write to two slots each
- `.signers` contract
  - Control interface for setting up `.signers-XXX-YYY` StackerDBs
  - Called internally by Stacks node when a new PoX reward set is computed
- `.signers-XXX-YYY` StackerDBs
  - `XXX` is 0 or 1 – corresponds to current and prior signer set (mod 2)
  - `YYY` is a step in WSTS DKG (0-12; 13 in all). Only `.signers-XXX-1` is used today.
  - Reads `.signers` to provide list of signer slot addresses (e.g. 43-ish slots)
  - Otherwise, these are “normal” StackerDBs – the node doesn’t modify them

Q: Technical overview of Signer/Miner Comms?



Q: Two signers have same public key?

- They still get only one StackerDB slot
- Weight is the sum of all STX locked behind that key
  - Signing key can represent multiple PoX reward addresses

stackslib/src/chainstate/stacks/boot/mod.rs: `pub fn make_signer_set()`

Stacks Academy

Day 4

Transactions

The ask:

- Types of transactions
- Defining a smart contract, available data and functions
- How contract-calls are validated and executed

How we'll talk about this:

- How chain state is stored and accessed by transactions
- How transaction-processing happens
- Types of transactions and how they are processed

How Transactions Work

- Read and append chain state within a Stacks fork
- Atomic execution, linearized history within a fork
- Key Components
  - MARF and Sidestore – fork-indexed chain state (`struct MarfedKV`)
  - Clarity DB – API for MarfedKV to read/write contract and account state
  - Clarity VM – executes smart contracts and contract-calls
  - `enum STXBalance` – represents STX account state (epoch-specific variants)
  - `struct MinerPaymentSchedule` – future block reward allotments
  - `struct MinerReward` – matured block reward
  - `struct StacksChainState` – top-level API to Stacks chain state

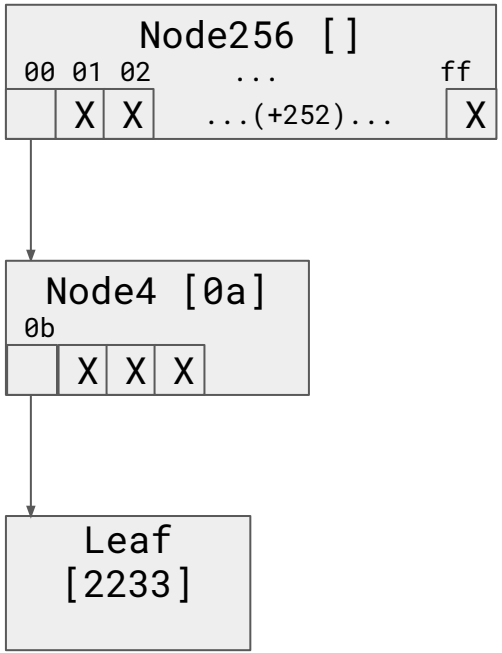
MARF and Sidestore

- **MARF**: Merklized Adaptive-Radix Forest (SIP-004)
- Fork-aware append-only authenticated persistent hash index
  - A set of authenticated hashed array-mapped tries, with variable radix (4,16,48,256)
- Deterministic order-dependent state with cryptographic commitment
  - Each node contains hashes of children
  - Root node contains hash of skiplist over ancestors
- Efficient Merkle proofs-of-inclusion ($O(\log^2 n)$ time and space)
  - Prove that key `k` has value `v` in ancestor trie `A` off of tip `T`
- Values are hashes; these map to values in a sqlite DB (**sidestore**)
- Efficient appends via COW semantics
  - Batch append – build new trie in RAM, compute node hashes, and dump to disk
  - $O(\log A)$ append due to skiplist hash computation over `A` ancestors

MARF and Sidestore

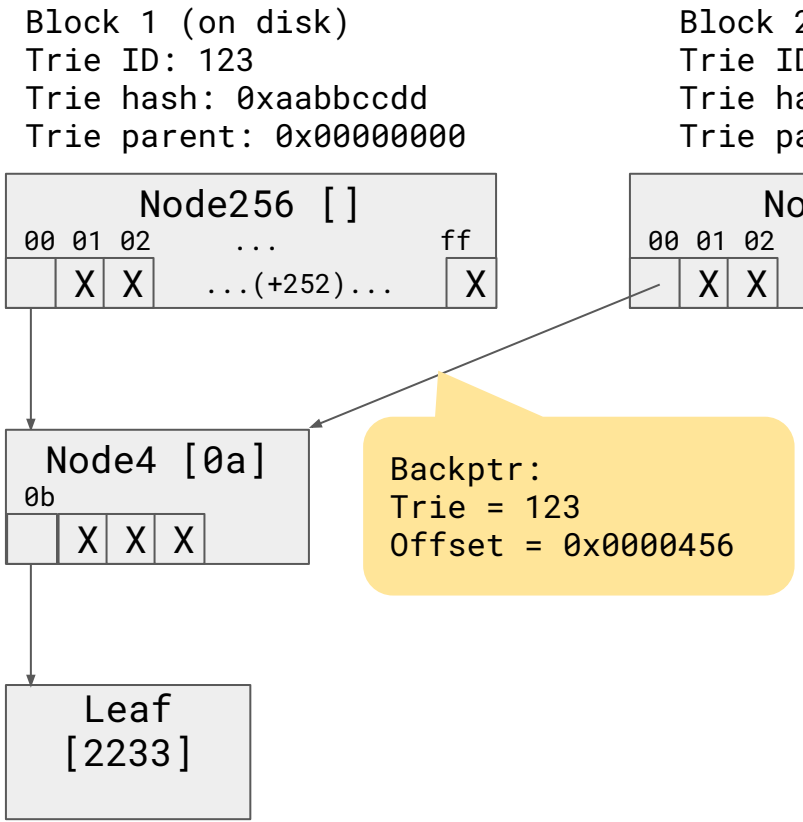
```
Block 1 (on disk)
Trie ID: 123
Trie hash: 0xaabbccdd
Trie parent: 0x00000000
```

```
Block 2 (in RAM)
Trie ID: 124
Trie hash: ???
Trie parent: 0xaabbccdd
```



- Create block 2 as a child of block 1
- All `Leaf` nodes in all tries must remain reachable from block 2's root node.
- When creating the root trie for block 2:
  - Load block 1's root node (`Node256`)
  - For each non-`backptr`, non-empty pointer in block 1, create a `backptr` to the child node in block 1 for block 2
  - For each `backptr` in block 1, simply copy it to block 2

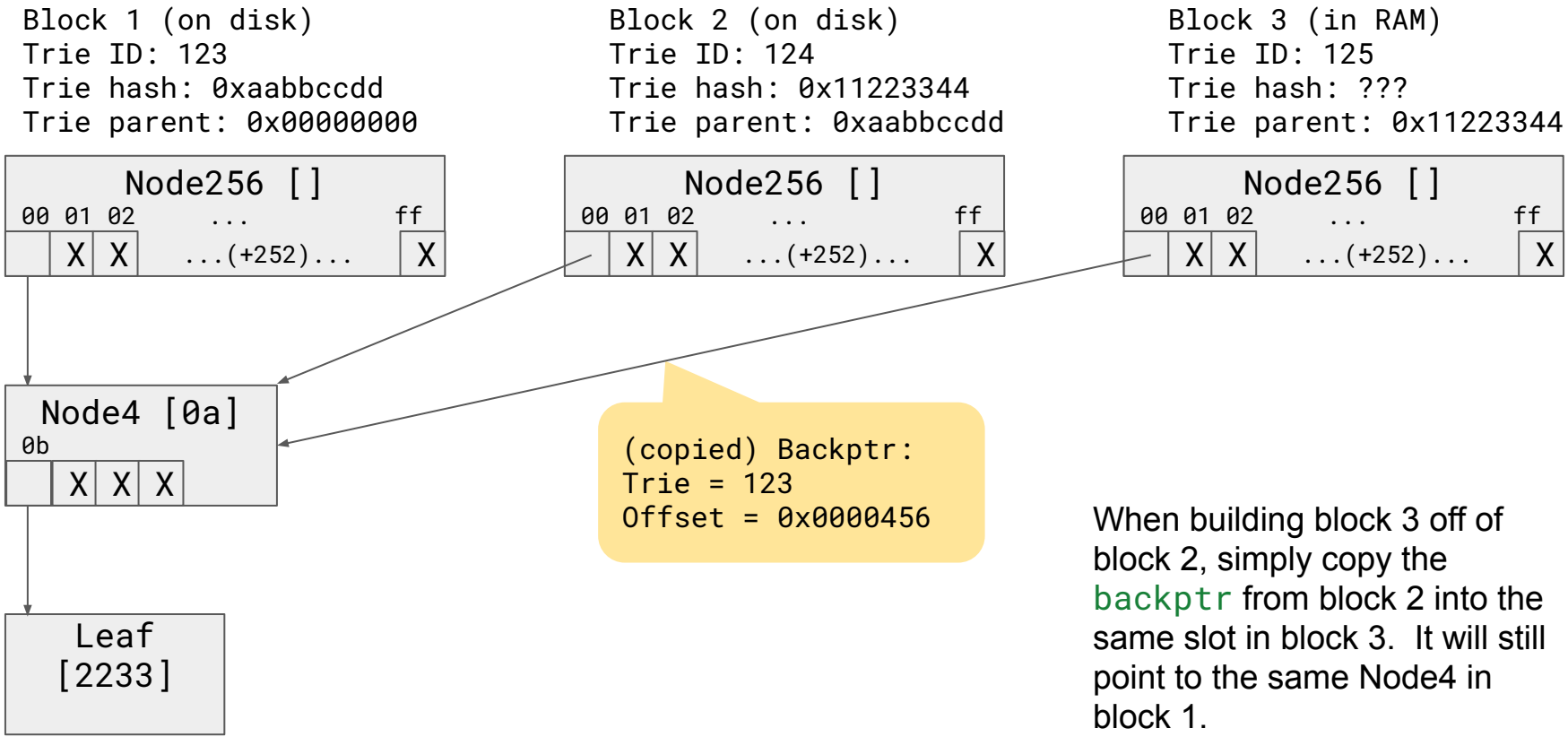
MARF and Sidestore



To read the leaf at `00 0a 1b 22 33` from Block 2:

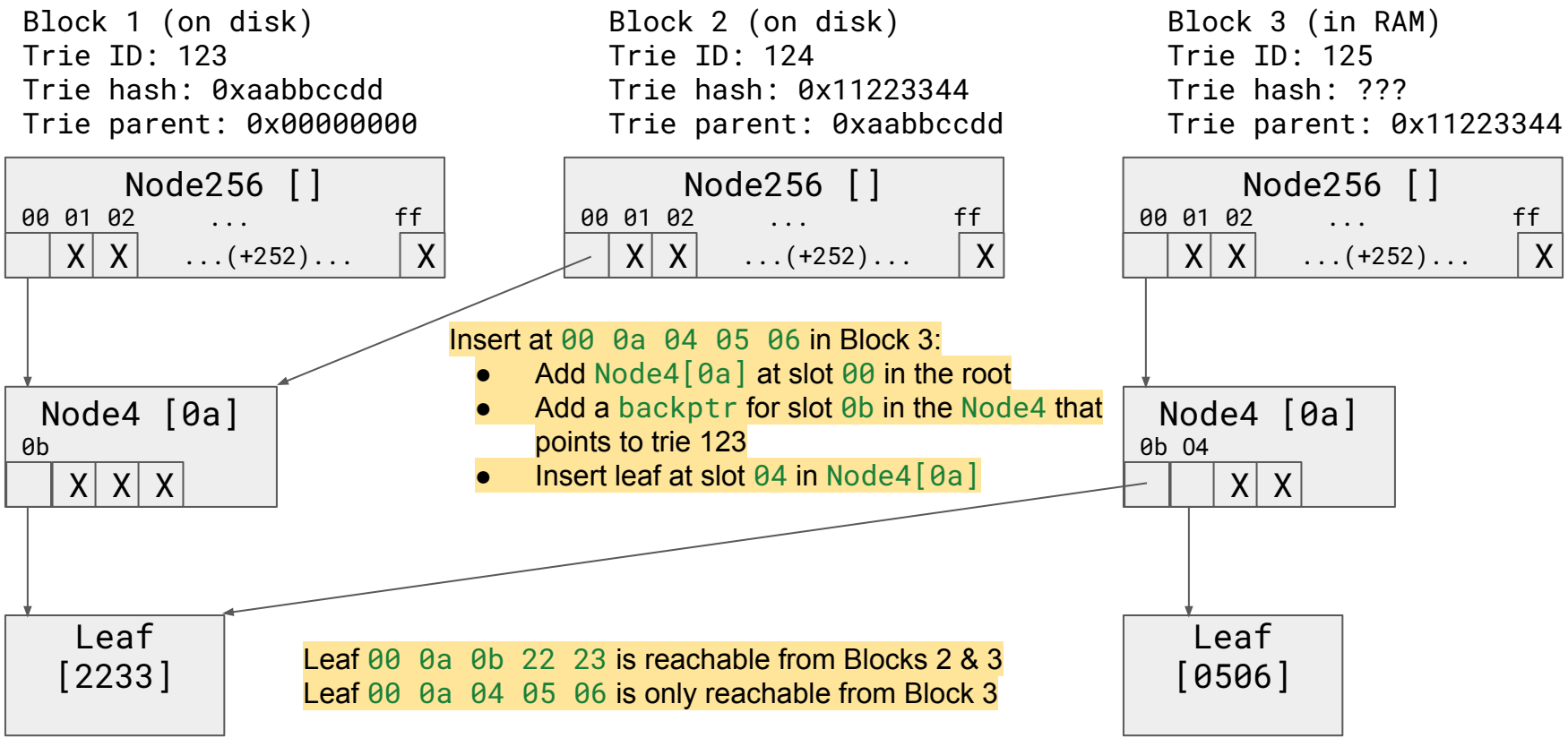
1. Walk to pointer at `00`
2. Resolve trie ID 123 to trie hash `0xaabbccdd`
3. Open trie `0xaabbccdd`
4. Seek to `0x00000456` (the `Node4` on disk)
5. Load the `Node4`
6. Walk compressed path `0a`
7. Walk to child at `0b`
8. Load the `Leaf`
9. Walk unexpanded path `22 33` (resolved!)
10. Return the `Leaf`'s content hash

MARF and Sidestore

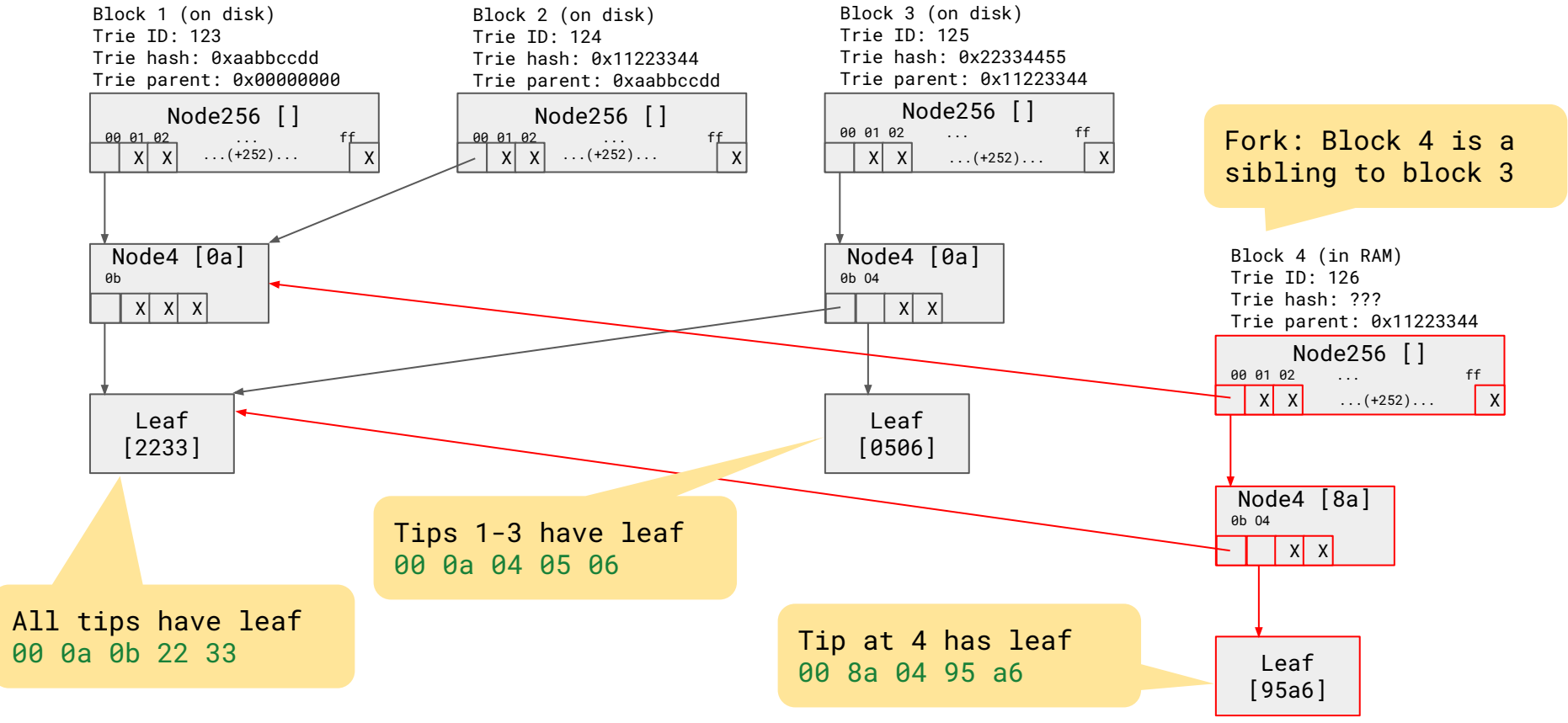


When building block 3 off of block 2, simply copy the `backptr` from block 2 into the same slot in block 3. It will still point to the same Node4 in block 1.

MARF and Sidestore



MARF and Sidestore



MARF and Sidestore

- All `get` operations are relative to a trie tip `T`
- All tries map parent trie ID to parent height, and vice versa
  - Tip `T` maps all ancestors' IDs to their heights, and vice versa
- Inserts are *O(1)*
- Lookups are *O(1)*
- Appending a trie is *O(log A)* due to skiplist calculation
- No deletes, but new trie may overwrite existing keys
- Inserts only permitted when creating a new trie (i.e. a batch)
- Batch inserts (new tries) are atomically persisted

MARF and Sidestore

- **Sidestore**: a hash-addressed sqlite DB
- MARF keys *and* values are hashes of arbitrary byte strings
- MARF values map to byte string value in sidestore DB
- MARF is external to DB (append-only file of concatenated tries)
- Sidestore maps trie IDs to byte offsets in trie file (`.blobs`)
- **MarfedKV**: instantiated sqlite DB and MARF index

MARF: `stackslib/src/chainstate/stacks/index/*.rs`

Sidestore: `stackslib/src/clarity_vm/database/marf.rs`

Clarity Instance: `stackslib/src/clarity_vm/clarity.rs`

ClarityDB

- API to read/write smart contract and account data in a MarfedKV
- Stores both consensus-critical data, and non-critical metadata
  - Wire formats for chain state data are all consensus-critical – see SIP-005
  - Metadata is not critical, and includes contract ABI, AST, IR, types, etc.
- Every smart contract, data var, and data map is in a MarfedKV
- Clarity VM uses ClarityDB to instantiate and run contracts
  - See the Clarity Book for the language itself: https://book.clarity-lang.org/
- `struct RollbackWrapper` to buffer up a block's writes
- `struct ClarityTx` for running transaction code
  - Runs Clarity VM with tx code, which reads/writes data to inner ClarityDB
  - `RollbackWrapper` condensed into a batch of `MarfedKV` inserts to build a trie

ClarityDB

- Read a data var
  - Compute “data var key” from contract ID and variable name
  - Hash the key and walk the `MarfedKV` to its value’s hash
  - Query the sidestore for the value bytes, and deserialize it into a `Value`
- Write a data var
  - Insert hash of data var key and value into `RollbackWrapper`
  - On commit, `RollbackWrapper` writes key’s hash and value’s hash to MARF, and value bytes to sidestore
- Read a data map key
  - Same idea, but compute “data map key” from contract ID, map name, and map key
- Write a data map key/value
  - Same idea, but with a data map key

`clarity/src/vm/database/*.rs`

Transaction Processing (high-level)

1. Open a `ClarityTx` for the block at given Stacks and Bitcoin tips
2. Evaluate each tx and make a `StacksTransactionReceipt`
   a. Check authorizations
   b. Load sending account(s) from `MarfedKV`
   c. Check nonce(s)
   d. Debit fee and write new account state to `MarfedKV`
   e. Process payload
   f. Update nonce(s)
3. Store MARF-indexed block header and metadata
4. Process `MaturedMinerReward` from 100 tenures ago
5. Commit `ClarityTx`

stackslib/src/chainstate/stacks/db/transactions.rs: `fn process_transaction()`

stackslib/src/chainstate/nakamoto/mod.rs: `fn process_next_nakamoto_block()`

Transaction Payload: Coinbase

- 32-byte memo field
- VRF proof
- Recipient Stacks principal
- Creates a `MaturedMinerReward` to be processed in 100 tenures

Transaction Payload: TenureChange

- Mostly a no-op; handled in block-processing preliminaries
- `Extended` case will reset ongoing resource usage counters
- Processed before any other transactions are processed (special case)

Transaction Payload: TokenTransfer

- 34-byte memo (legacy hold-over)
- Recipient Stacks principal
- Amount of uSTX to send (unsigned 64-bit)
- Debits origin and credits recipient the given amount of uSTX
  - Read sender's account from `MarfedKV`
  - Read recipient's account from `MarfedKV`
  - Debit sender's account
  - Credit recipient's account
  - Write sender's updated account to `MarfedKV`
  - Write recipient's updated account to `MarfedKV`

Transaction Payload: SmartContract

- Name string (`ContractName`)
- Code body (`StacksString`)
- Optional: Clarity language version (if different from the active one)
- Run contract analysis in `ClarityTx` and store it if successful
  - Stores AST, IR, ABI, and data types, which the Clarity VM will evaluate
- Evaluate code body in `ClarityTx`'s inner Clarity VM
  - Evaluation logic calls `ClarityDB` to make data vars and data maps in `MarfedKV`
- Store contract code, contract address, contract hash
- Always mined even if code is bad, since analysis and eval aren't free
- Invalidates block if:
  - tenure budget is exceeded (remember, there are no out-of-gas errors)
  - Defensive runtime sanity-check fails or interpreter-level error happens

Transaction Payload: ContractCall

- Contract address (`StacksAddress`, `ContractName`)
- Function name (`ClarityName`)
- Function args (`Vec<Value>`)
- Load up contract's analysis state from `ClarityTx`'s `MarfedKV`
- Evaluate the named function with the given arguments
- Invalidates block if
  - Contract or function does not exist
  - Argument types or argument list length are wrong
  - Runs out of tenure budget
  - Defensive runtime sanity check fails or interpreter-level error happens

Transaction Payload: PoisonMicroblock

- Deprecated; only supported in Stacks 2.x
- Two sibling `StacksMicroblock` headers signed by same miner
- Destroys miner's `MaturedMinerReward` if less than 100 tenures old
- Transaction origin account gets 5% commission on block reward

Transaction Authentication

- `enum TransactionAuth`: type of authorization
  - `Standard`: fee paid by origin account (one `TransactionSpendingCondition`)
  - `Sponsored`: signed by origin but fee paid by separate sponsor (two `TSC`'s)
- `enum TransactionSpendingCondition`: authorization signature(s)
  - `SingleSig`: Exactly one signer
    - Recover pubkey from signature, compute hash160 to match address
  - `MultiSig`: A sequence of signers must sign
    - Recover pubkeys from signatures via rolling sighash, compute hash160 of keys
  - `OrderIndependentMultiSig`: A set of signers must sign
    - Recover pubkeys from signatures, compute hash160 of keys
- Uses Bitcoin-style `p2pkh`, `p2sh`, `p2wpkh-p2sh`, `p2wsh-p2sh` hashing
  - Recover BTC tx scriptSig/witness to Stacks address
  - Every STX address is a BTC address, and vice versa

Transaction Postconditions

- Clarity VM tracks movements of STX, tokens, and NFTs already
- Exposes this “asset map” to transaction-processing
- **Postcondition**: assertion about asset map at the end of the tx
  - Allow/deny mode
  - “Principal *A* sent *ARITHMETIC_COMPARISON* tokens to principal *B*”
  - “Principal *A* sent / did not send NFT to *B*”
  - Only possible since assets are first-class types in Clarity
- Future work
  - Support assertions on asset receipt, not just transfers
  - Lift into Clarity VM, for e.g. post-conditions on `(as-contract ...)` blocks

Q: How is tx atomicity guaranteed?

- State Machine Replication – all nodes see the same tx histories
- By construction, materialized view represents a linearized history
- Stacks blocks commit to MARF’s tip trie’s root hash + skiplist tip
- Any sanity or interpreter errors in tx processing invalidate the block
- Post-condition evaluation is part of tx payload evaluation

Q: How would SIP-021 Tx Replay Impact This?

- It doesn’t

Q: Support for Schnorr signatures and others?

- Requires a hard fork and a corresponding SIP
- Requires new `TransactionSpendingCondition` variant
- This is how `OrderIndependentMultiSig` was added (in SIP-021)

Stacks Academy

Day 5

Sortition code walkthrough

- Business needs beget code structure
  - “What does X do?” does not have a meaningful answer without knowing a process
  - A process encodes both current *and* historical business needs
  - Learn the business need, then the process. Then, it’ll be obvious what X does.
- Blockchain code is append-only
  - Backwards compatibility is an uncompromisable requirement unique to blockchains
  - All the more important to understand historical business needs and processes
- I don’t know your unknown unknowns
  - You can figure out what code does without me
  - You’re here to learn *why* the code needed to be written

Sortition code walkthrough

- Process: compute winning miner
  - Business need: Built on Bitcoin
    - Bitcoin has no smart contracts, so relevant Stacks txs encode data
    - Need to download, parse, and store Stacks-specific Bitcoin transactions
  - Business need: Bitcoin can fork
    - Need to track all Bitcoin forks, since the true Stacks fork could be on any one
  - Business need: PoX consensus
    - Can only compute winning miner if we know the reward cycle's anchor block
    - Process is intrinsically tied to Stacks block processing
  - Historic business need: Stacks 2.x
    - Stacks can fork, and there can be multiple PoX anchor blocks in a cycle
    - All Stacks nodes must agree on the current winner, *without* signers' help
    - One winner per Bitcoin block, since local selection is also the global selection
    - Code can't be deleted

Sortition code walkthrough

- `testnet/stacks-node/run_loop/nakamoto.rs`
  - Entry point: `RunLoop::start()`
  - Loop to poll Bitcoin for new blocks
    - `BurnchainController::sync()`, which calls
    - `BurnchainController::receive_blocks()`, which calls
    - `Burnchain::sync_with_indexer()`, which spawns threads to call
      - `BurnchainBlockDownloader::download()`, which pass blocks to
      - `BurnchainBlockParser::parse()` on each block, which passes block to
      - `Burnchain::process_block()` to store relevant txs and header, and then
      - `CoordinatorChannels::announce_new_burn_block()`
- `stackslib/src/chainstate/coordinator/mod.rs`
  - Entry point: `ChainsCoordinator::run()` (in thread spawned by `RunLoop::start()`)
  - Waits for other threads to wake it up, and it'll process everything it can
    - `CoordinatorChannels` instance is inter-thread wake-up signaler

Sortition code walkthrough

- `ChainsCoordinator::run()`
  - Loops for wakeup bit flags from `CoordinatorChannels`
  - Dispatches to epoch handler (`ChainsCoordinator::handle_comms_nakamoto()`)
  - If the bit field indicates a new “burn block” (Bitcoin block) was downloaded, call
    - `handle_new_nakamoto_burnchain_block()`, which calls
    - `find_sortitions_to_process()` to get new downloaded Bitcoin blocks
    - Process each downloaded Bitcoin block
      - Compute new reward set if we’re at a reward cycle boundary
      - Call `SortitionDB::evaluate_sortition()` with it

Sortition code walkthrough

- `SortitionDB::evaluate_sortition()`
  - Create a MARF'ed sortition DB transaction, `SortitionHandleTx`
    - Query any tables with TX, as well as ancestor sortitions & data via MARF
  - Compute new VRF seed (`mix_burn_header()`)
  - Pick next PoX recipients (`pick_recipients()`)
  - Process all Bitcoin transactions (`process_block_txs()`)
    - This is where sortitions happen
  - Store them (`store_transition_ops()`)

Sortition code walkthrough

- `SortitionHandleTx::process_block_txs()`
  - `process_block_ops()`
  - Sort transactions in the order they appear in the Bitcoin block
  - Check each transaction (`check_transaction()`) for validity
    - `BlockstackOperationType::check()`
    - Retain ones that are valid
  - Remove duplicate VRF key registers
  - Process them (`process_checked_block_ops()`)
    - `BurnchainStateTransition::from_block_ops()`
      - Everything needed to compute a sortition
    - Next PoX ID (a bitvec of anchor blocks, always `11111…` with Nakamoto)
    - Next Sortition ID (internal unique identifier for snapshots)
    - Make sortition with `BlockSnapshot::make_snapshot()`
      - This computes the sortition
    - Store sortition and MARF'ed fork metadata (`append_chain_tip_snapshot()`)

Sortition code walkthrough

- `BurnchainStateTransition`
  - `burn_dist`: Probability distribution of block-commits winning
  - `accepted_ops`: Valid Stacks-specific Bitcoin transactions, decoded
  - `consumed_leader_keys`: which VRF keys got used by block-commits
  - `windowed_block_commits`: UTXO-linked histories of miners' valid block-commits
  - `windowed_missed_commits`: UTXO-linked histories of miners' late-mined commits
- `BurnchainStateTransition::from_block_ops()`
  - Computes probability distribution for block-commits
    - Finds valid and missed block-commits, to compute probability distribution via
    - `BurnSamplePoint::make_min_median_distribution()`

Sortition code walkthrough

- `BlockSnapshot::make_snapshot()`
  - `make_snapshot_in_epoch()`
  - Compute due coinbase reward
  - Empty sortition if no valid commits or no miner is active enough
  - Compute total BTC spent
  - Pick winning block via `BlockSnapshot::select_winning_block()`
  - If in epoch 3.0 or higher, then apply ATC-C
    - No winner if miner is not active enough (`check_miner_is_active()`)
    - Find carry-over and null miner win prob. (`get_miner_commit_carryover()`)
    - Check if null miner wins (`null_miner_wins()`)
      - Empty sortition if so
  - Find winning miner's public key hash (`get_leader_key_at()`)
  - Return `BlockSnapshot`

Reading Sortition Data

- Primary keyed by `SortitionId` and `ConsensusHash`
  - Local and global identifiers
- Query data in any fork via bare `rusqlite::Connection`
- Query any ancestor data and fork metadata with `SortitionDBConn`
  - Use `SortitionDBTx` to write data while needing this information
  - Used mainly by legacy affirmation map system from Stacks 2.x
- Query data within a fork with `SortitionHandleConn` ancestor
  - Used by Clarity VM – implements `BurnStateDB`
  - Use `SortitionHandleTx` for appending data to a fork
- Fork Metadata (`sortdb::db_keys`)
  - Sortition MARF key/value pairs specific to a fork
  - Mostly PoX-related information and some legacy Stacks 2.x stuff

Q: which files handle sortitions?

- Files aren't as important as functions and structs
- `stackslib/src/burnchains/` – Bitcoin indexer
- `stackslib/src/chainstate/burn/db/` – Sortition DB
- `stackslib/src/chainstate/burn/operations/` – Bitcoin operations
- `stackslib/src/chainstate/burn/` – Sortition calculation
- `stackslib/src/chainstate/coordinator/` – Chains coordinator
  - `stackslib/src/chainstate/nakamoto/coordinator/` – Naka-specific chains coord.
- `testnet/stacks-node/src/burnchains/` – Bitcoin "driver" for node
- `testnet/stacks-node/src/run_loop/` – Main Bitcoin indexer loops

Q: which parts of sortdb.rs are irrelevant?

- `impl ChainstateDB for SortitionDB` was never used

Q: sortdb types and traits?

- We just covered them :)

Q: What domains does SortitionDB serve?

- Tracks all PoX forks ever processed
  - What PoX forks exist (in Nakamoto, there is 1 per BTC fork)
  - What Bitcoin operations happened in which PoX forks
  - PoX reward set state at each sortition in each PoX fork
  - Memoization of canonical Stacks tip in each PoX fork
  - Exposes PoX fork info to Clarity VM

Q: Primary readers and writers?

- One writer – the `ChainsCoordinator`
- Many readers for many purposes
  - P2P thread
  - Relayer thread
  - ChainsCoordinator thread
  - Miner thread

Q: Purpose of `BlockSnapshot`?

- Encode the state of PoX at a given Bitcoin block
  - Can be different PoX states for the same Bitcoin block, pre-Nakamoto
  - Determine winning miner and block, and Bitcoin ops that were picked up
  - Link to parent sortition to embody a fork
- Tabulate running totals
  - Accumulated coinbases
  - Accumulated BTC spends
- Canonical Stacks tip pointer (Stacks 2.x)
  - Tracked the presence of Stacks blocks in a particular Bitcoin fork
  - Each snapshot has a memo to point to the canonical Stacks fork
    - Canonical PoX fork points to canonical Stacks fork
  - This still happens in Nakamoto, but via a dedicated table

Q: Sortition fork choice?

- We don’t care about chain work at this layer (bug?)
- Sortitions at the same height are concurrent
- We pick tie-break ordering arbitrarily

Stacks Academy

Day 6

Design of the Stacks P2P Network (part 1)

- Core design principles
- Common protocols
- Control Plane
- Data Plane (part 2)
- Code walkthrough (part 2)

Blockchain P2P Overview

- Networking is a non-outsourceable core competency
  - Every blockchain problem is ultimately a network problem
  - The network is the platform upon which the blockchain is built
  - Network design greatly influences blockchain design
- Goal: fast, complete, reliable and robust block replication
  - Avoid partitions, especially eclipses
  - Maintain redundant paths
  - Anti-entropy protocol with fast paths
- Latent bipartite topology
  - NAT’ed vs Public nodes
  - Overlay networks cannot hide or ignore this

Stacks P2P Overview

- **Control-plane**: establish peer knowledge
- **Data-plane**: all P2P services that run atop peer knowledge
- Key concepts
  - `ConnectionP2P` – P2P transport to a remote peer
  - `ConversationP2P` – an authenticated stateful session with a peer
  - `NeighborComms` – a set of peer sessions managed by a state machine
  - `NeighborWalk` – a state machine for populating the PeerDB via random walk
  - `PeerDB` – an persisted map of the P2P network
  - `PeerNetwork` – drives socket I/O, drives state machines, maintains active sessions

Key Design Choices

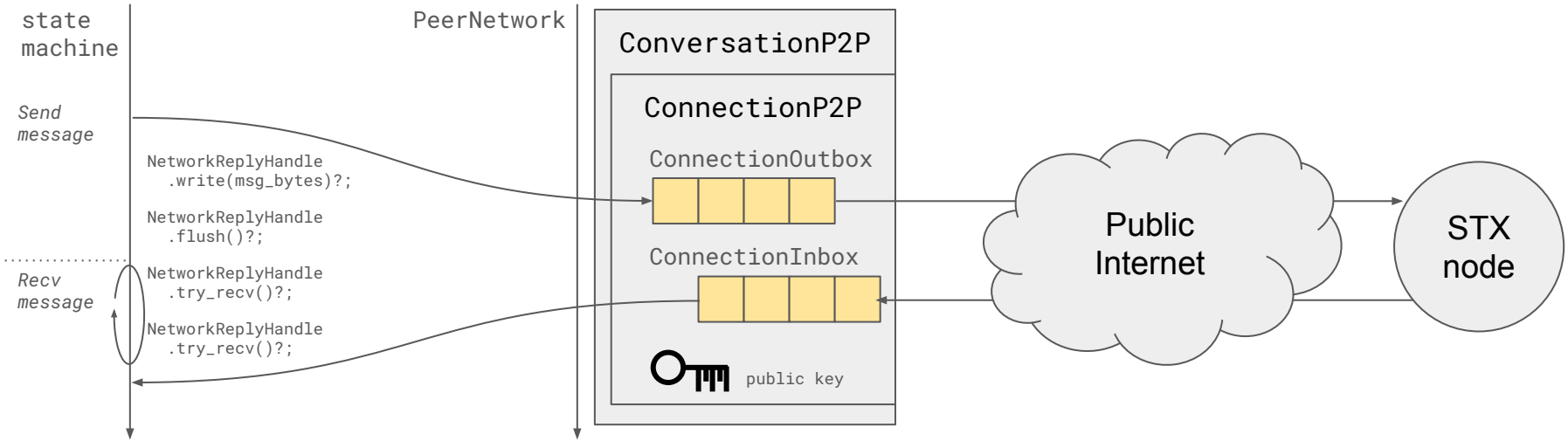
- TCP only
  - You'll need flow control either way
  - We use stateful TCP sessions to traverse NATs, so NAT'ed nodes participate in p2p
  - You probably don't want to open ports on your router
- Separate HTTP server for bulk data download
  - Deploy Stacks node with reverse proxy, web cache, CDN
  - Keep a "small" but stable P2P protocol, and evolve a "big" set of HTTP endpoints
- Message authentication via public-key cryptography
  - Learn public keys through config or TOFU
- Native message codec
  - Unambiguous big-endian byte representation via `StacksMessageCodec` (SIP-003)
  - Optimize for fast partial decoding – fields have known offsets in the bytestream
  - Track message relayers to assess end-to-end P2P graph structure

Stacks P2P Transport (`ConnectionP2P`)

- Mailbox paradigm
  - Every “sent” message is matched to a “received” message via a sequence number
  - Supports “send-only” (no reply expected) – used for broadcast
  - Inbox and outbox buffers
- Channel-oriented API
  - `.write()` and `.flush()` message bytes to an outbox slot
  - `.recv()` or `.try_recv()` a `StacksMessage` from an inbox slot
  - Sender and receiver can be in different threads from the p2p thread
  - No `async/await`
- Used to send/recv messages in a peer session

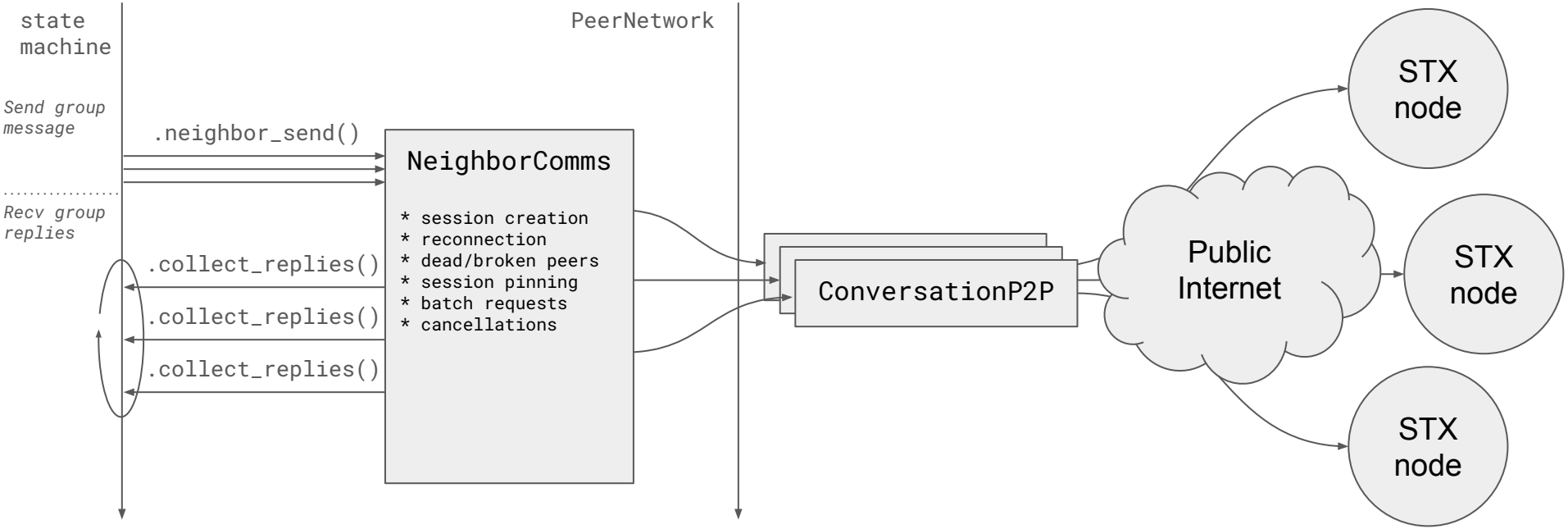
Stacks P2P Sessions (`ConversationP2P`)

- Wraps a `ConnectionP2P`
- Implements P2P control plane protocol
  - Tracks public key expiry, peer liveness, bandwidth, diagnostics, peer graph view, etc.



Stacks P2P Peer Sets

- State machines talk to a *set* of peers, not singular peers
- `NeighborComms` – a “group session” managed by a state machine



Stacks P2P Control Plane

- Open/establish sessions to/from remote peers
- Send/receive keep-alive `Ping` and `Pong` messages to check liveness
- Help remote peers discover their public IP addresses
  - `NatPunchRequest`, `NatPunchReply`

Opening a Session (`ConversationP2P`)

- `Handshake` – open a session
  - `HandshakeAccept` / `StackerDBHandshakeAccept` – remote peer accepts the session
  - `HandshakeReject` – remote peer rejects the session for some reason
- `Handshake` / `.*HandshakeAccept` contains
  - Sender's P2P address and HTTP RPC URL (maybe different from socket address)
  - Which services the sender supports
  - Public key and expiration (as a Bitcoin height)
  - `StackerDBHandshakeAccept`: list of StackerDB contract IDs it replicates
- Pre-conditions to opening a session
  - `HandshakeAccept` public key is not the local public key, and is not expired
  - If a public key is already known, then the message is signed with its (previous) private key
- Post-conditions to opening a session
  - Both peers update their PeerDBs with each other's `.*HandshakeAccept` data
  - Both peers instantiate a `ConversationP2P` to each other

Managing a Session (`ConversationP2P`)

- Track liveness with `Ping` and `Pong`
- Update configuration on `.*HandshakeAccept`
- Push new configuration via `Handshake`
- Discover public IP address with `NatPunchRequest` / `NatPunchReply`
- Measure and throttle bandwidth usage
- Prevent relay cycles / enforce TTL
- Identify “choke points” in the topology
- Track message arrivals to find less-connected nodes for relay
  - “If we regularly get blocks from peer P, then don’t push blocks so often to P”
- Keep PeerDB row up-to-date with `.*HandshakeAccept`

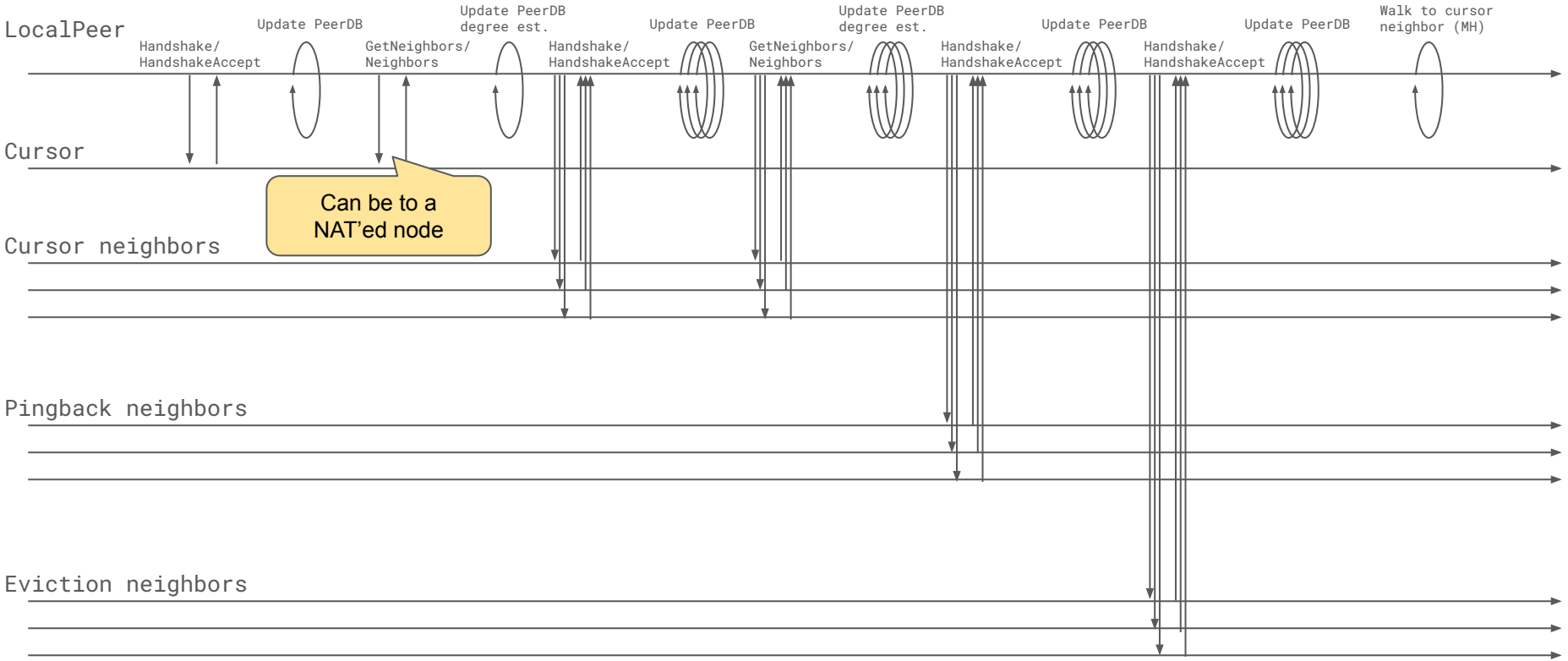
`NeighborWalk`: Building the P2P Map

- Map all peers to which the local peer ever opened a session
- Built up opportunistically on receipt of `.*HandshakeAccept`
- IP address/port, public key, public key expiration, etc.
- Cannot grow unbounded without DDoS risk
- `INSERT`
  - Hash neighbor to eight keys in 2^{24} key space, and insert into the first free key
  - If there are no free keys, then select an occupied key at random for eviction
- `EVICT`
  - Exploit the Lindy Effect – prefer to remember peers that are often online
  - Ping the peer's row before evicting – if it's still online, then drop the newcomer
- Lessons learned from Bitcoin's neighbor discovery algorithm

`NeighborWalk`: Building the P2P Map

- Random walk of peer graph to build a representative sample
  - Distributed variant of Metropolis-Hastings graph sampling
  - Track node degrees and node's ASN's representation in PeerDB
- Node "cursor" steps randomly through peer graph
  - P[step from cursor *C* to neighbor *N*] ~ *C*.degree() / *N*.degree()
  - P[step from cursor *C* to neighbor *N*] ~ *C*.ASN_count() / *N*.ASN_count()
- Peer discovery establishment through several means
  - A neighbor of a known neighbor
  - An unauthenticated peer that contacted us recently ("pingback")
  - A neighbor of an inbound peer (i.e. a NAT'ed peer) with an established session
  - A candidate for PeerDB eviction

NeighborWalk: Building the P2P Map



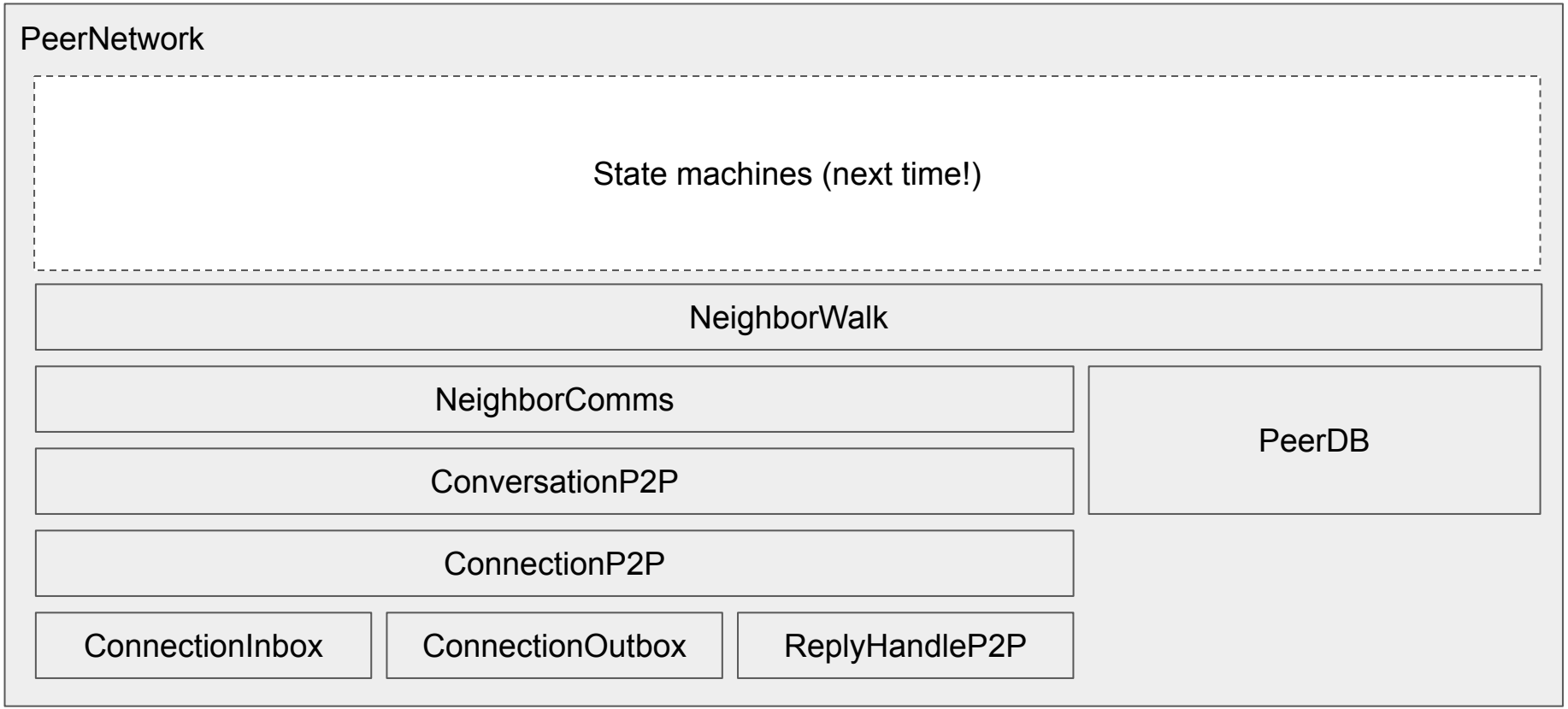
`PeerNetwork`: Session Lifecycle Management

- Drives all socket I/O and sessions via `epoll(2)` wrapper
  - `ConversationP2P` paired with `PeerNetwork`-managed non-blocking socket
  - Inner `ConnectionP2P` consumes `StacksMessage`s from read-ready sockets to inbox
  - Inner `ConnectionP2P` sends `StacksMessage`s to write-ready sockets from outbox
  - `ConversationP2P` consumes control-plane msgs and yields data-plane msgs
- Drives all state machines, including `NeighborWalk`
  - Use inner `NeighborComms` instance to manage sessions
  - `NeighborComms` will re-use open sessions in `PeerNetwork`
- Curates active set of P2P sessions
  - In quiescent state, 32 outbound sessions and 256 inbound sessions (configurable)
- Probabilistically closes unused but open sessions
  - P[session to peer *P* is closed] ~ 1.0 - “*P*’s normalized health score”
  - P[session to peer *P* is closed] ~ 1.0 - “#peers in *P*’s AS” / “#peers”
  - State machines may need to keep sessions around; they “pin” and “unpin” sessions

`PeerNetwork`: Session Lifecycle Management

- State machines build `NeighborComms` from `PeerDB` and `PeerNetwork`
  - Select peers at random open sessions
  - Select peers at random that have recently been contacted
- State machines report dead or misbehaving peers to `PeerNetwork`
  - `PeerNetwork` will ban them for exponentially-increasing amounts of time
- State machines report data to `NetworkResult` for relayer thread
- Other threads can issue `PeerNetwork` directives
  - Broadcast message, advertize data availability, ban peers

Conceptual Map So Far



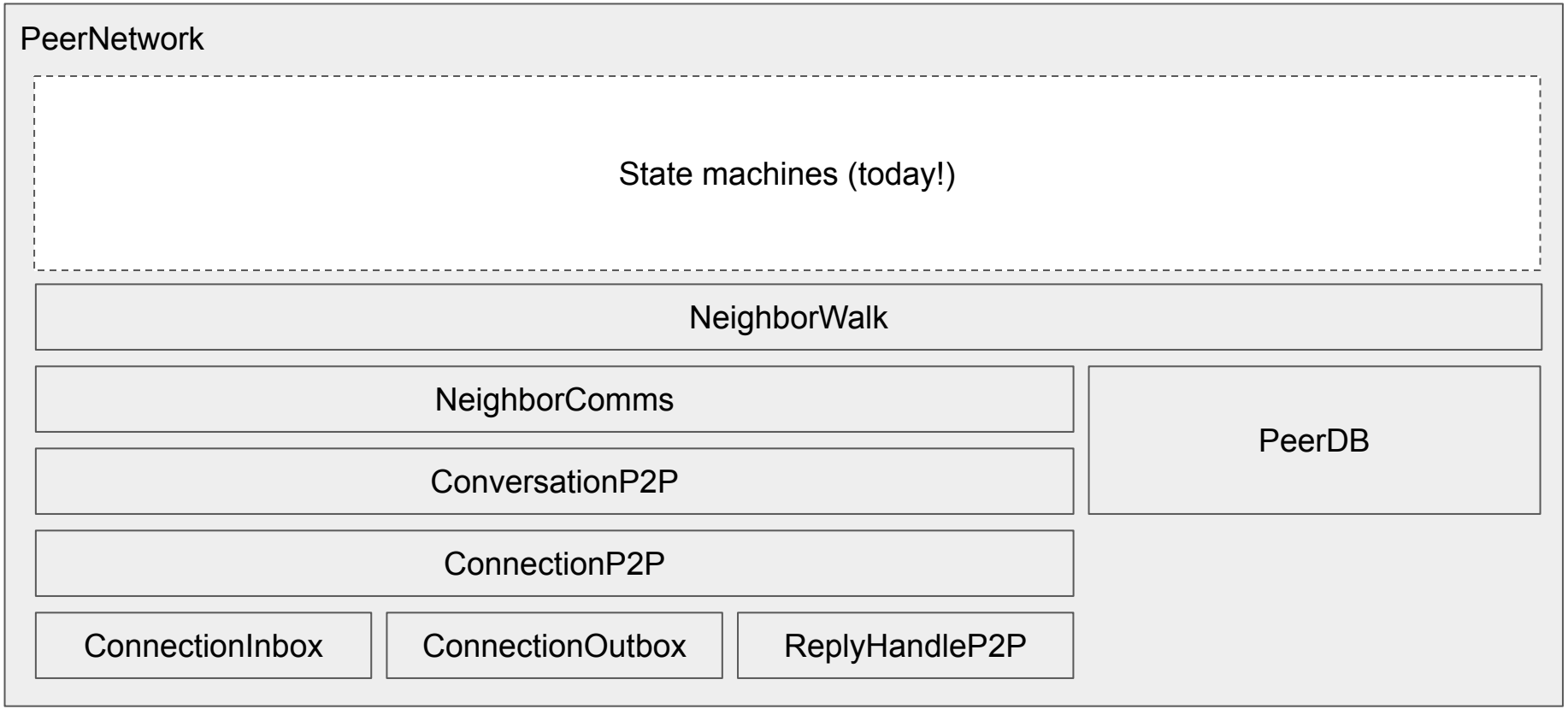
Stacks Academy

Day 7

Design of the Stacks P2P Network (part 2)

- State Machines (today)
- Code walk-through (tomorrow)

Conceptual Map So Far



Nakamoto State Machines

- `NeighborWalk` (last time) – can’t do anything without peers!
- `NakamotoInvStateMachine`
- `NakamotoDownloadStateMachine`
- `MempoolSync`
- `StackerDBSync`

Block Synchronization

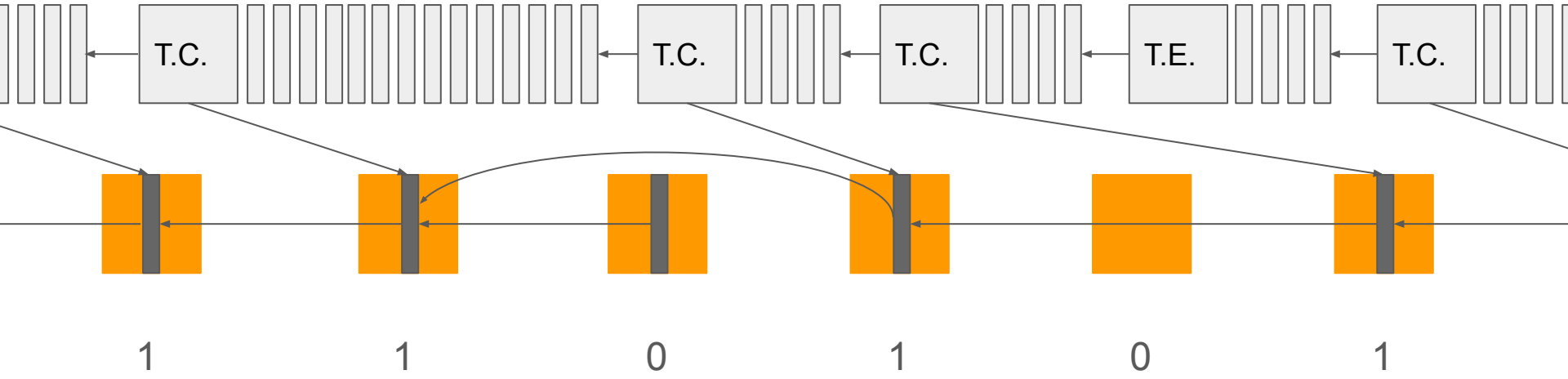
- #1 priority of any blockchain's p2p network
- Uses sortition history as a "shared codebook"
  - Akin to an append-only `.torrent` file
- **Inventory**: a bit vector summary of which tenures are present
- Use peers' inventories to find and download rare tenures
  - Via HTTP interface, learned in `.*HandshakeAccept`

`NakamotoInvStateMachine`

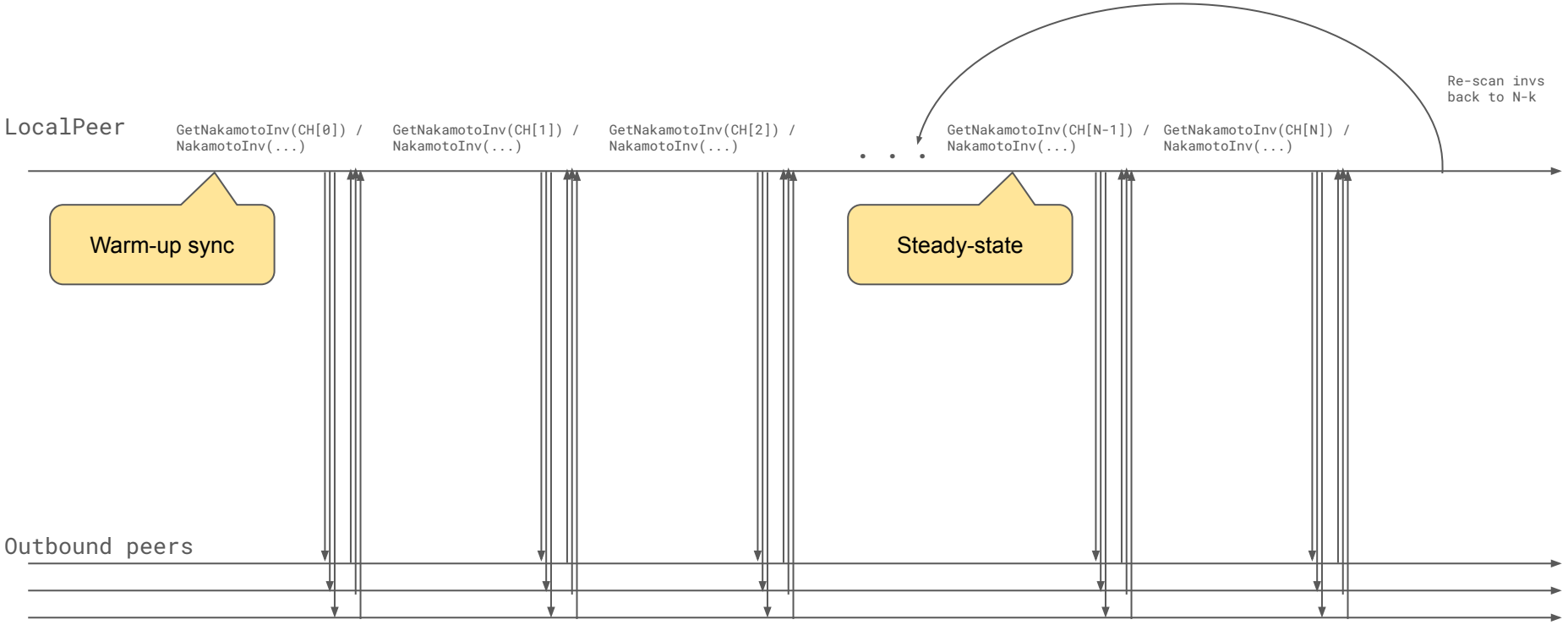
- Computes tenure inventory per reward cycle
  - Bit `i` indicates whether or not the `i`th tenure is locally present
  - Only represents confirmed tenures
- Queries each connected peer for its inventories for all reward cycles
  - Initial sync on boot-up
  - Only queries past `[connection_opts].inv_reward_cycles` inventories after
  - Only queries bootstrap peers in IBD
- `GetNakamotoInv` – query inventory at a reward cycle
  - `consensus_hash`: reward cycle's starting sortition identifier
- `NakamotoInv` – reward cycle tenure inventory
  - `BitVec<2100>`: 2100-member bitfield

NakamotoInvStateMachine

0: we do NOT have a tenure starts with this consensus hash
1: we have a tenure that starts with this consensus hash

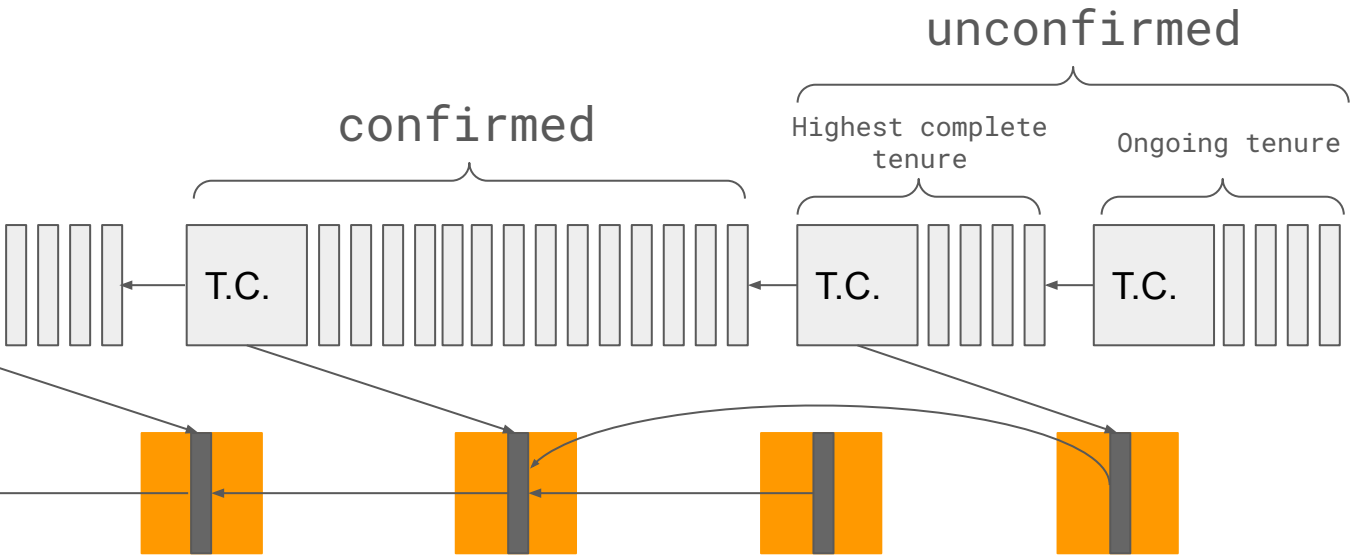


NakamotoInvStateMachine



NakamotoDownloadStateMachine

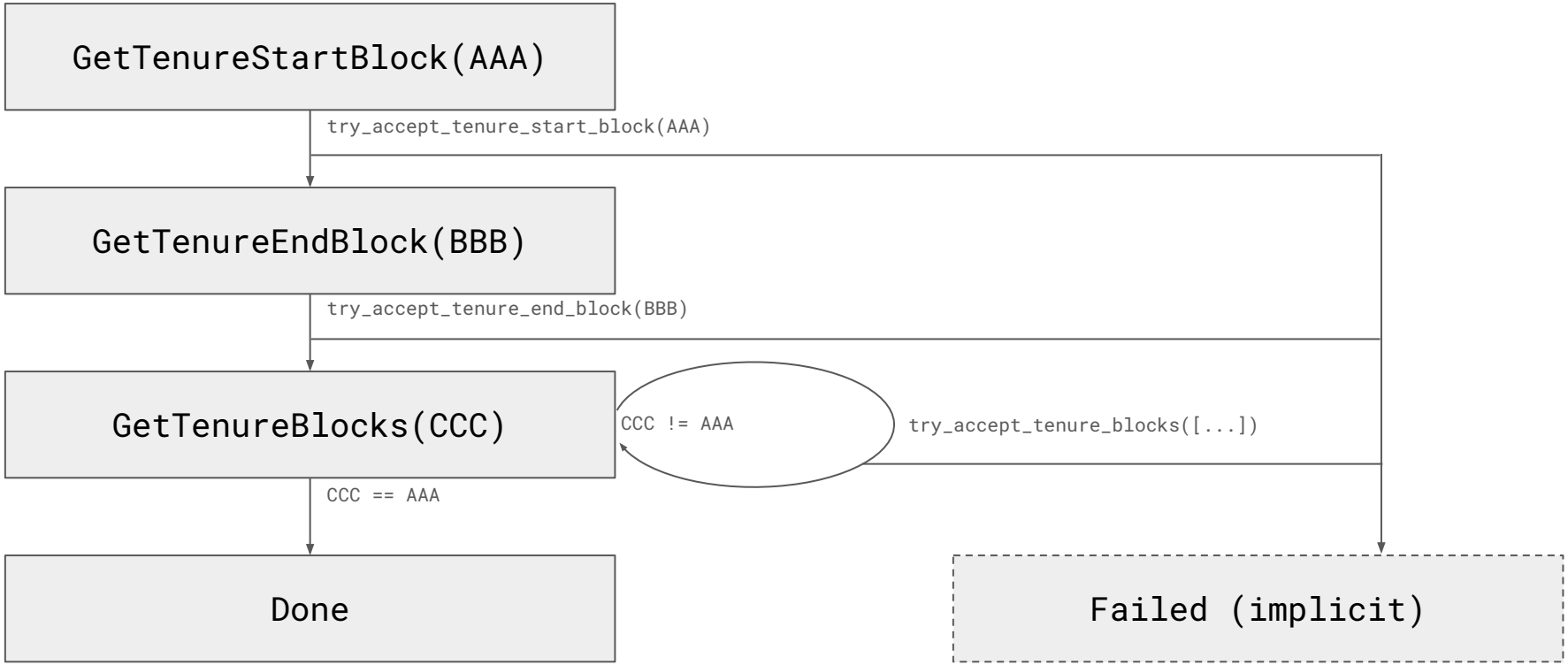
- Use NakamotoInvs to deduce which tenure(s) are remotely available
- **Confirmed tenure**: hash of a child tenure exists on Bitcoin
- **Unconfirmed tenure**: hash of a child tenure does NOT exist on BTC



NakamotoDownloadStateMachine

- `WantedTenure` – a (consensus hash, winning block-commit hash)
- `TenureStartEnd` – start and end block IDs
  - `TenureStartEnd::from_inventory()` – compute all confirmed tenures in current and penultimate reward cycles, given their lists of `WantedTenure`s
- `AvailableTenures` – `HashMap<ConsensusHash, TenureStartEnd>`
- `NakamotoTenureDownloader`
  - Confirmed tenure download state machine
- `NakamotoTenureDownloaderSet`
  - Instantiates and runs a set of `NakamotoTenureDownloader`s
- `NakamotoUnconfirmedTenureDownloader`
  - Unconfirmed tenure download state machine

NakamotoTenureDownloader



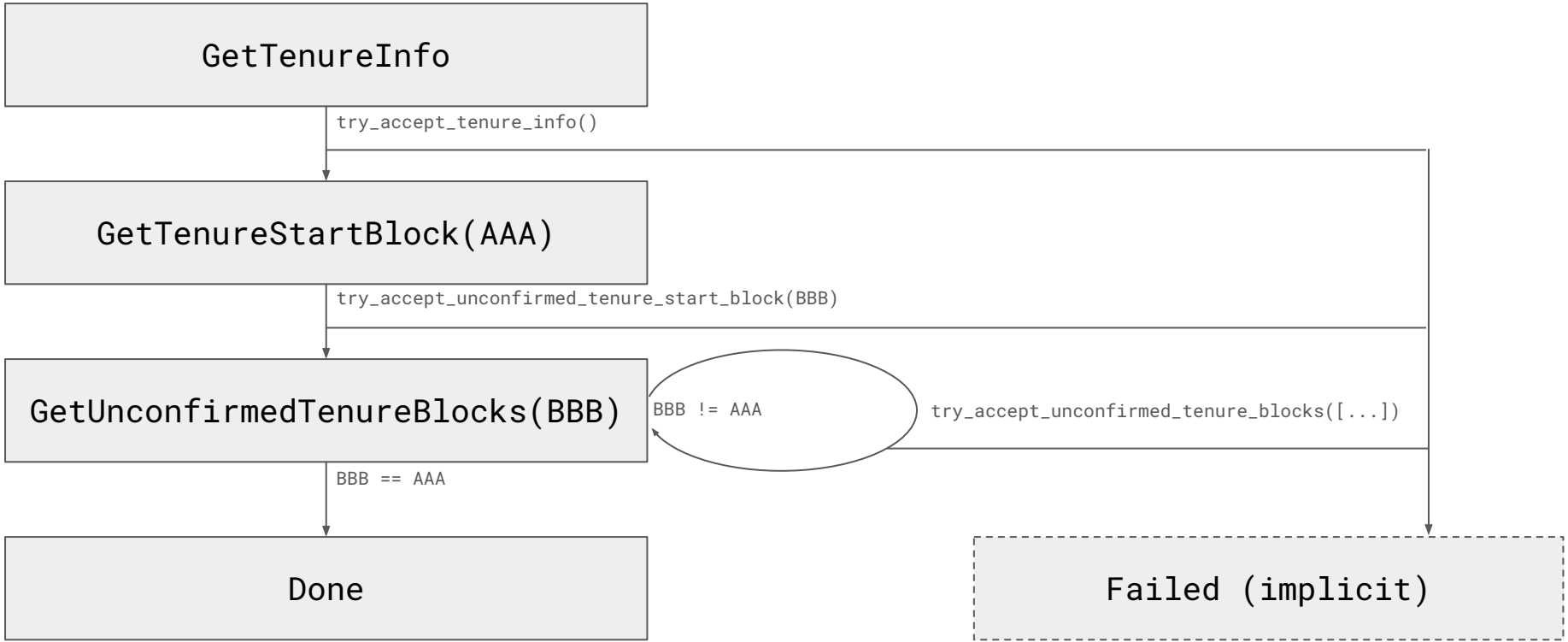
NakamotoTenureDownloader

- `make_next_download_request()`
  - Convert state into an HTTP request
- `try_advance_from_chainstate()`
  - See if we got relevant block data by another means (e.g. the relayer)
- `handle_next_download_response()`
  - See if we got relevant block data from a download request we made

NakamotoTenureDownloaderSet

- Instantiate `NakamotoTenureDownloaders` for tenures we don't have
  - `schedule: VecDeque<ConsensusHash>`
  - `available: HashMap<ConsensusHash, Vec<NeighborAddress>>`
  - `tenure_block_ids: HashMap<NeighborAddress, AvailableTenures>`
- Schedule RPC requests from `NakamotoTenureDownloaders`
  - At most one in-flight RPC request per peer
  - At most *k* `NakamotoTenureDownloaders` at once
  - Track which tenures have been successfully fetched (even if not stored)
- Run RPC requests
  - `NeighborRPC` – like `NeighborComms`, but for HTTP requests
  - Similar API – `.has_inflight()`, `.send_request()`, `.collect_replies()`, `.add_dead()`, `.add_broken()`, `.is_dead_or_broken()`

NakamotoUnconfirmedTenureDownloader



NakamotoUnconfirmedTenureDownloader

- `make_highest_complete_tenure_downloader()`
  - Instantiate a `NakamotoTenureDownloader` for the highest complete tenure, now that we have its end-block (i.e. the start-block of the ongoing tenure)
- `try_advance_from_chainstate()`
  - See if we got relevant block data by another means (e.g. the relayer)
- `make_next_download_request()`
  - Convert state into an HTTP request
- `handle_next_download_response()`
  - See if we got relevant block data from a download request we made

`NakamotoDownloadStateMachine`

- Compute and maintain view of chainstate and block availability
  - `Vec<WantedTenure>` for current and penultimate reward cycles
  - `HashMap<NeighborAddress, AvailableTenures>` – who can serve which tenure
  - Tenure download schedule (rarest first)
- Drive inner state machines
  - `NakamotoTenureDownloadSet`
  - `HashMap<NeighborAddress, NakamotoUnconfirmedTenureDownloader>`
- Determine when to switch between confirmed and unconfirmed
  - To unconfirmed if we have all confirmed tenures available to us
  - To confirmed if we instantiate a downloader for the highest complete tenure

Mempool Synchronization

- #2 priority of any blockchain’s p2p network
- Transactions need to find their way to miners
- One-off broadcast isn’t good enough
    - Miners may not be publicly routable

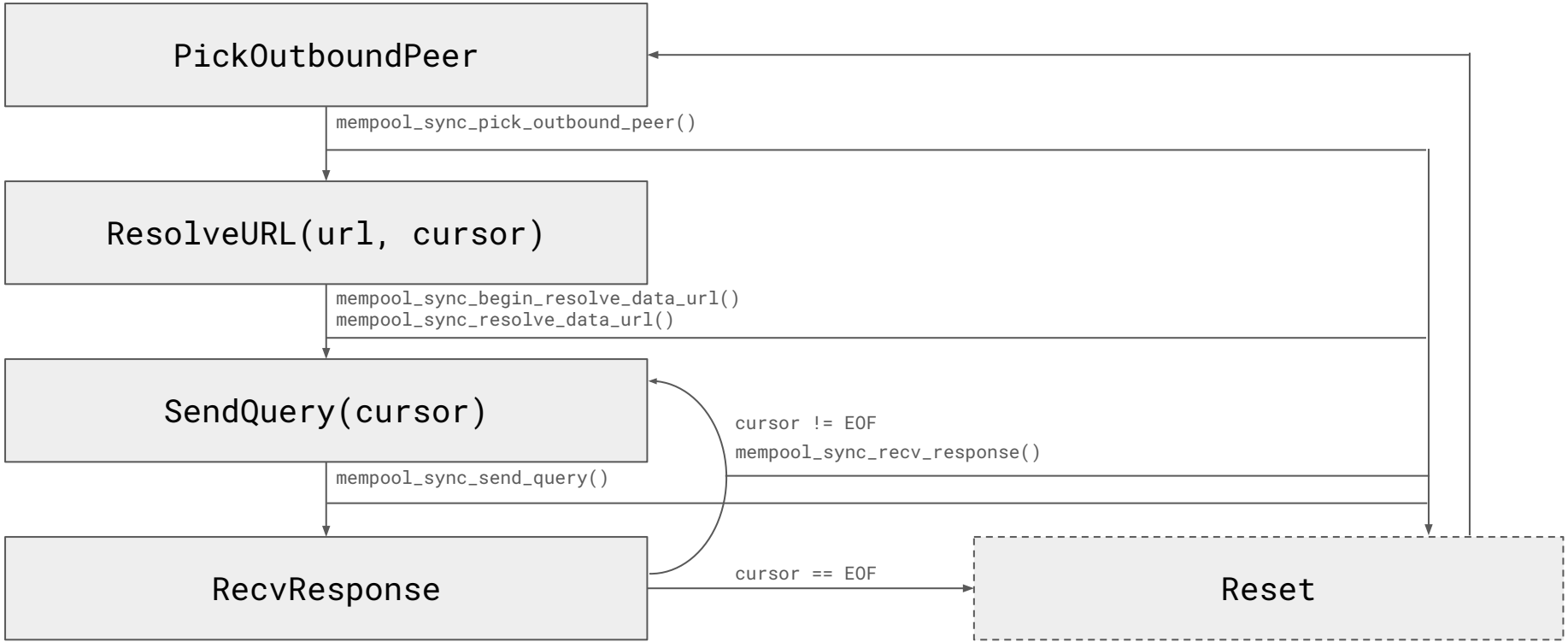
MempoolSync

- Mempool anti-entropy protocol
  - Exploit latent bipartite peer graph structure
  - Push to public node, public nodes sync, NAT’ed miner downloads
- Compute “sketch” of local mempool
- Select a random remote outbound peer
- Compute set difference with remote peer’s “sketch”
- Download remote peer’s novel transactions in random order

MempoolSync

- Pagination order is randomized with peer-local state
  - Don’t want to favor / induce a PoW game on txids
  - `MemPoolDB::find_next_missing_transactions()`
- `MemPoolSyncData::TxTags`
  - 8-byte `hash(TXID)` prefix of “recent” transactions in the mempool
    - Arrived no more than 2 coinbases ago
  - Contains `hash()` function and list of 8-byte prefixes
- `MemPoolSyncData::BloomFilter`
  - When there are too many tags to send
  - Compression of a counting bloom filter maintained by mempool admission logic
  - Only transactions from 2 coinbases ago are represented
  - Bloom filter keys are also `hash(TXID)`, so send `hash()` function as well

MempoolSync



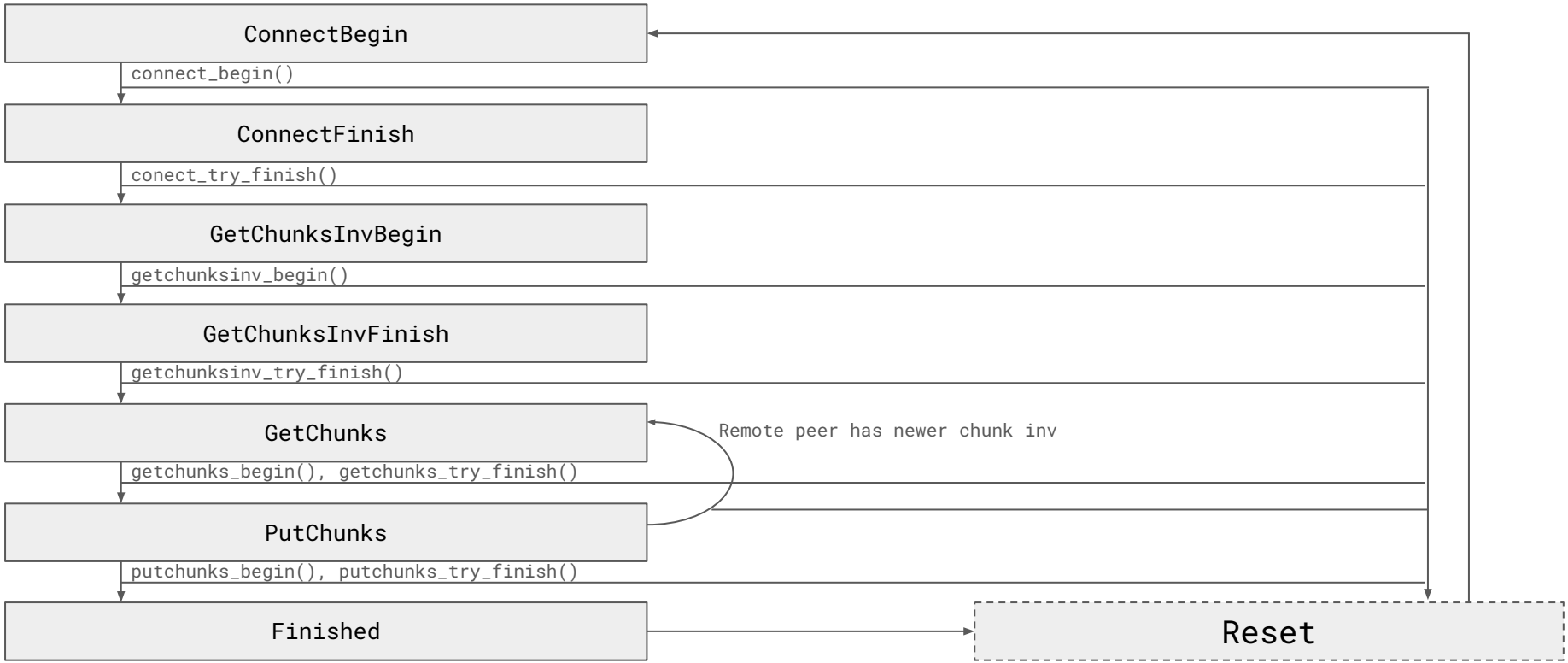
StackerDB Synchronization

- Unique to Stacks
- Part of block production
- Need quick, reliable all-to-all replication for each replica

`StackerDBSync`

- StackerDB anti-entropy protocol
  - Like `MempoolSync`, exploit latent bipartite peer graph structure
  - Push chunks to public nodes, public nodes sync, NAT'ed nodes download
- Find StackerDB replicas
- Get StackerDB inventories
- Get rarest novel remote chunks
- Push rarest novel local chunks

StackerDBSync



Legacy Epoch 2.x State Machines

- `InvState` – synchronize PoX, block, and microblock inventories
- `BlockDownloader` – download blocks and microblocks
- `Atlas` – synchronize transaction attachments
- `AntiEntropy` – push blocks and microblocks to needful peers

Stacks Academy

Day 8

Design of the Stacks P2P Network (part 3)

- Jude’s extra-spicy take on `async/await` (and why we don’t use it)
- Code walkthrough

Against `async/await`

1989 called. It wants its cooperative M:N thread model back.

Background: Parallelism

- **Task**: a sequence of instructions & state which are not interrupted
  - E.g. “task” in Rust or Javascript is a sequence of code between `await`-ed operations
- Blocking operations make parallel task execution hard
  - Suspends the OS thread, even if subsequent tasks are concurrent
  - But, we want maximum degree of parallel execution for concurrent tasks!
  - Not just I/O – also includes mutual exclusion, timers, and so on – things that block
  - How do we run some other (runnable) tasks while a given task is stalled?
- Wanted: a way to schedule tasks around blocking operations
  - Insight: not all tasks are blocked at a time, and many tasks are concurrent
- Tasks are (usually, but not always) encapsulated into threads
  - Alternative to threads: callbacks, coroutines, continuations, etc

Background: Threading

- Threading models
  - **1:N**: 1 kernel-space process shared by N user-space threads
    - Python, Perl, “green threads”
  - **1:1**: 1 kernel space process per user-space thread
    - (modern) pthreads, (modern) Java
  - **M:N**: M kernel-space processes shared by N user-space threads
    - Rust, Go, Erlang, Haskell, Gambit Scheme
- Why M:N threading?
  - Space-efficient (1 user-space thread’s context is far smaller than 1 VM page)
  - Fast context-switching within a kernel thread (no need to save registers)
  - Maximal parallelism if M == #cpus
  - Some degree of portability – not directly dependent on OS-specific thread behavior

Background: Scheduling

- Background: scheduling models for tasks
  - **Cooperative** – a user-space task (thread) explicitly yields to another
    - Rust, Javascript
    - *Explicit* tasks – business logic encodes scheduler wakeups
  - **Preemptive** – the runtime switches between user-space tasks (threads)
    - (modern) Go, Erlang, Haskell, Gambit Scheme
    - *Implicit* tasks – scheduler decides task boundaries
- `async/await`: cooperative task scheduling in M:N thread model
  - `await` yields to the next task

`async/await` is a symptom of bigger problems

Against cooperative M:N threads

Extremely leaky parallelism

- CPU-bound tasks will stall other *unblocked* tasks
  - Eats up the OS thread's quanta
  - Not *all* other tasks; just ones using the same OS thread (partial degradation)
- A task can (still) block by forgoing a scheduler wakeup
  - Also starves all other tasks on the same OS thread
- Each blocking operation needs explicit scheduler wakeup
  - Liveness is a global property, so requires treating *all dependencies*
  - Leads to dependencies requiring specific runtimes (`tokio`)
- **It is hard to reliably achieve good parallelism**
  - Great for concurrent I/O-heavy CPU-light workloads, but bad at everything else
  - Doesn't magically speed up heterogeneous workloads; it may slow them down

It didn’t have to be this way

In praise of preemptive M:N threads

- Tasks are preempted by the user-space scheduler
  - CPU-bound tasks do not starve other tasks in the same OS thread
- A task cannot prevent or forgo a scheduler wakeup
  - Eventually, every task will be scheduled, *no matter what*
- A blocking operation only puts the calling *task* to sleep
  - Other unblocked tasks get scheduled automatically
  - Can defer long-running blocking tasks to dedicated threads (e.g. I/O threadpool)
- No special `async/await` keywords or type annotations needed
- **Achieves parallelism with minimal effort**
  - Works in any kind of workload, akin to kernel-level preemptive multitasking
  - You can still do tuning, but it’s not required to fulfill the promise of multithreading

Why aren’t preemptive M:N threads common?

- FFI is hard, but solvable
  - Tasks are stackless, but FFI requires following “stackful” C calling convention
  - Solvable – create a temporary C-style stack at the callsite
- Scheduling is hard, but solvable
  - The kernel and linked libraries don’t know about your tasks
  - Linked libraries accessed through FFI aren’t generally reentrant
    - E.g. You could preempt a task while it’s in FFI-C code, while it’s holding a lock
  - Solvable – don’t allow other tasks to enter the same FFI-C code when suspended
    - Scheduler needs to emulate C-style pre/post conditions for FFI calls
    - Can run FFI in separate, dedicated OS task
- Misguided (IMHO) belief that programmers *want* to schedule tasks
  - Cooperative M:N offers very high degree of control over task atomicity
  - Rust does not want a built-in scheduler (ironic because `tokio` is so pervasive)
  - Unforced design error

So where does that leave us?

- Cooperative M:N threading is not useful to us
- Preemptive M:N threading isn’t available in Rust
- How hard is it to roll your own task parallelism?
  - Not hard at all! Rust offers “fearless concurrency”
  - P2P system offers hand-rolled but easy-to-use abstractions

Code Walkthrough

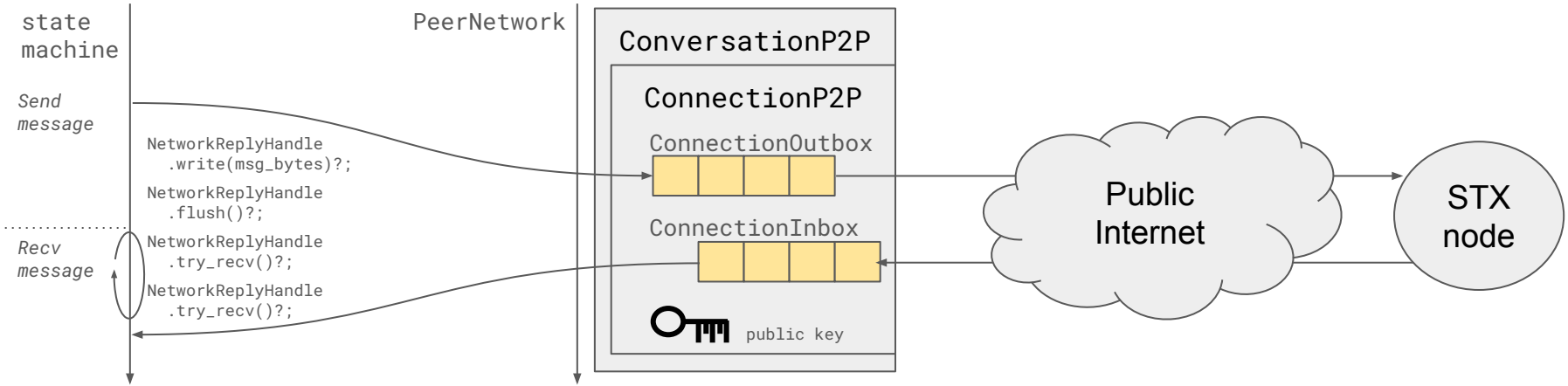
We don’t need `async/await`

Task parallelism without `async/await`

- `async/await` schedules tasks around blocking operations
  - Compiles a “function” to a state machine of tasks
  - State transitions happen at `.await`
  - Tasks get scheduled at a state transition
- We schedule blocking operations around tasks
  - Dispatch blocking operations to other threads (but we’re agnostic)
  - Get back a handle to wait in, or poll on
  - A function’s execution blocks only when its subsequent tasks cannot make progress without the completion of a blocking operation (true for `async/await` as well)
- Why does this work?
  - State machines can run in parallel → we can just multi-thread
  - Threadpool for offloading CPU-intensive tasks from the state machines
  - Socket I/O polling with (1) message hand-off to send and (2) dispatch to state machine on receive

Task parallelism without `async/await`

- `PipeRead`, `PipeWrite` – connect two stack frames, anywhere
  - Senders use a `PipeWrite` to feed messages to P2P (which has the `PipeRead`)
  - Receivers have a channel receiver for `StacksMessageType` (P2P has sender)
- `NetworkReplyHandle` – send/receive msgs between stack frames
  - `Send` and `Sync` safe – can be passed to other threads



State Machine API

- State machines have top-level `run()` method to loop through steps
  - State machines are “linear” so code maps closely to a sequential program
- `NeighborComms`
  - `.is_neighbor_connecting()`
  - `.has_neighbor_session()`
  - `.neighbor_session_begin()`
  - `.neighbor_send()`
  - `.collect_replies()`
- `NeighborRPC`
  - `.has_inflight()`
  - `.send_request()`
  - `.collect_replies()`

State Machine Invocations

- `PeerNetwork::do_network_neighbor_walk()`
  - `self.walk_peer_graph()` – Drive `NeighborWalk`
    - Bookkeeping about number and kinds of walks
    - State machine throttling
  - `NeighborWalk::run()` – `NeighborWalk` state machine
- `PeerNetwork::run_mempool_sync()`
  - `MempoolSync::run()` – `MempoolSync` state machine
- `PeerNetwork::run_stacker_db_sync()`
  - `StackerDBSync::run()` – `StackerDBSync` state machine for each replica

State Machine Invocations

- `PeerNetwork::do_network_work()` – Nakamoto data-plane
  - `do_network_work_nakamoto()` – Nakamoto-specific behaviors
    - `do_network_inv_sync_nakamoto()` – run `NakamotoInvStateMachine`
    - `do_network_block_sync_nakamoto()` – run `NakmatoBlockDownloader`
    - `network_result.consume_nakamoto_blocks()` – pass blocks to relayer
  - `do_network_work_epoch2x()` – run legacy Stacks 2.x state machines

PeerNetwork

- `PeerNetwork::run()`
  - `network.poll()` – get edge-triggered read/write states for all sockets
    - Wrapps `mio`
  - `refresh_burnchain_view()` – reload p2p thread's view of chainstate
    - Stacks tip, sortition tip, cached PoX reward sets, last anchor block
    - `BurnchainView` – used to construct p2p messages
  - `Self::with_attachments_downloader()` – run `Atlas` state machine
  - `http.run()` – run HTTP server pass
  - `dispatch_network()` – run P2P server pass

PeerNetwork

- `http.run()`
  - `process_new_sockets()-` set newly-active sockets to connecting
  - `process_connecting_sockets()` – set connecting sockets to ready
  - `process_ready_sockets()` – drive socket I/O
  - `flush_conversations()` – consume msgs from other stack frames
  - `clear_timeouts()` – drop connections on request timeout
  - `disconnect_unresponsive()` – drop connections if peer is too idle

PeerNetwork

- `PeerNetwork::dispatch_network()`
  - `process_new_sockets()` – set up connecting sockets
  - `process_connecting_sockets()` – set up newly-connected sockets
  - `process_ready_sockets()` – read data from sockets and write new data
  - `do_network_work()` – drive data-plane state machines
  - `prune_connections()` – garbage-collect excessive `ConversationP2P`s
  - `do_network_neighbor_walk()` – run `NeighborWalk`
  - `run_mempool_sync()` – run `MempoolSync`
  - `run_stacker_db_sync()` – run `StackerDBSync`
  - `disconnect_unresponsive()` – drop idle or “stuck” connections
  - `queue_ping_heartbeats()` – send heartbeats to neighbors if needed
  - `flush_relay_handles()` – read data from other stack frames & write to sockets
  - `update_relayer_stats()` – maintain forwarding inferences about neighbors
  - `dispatch_requests()` – handle P2P requests from other threads

PeerNetwork

- `PeerNetwork::process_ready_sockets()`
- read data from sockets and write new data
  - `process_p2p_conversation()` – drive `ConversationP2P`
    - `convo.recv()` – get new `StacksMessageType`(s) from `ConnectionP2P`
      - `ConnectionP2P::recv_data()`
      - Puts `StacksMessageTypes` into inbox for receiver to dequeue
    - `convo.chat()` – handle control-plane msgs & pass up data-plane msgs
    - `convo.send()` – send out bytes in `ConnectionP2P`
      - `ConnectionP2P::send_data()`
      - Writes `StacksMessageTypes` into outbox for P2P to dequeue
  - Buffer up received messages
- Forward unsolicited (pushed) messages to `NetworkResult`

PeerNetwork

- `ProtocolFamily` – Abstraction over P2P and HTTP messages
  - `Type Preamble` – P2P message header or HTTP headers
  - `Type Message` – P2P payload or HTTP payload
  - `NetworkConnection<P: ProtocolFamily>` – protocol-specific message channels
    - `ConnectionP2P` and `ConnectionHttp` are type aliases

Future Work

- 1 thread for everything has been “good enough”
- State machines run in P2P thread, but should be separated out
  - Need to think about dependencies on `PeerNetwork` state
- RPC API handlers run in P2P thread, but could use a threadpool
  - This is a good first issue