ArcBlock Forge

A Framework to Build Reusable Interconnected Blockchains & Decentralized Apps

ArcBlock Inc.

About me

- VPE ArcBlock
- Father of two adorable princesses
- Coder (~200 repos, from tiny CLI to complicated systems)
- Writer (a book and ~500 posts)
- Runner (4:48:06, 2014 Beijing)

Agenda

- Blockchain intro: How trust is achieved? (10 min)
- A quick tour of Bitcoin and Ethereum (10 min)
- Forge framework: why and how (30 min)

Blockchain intro: how trust is achieved?

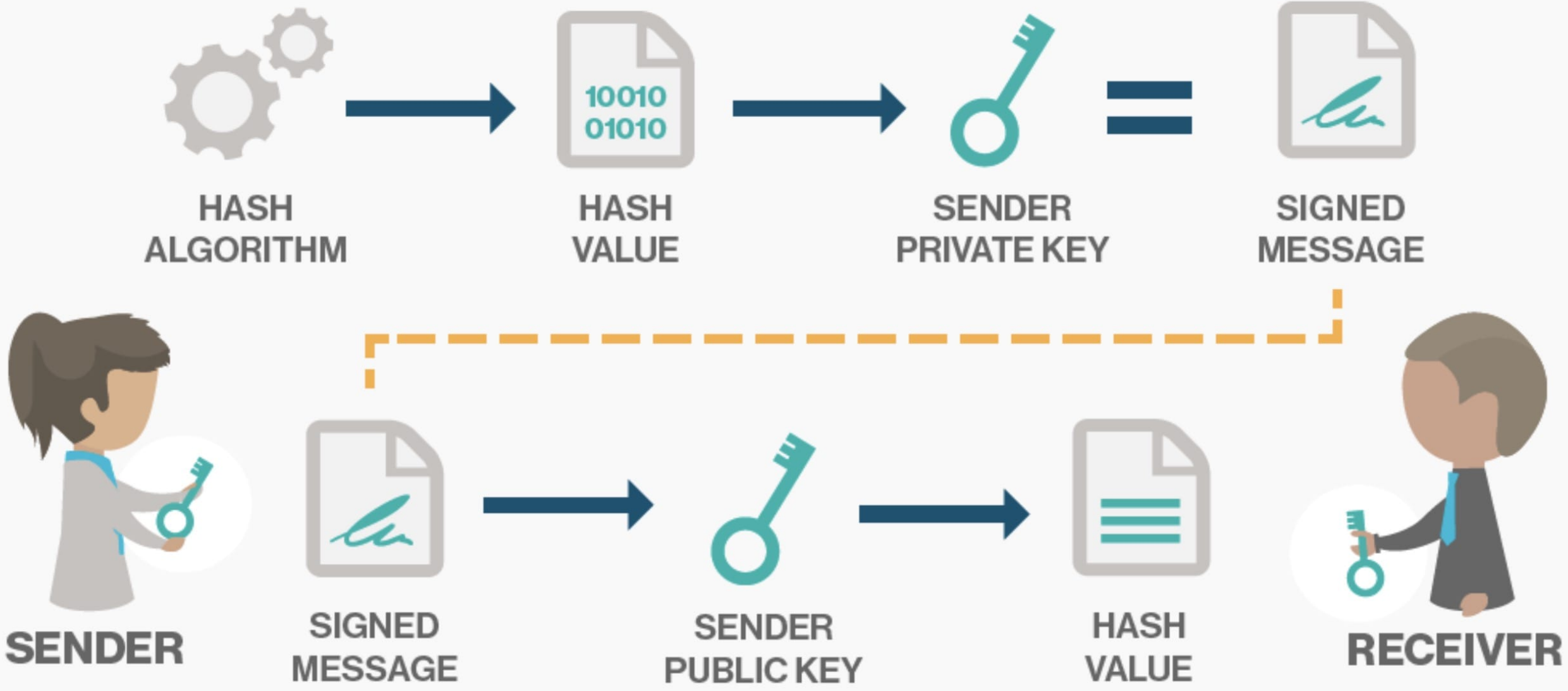
- Trust between different parties:
  - person / person
  - system (e.g. a bank) / person
  - system / system
- What's in common?
  - identification: **address**
  - anti-forgery: **signature**

To build trust among systems, we need the events to be public verifiable

Is this event public verifiable?

```
UPDATE accounts a
SET balance = a.balance + 100
WHERE a.id = 1;
```

Connect the dots: How to make an event public verifiable?


10010
01010
HASH ALGORITHM
HASH VALUE
SENDER PRIVATE KEY
SIGNED MESSAGE
SENDER
SIGNED MESSAGE
SENDER PUBLIC KEY
HASH VALUE
RECEIVER

What about a series, related events public verifiable?

- If everyone can keep a same ordered list of related events, then we got a public ledger
- To achieve that, all nodes in the network shall agree on the order

The public ledger with public verifiable state

- each event will be applied to current state and generate a new state
- the procedure that applies event to state is **deterministic**
- the event(s) shall have a way to take the hash of the calculated state

The rise of blockchain technology

- such system (manage to keep a public verifiable ledger) is called a **blockchain**
- the computer that runs blockchain software is called **node**
- a set of nodes consist of a **network**
- a public verifiable event being processed by the network is called **transaction** (tx)
- the container that contains a list of ordered events is called **block**
- the algorithm that form the agreement is called **consensus algorithm**
  - Nakamoto Consensus / pBFT / etc.
- upon executing the transaction, **state** evolves

How to verify the states?

$$S_{n+1} = f(S_n, [tx_1, tx_2, ..., tx_m]), tx_i \in B_{n+1}$$

$$B_{n+1}.app\_hash = hash(S_{n+1})$$

- S_n: state at block n
- tx_m: the transactions in block $n + 1$
- B_{n+1}: the new block that is being processed. A node will
- S_{n+1}: state after applying $n + 1$

Interesting facts on block

- A block is a container of a list of ordered txs
  - What if a block only contains 1 tx?
- A block is the minimum unit for transformation
  - txs inside a block either be processed as a whole or nothing
  - the minimum unit for rollback
- A block represents time in a blockchain
  - Time is a strict progression of cause to effect
  - Time is irreversible: past events and states are there unchangeable

Deterministic

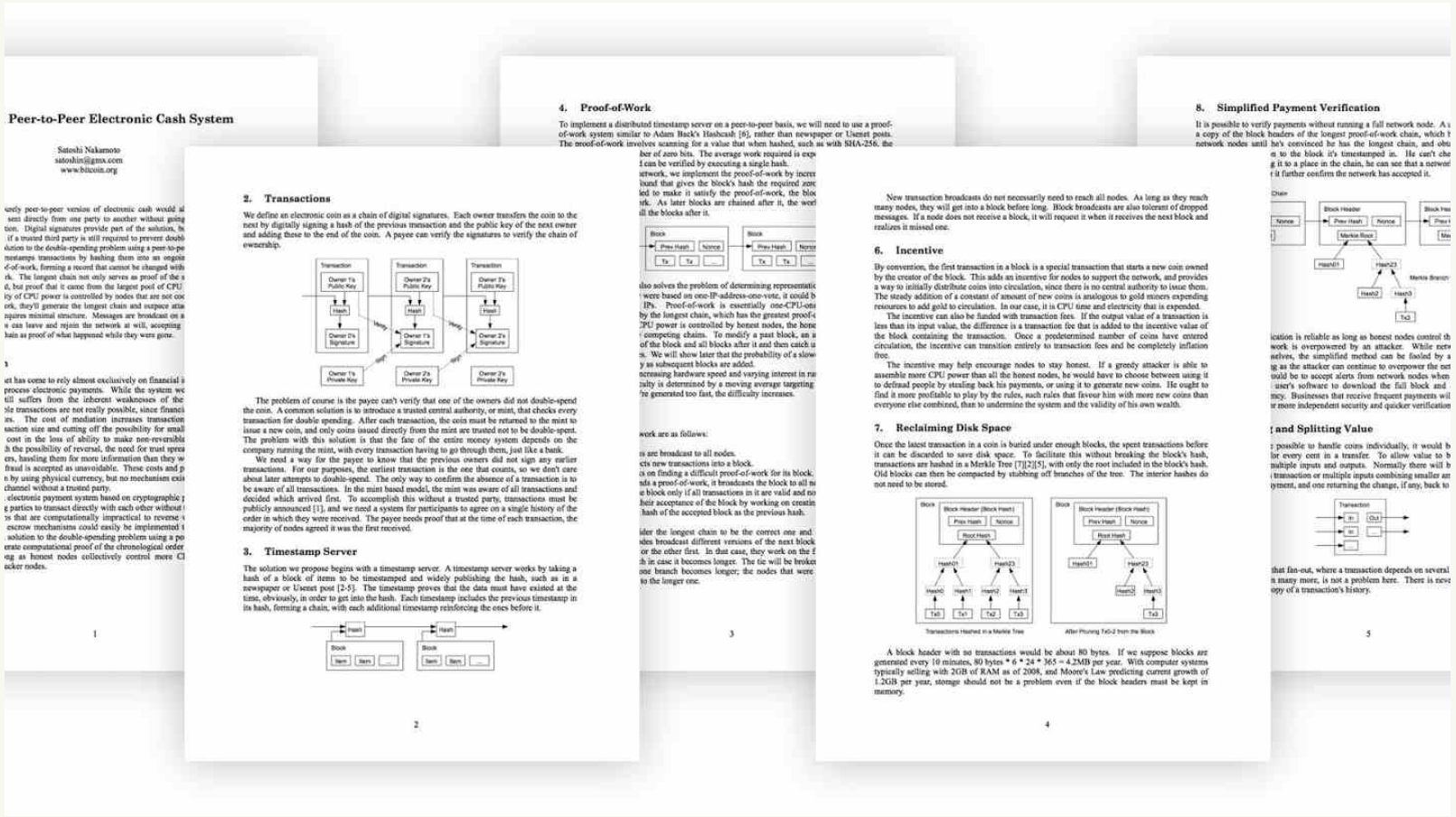
why is it so important?

- The code execute independently on each node
  - given a B_n and S_{n-1}, everyone shall deduce to S_n independently
  - otherwise you don't know who to follow and no consensus could be made
- Things to avoid:
  - random number generators (without deterministic seeding)
  - race conditions on threads (or avoiding threads altogether)
  - the system clock (you can't rely on it, e.g. the seed)
  - uninitialized memory (in unsafe programming languages like C or C++)
  - floating point arithmetic (generally not deterministic)
- language features that are random (e.g. iteration on map)

A quick tour to

Bitcoin / Ethereum

Bitcoin: p2p cash system



Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.

The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced [1], and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

3. Timestamp Server

The solution we propose begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash, such as in a newspaper or Usenet post [2-5]. The timestamp proves that the data must have existed at the time, obviously, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.

4. Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-of-work system similar to Adam Back's Hashcash [6], rather than newspaper or Usenet posts. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the

New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

6. Incentive

By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.

The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.

The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

7. Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the block's hash. Old blocks can then be compacted by stabbing off branches of the tree. The interior hashes do not need to be stored.

A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be kept in memory.

8. Simplified Payment Verification

and Splitting Value

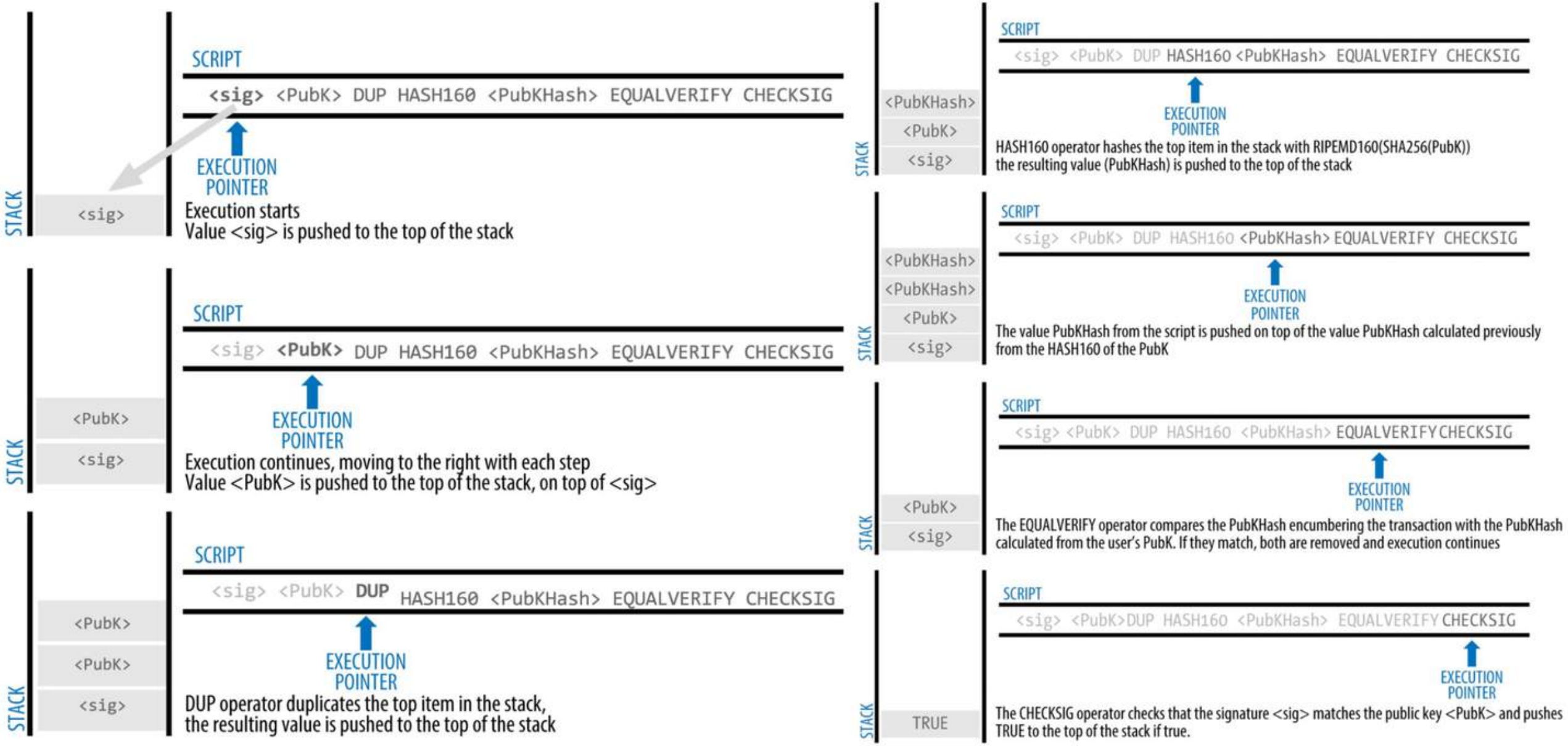
What is cash, anyway?

How Bitcoin forms a cash system?

- UTXO: Unspent TX Output. The "cash" in bitcoin system.
- wallet: an address backed by a key pair that anyone can send coin to it, and the owner of private key can consume its UTXO.
- Transaction: contains tx input & tx output on how "cash" is being spent
- State: UTXO pool (think about a god looking at every piece of cash in everyone's wallet)
- Transformation: stack based VM that execute the script based on the UTXO and current transaction to modify the UTXO pool.
  - previous transaction output: first half of the script
  - transaction input: second half of the script
- use PoW & Nakamoto Consensus to generate block and form consensus

What is a script, and why?

- Bitcoin script is Forth-like, stack-based, and processed from left to right. It is intentionally not Turing-complete.
- a list of instructions recorded with each transaction that describe how the next person wanting to spend the Bitcoins being transferred can gain access to them
- To support many use cases, a script backed by a VM made it extensible
  - a great design that inspires us for architecture
- Templates:
  - P2PK
  - P2PKH
  - P2MPK
  - P2SH
  - P2WPK
  - P2WPKH


STACK
<sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
Execution starts
Value <sig> is pushed to the top of the stack
STACK
<PubK>
<sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
Execution continues, moving to the right with each step
Value <PubK> is pushed to the top of the stack, on top of <sig>
STACK
<PubK>
<PubK>
<sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
DUP operator duplicates the top item in the stack,
the resulting value is pushed to the top of the stack
STACK
<PubKHash>
<PubK>
<sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
HASH160 operator hashes the top item in the stack with RIPEMD160(SHA256(PubK))
the resulting value (PubKHash) is pushed to the top of the stack
STACK
<PubKHash>
<PubKHash>
<PubK>
<sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
The value PubKHash from the script is pushed on top of the value PubKHash calculated previously
from the HASH160 of the PubK
STACK
<PubK>
<sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
The EQUALVERIFY operator compares the PubKHash encumbering the transaction with the PubKHash
calculated from the user's PubK. If they match, both are removed and execution continues
STACK
TRUE
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
The CHECKSIG operator checks that the signature <sig> matches the public key <PubK> and pushes
TRUE to the top of the stack if true.

Ethereum

A Next-Generation Smart Contract and Decentralized Application Platform

ArcBlock Inc.

In a nutshell

- Inspired and inherited many idea from Bitcoin
- 2nd generation of distributed ledger, with much more aspiration
- Has a built-in Turing complete VM called EVM
  - so it is a distributed computer
  - gas is used to solve the "halting problem"
- Allow custom code to be deployed into the node - smart contract
- A tx can trigger the execution of specific smart contract
- Uses PoW / Nakamoto Consensus as consensus at the moment
  - will switch to PoS / Casper in future

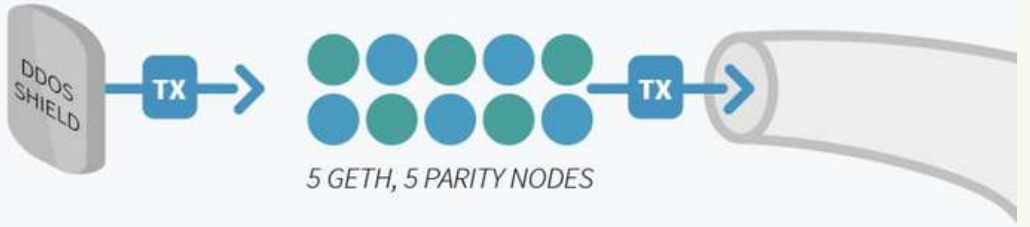
START HERE

1. You **sign** your transaction
2. You **send** your transaction

WITH MEW!

infura / etherscan /

3. Your transaction goes to a **MEW node...**



5 GETH, 5 PARITY NODES

***ABOVE** THIS LINE IS WHAT MEW IS RESPONSIBLE FOR*
BELOW IS JUST HOW THE BLOCKCHAIN WORKS.

5. Miners **pick transactions** from the pool

HIGH GAS PRICES PICKED FIRST!

4. ...who put **it in the pool** of all signed transactions

IF YOUR TX IS PENDING, IT'S IN THE POOL

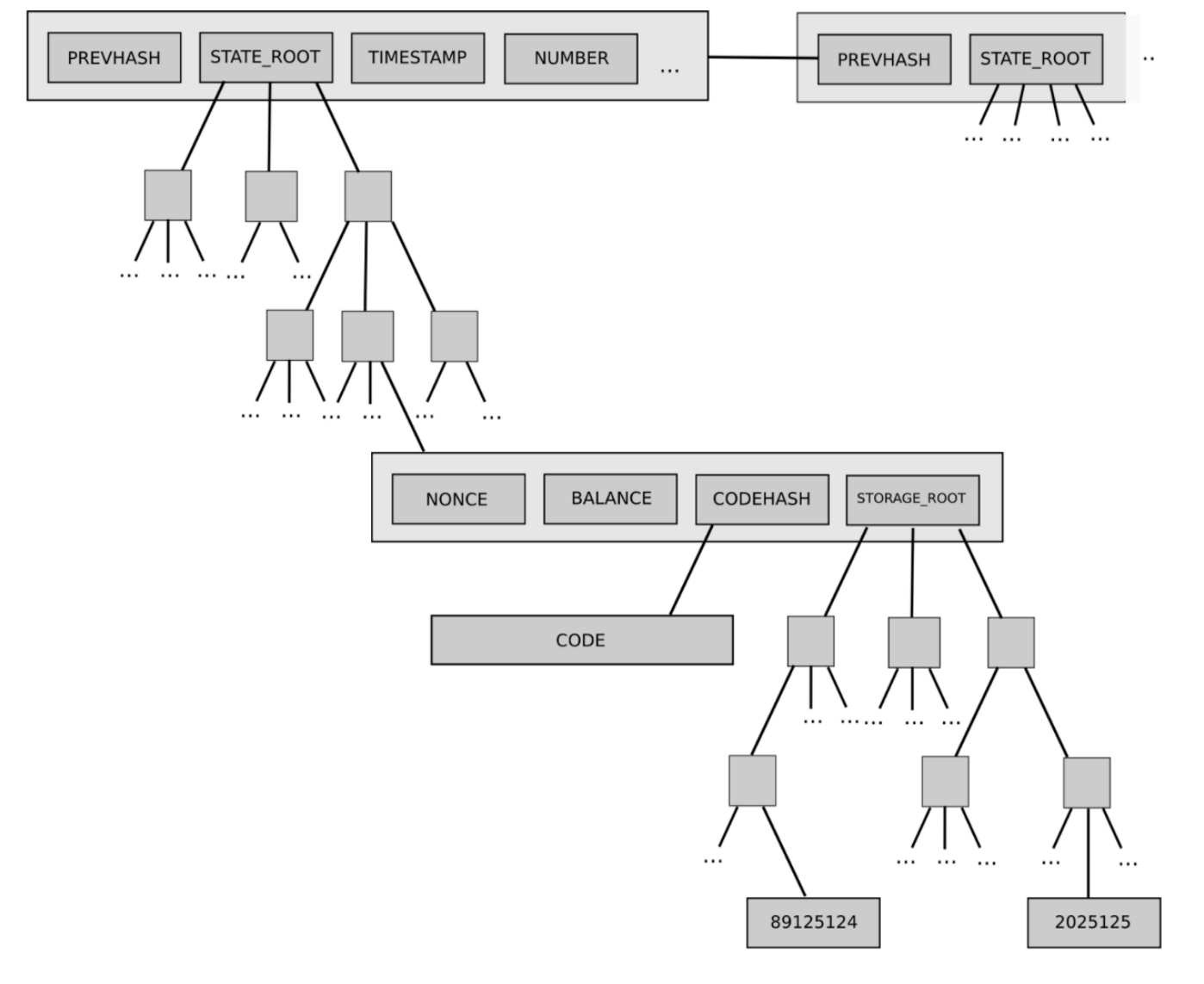
6. Miners then **put transactions in a block and add it to the chain**

ONCE IN THE BLOCKCHAIN, YOUR TX IS PERMANENT!

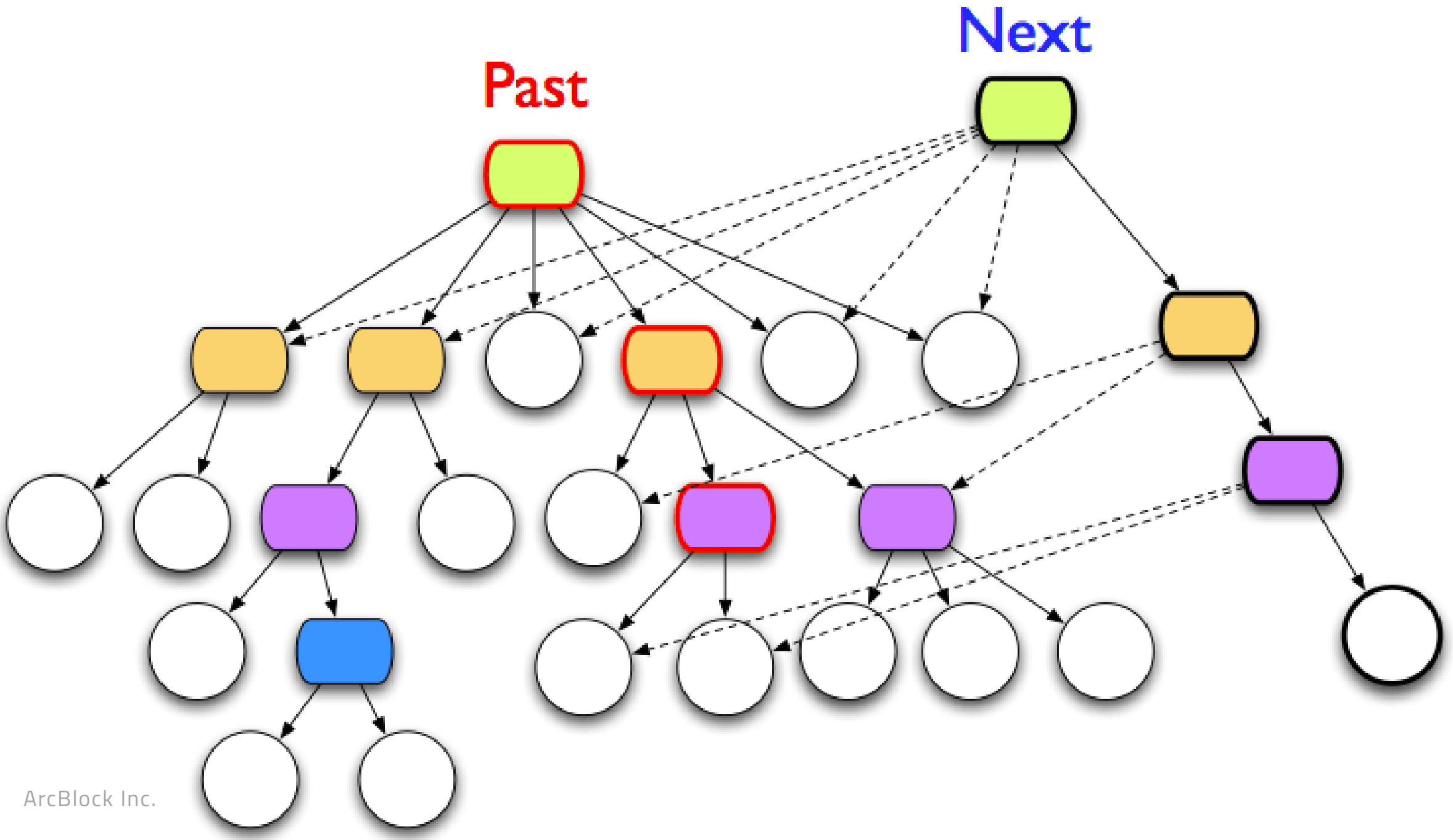
The states inside Ethereum account

- nonce: for anti-replay purpose
  - external account: txs sent by this account
  - contract account: number of contracts created by this account
- balance: number of wei owned by this address (1 ether = 1e+18 wei)
- storateRoot:
  - external account: hash of empty string
  - contract account: a hash of root node of a Merkle Patricia Tree which encodes storage contents
- codeHash:
  - external account: hash of empty string
  - contract account: hash of contract code

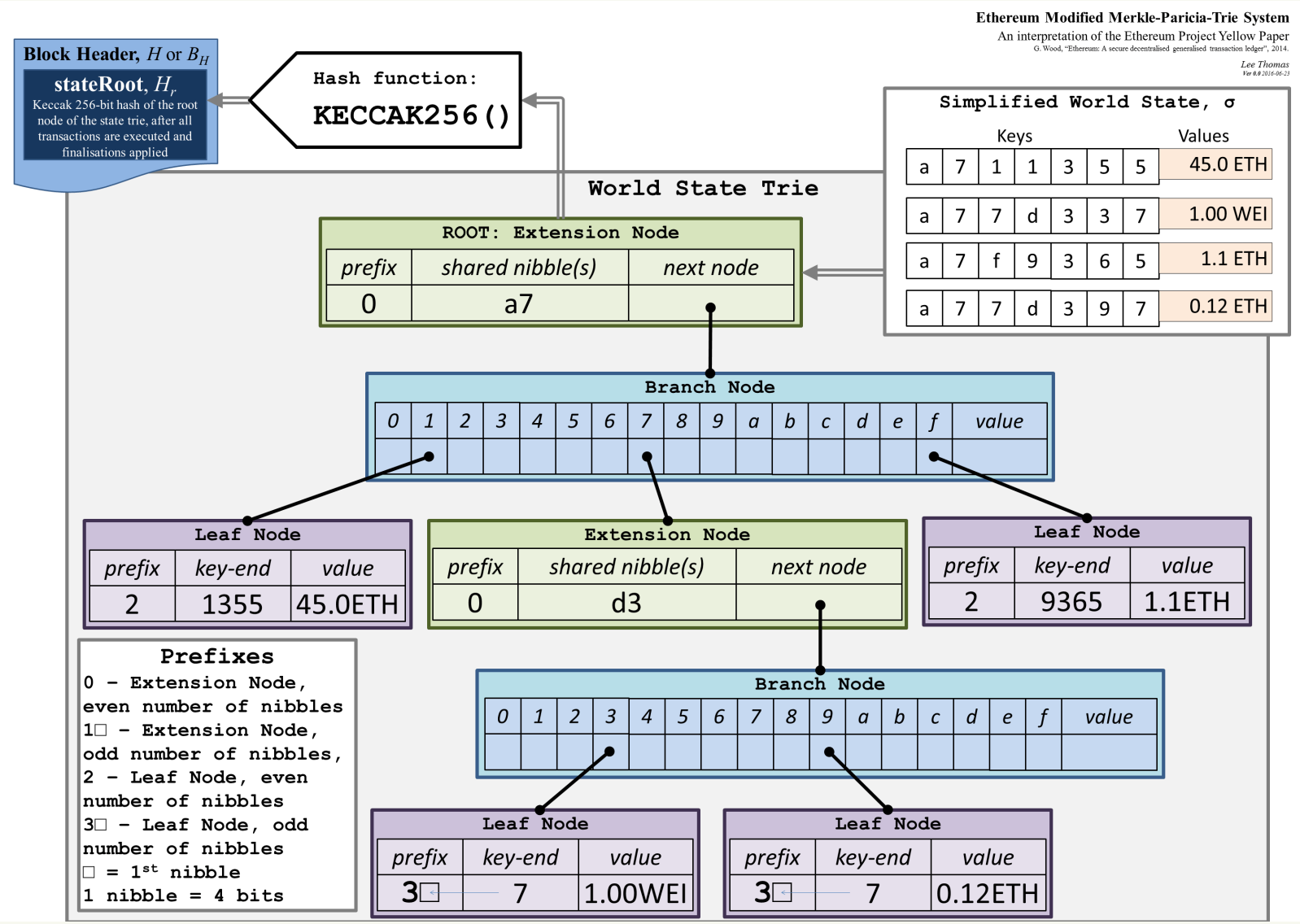
Let's zoom in... world state

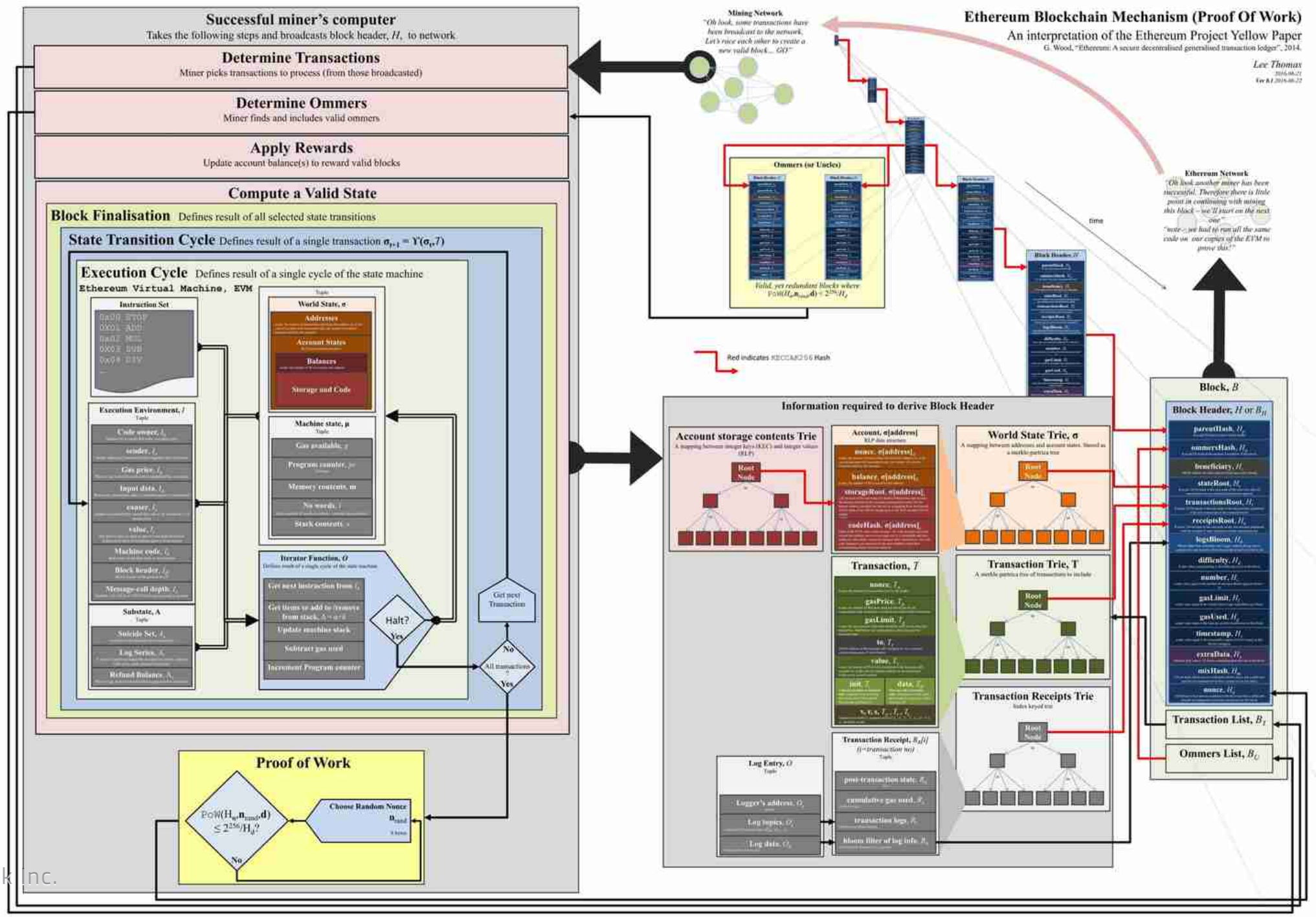


Question: how a node can rollback blocks?


Past
Next

Ethereum Merkle Patricia Tree





Lessons learned

Bitcoin / Ethereum

The good parts

- The epoch-making integration of existing technologies to make digital assets public verifiable
  - p2p, merkle tree, public key encryption, hash, trie, VM, event sourcing, etc.
- Push the use of merkle tree to a new stage
  - esp. MPT in ethereum, made time travel possible
- Bring some important concepts back to public awareness
  - public verifiable
  - event sourcing & CQRS
  - immutability
- Future oriented programming
  - bitcoin to use script for verifying transaction
  - bitcoin hashed the public key as the wallet to anti possible security issues

The bad parts

- The design of Ethereum didn't fit its aspiration
  - The gas system made the use of sophisticated app very costly (e.g. a `sstore` inst cost 5000 ~ 20000 gas). So how can it be a world computer?
  - No support for "on-chain" or "off-chain" filesystem
- Ethereum "smart contract" is a bad name with incorrect implications
- State is still **centralized**, though "replicated" world-wide
  - The limit of a single machine is its ceiling
  - before decentralizing computing, we shall decentralize data
- The **single core single machine** curse
- Protocol design is outdated (esp Ethereum, why not using protobuf?)
- A bit far away from the non-geek user and real-world applications

Forge: Ruby on Rails for Blockchain

ArcBlock Inc.

Why Forge?

or, why bother build your own chain?

- Should we rely on public chains?
  - public chains are either too crowded or too risky for apps
  - Build an app on its own chain should be easy and flexible enough
- Should we build everything from scratch?
  - A wholistic framework that all the batteries are included
  - A powerful CLI that alleviate most day to day work
  - A ready-to-use UI to explore the chain and manage accounts / states
- Shouldn't we focus on real world problems?

Build your own chain...

things you need to consider

Forge Deploy
Forge Patron
Forge dApp workshop
Forge starter app

App Layer (prototype, skeleton, testing, deployment)

Forge, Cosmos SDK core

Transaction Layer (what data to write)

Tendermint
Conflux, NATO

Consensus Layer (who is authorized to write)

Merkle Tree

Merkle Patricia Tree

State Layer (data structure that can prove itself)

LevelDB
RocksDB

Data Layer (Store and load data)

LibP2P
Tendermint
LASP

Network Layer (how data is synced across network)

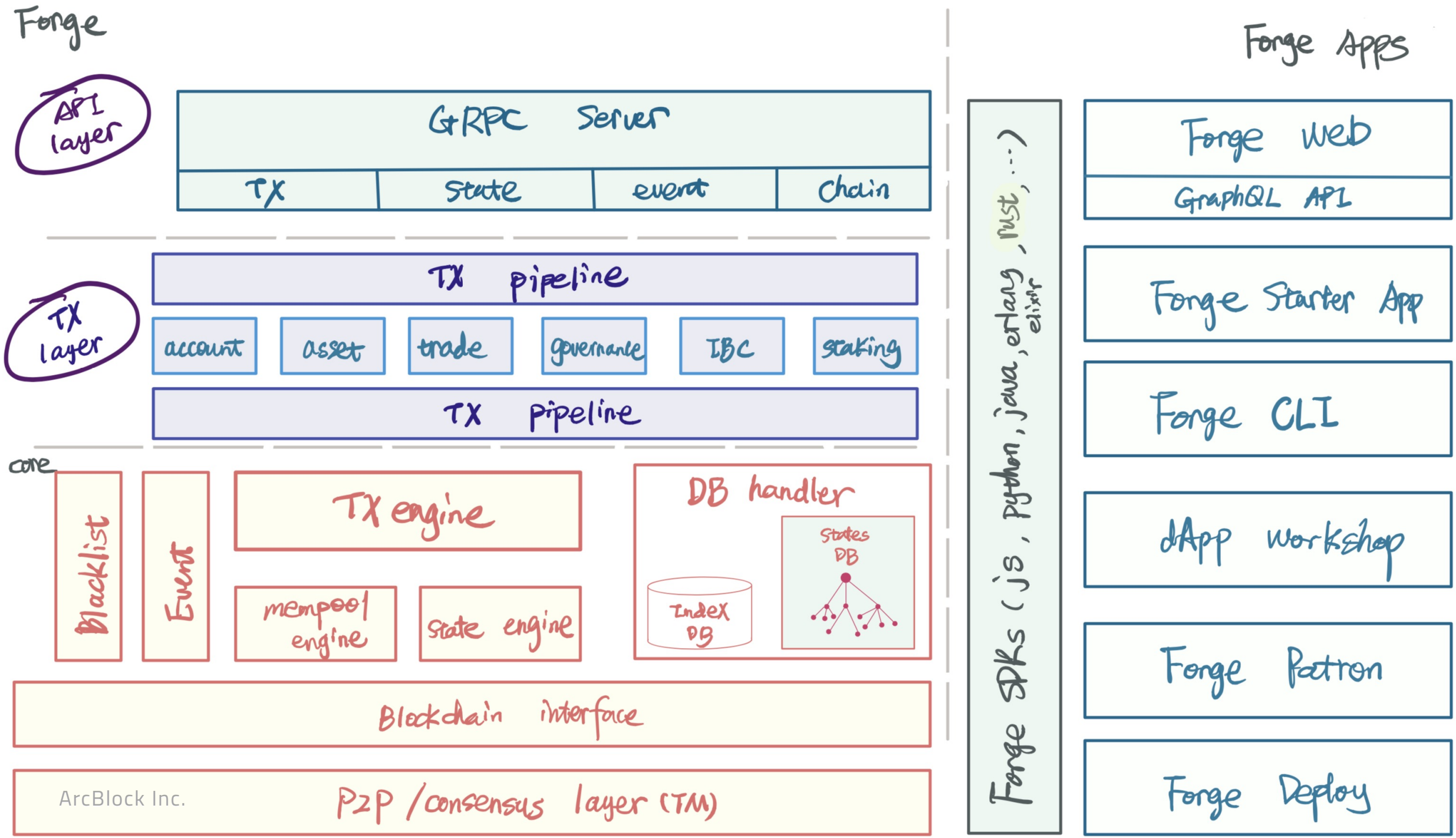
Forge builds all these for you, with extensibility deeply rooted in its design

And of course, the whole pipeline to build a real world app

```
|
build – test – deploy
```

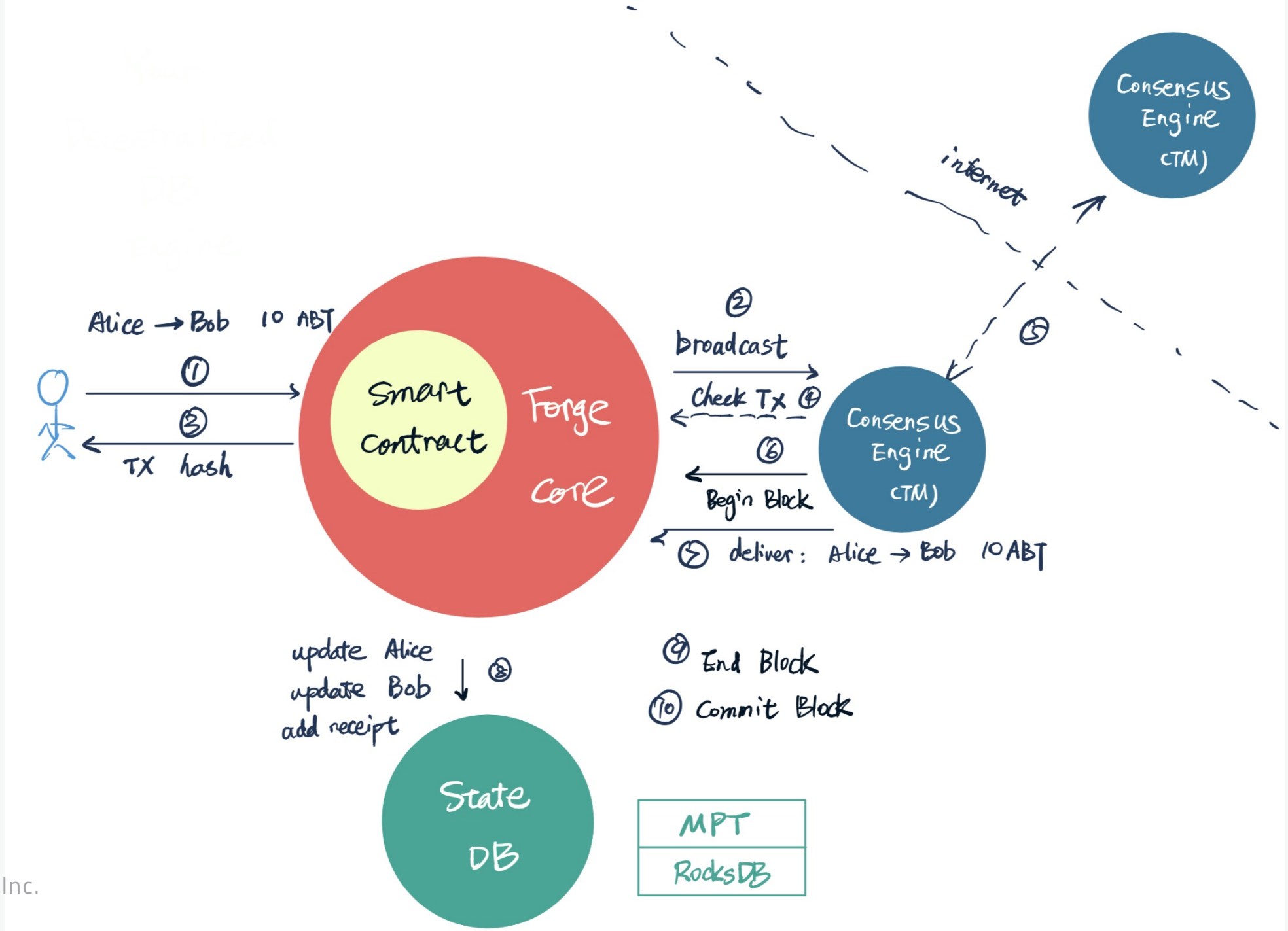
What does it contains?

- Forge core: processing smart contract and consensus / p2p
- Core protocols: various contracts that could be plugged into a chain
- Forge web: block explorer and management console
- SDK: development kits for js / python / java / erlang / elixir / rust (ongoing)
- Forge Starter Apps： starter template for building apps
- Forge CLI： CLI for managing and interacting with the node
- Forge Simulator： massive transaction traffic generator
- dApp Workshop： product prototype for dApps
- Forge Patron： an solution test tool allows you to write yaml DSL for tests
- Forge Deploy： deploy a chain to many nodes across data centers


Forge
API layer
GRPC Server
TX
State
event
Chain
TX layer
TX pipeline
account
asset
trade
governance
IBC
staking
TX pipeline
core
Blacklist
Event
TX engine
mempool engine
State engine
DB handler
States DB
Index DB
Blockchain interface
P2P / consensus layer (TM)
ArcBlock Inc.
Forge SDKs (js, python, java, erlang, rust, ... elixir)
Forge Apps
Forge Web
GraphQL API
Forge Starter App
Forge CLI
dApp Workshop
Forge Patron
Forge Deploy

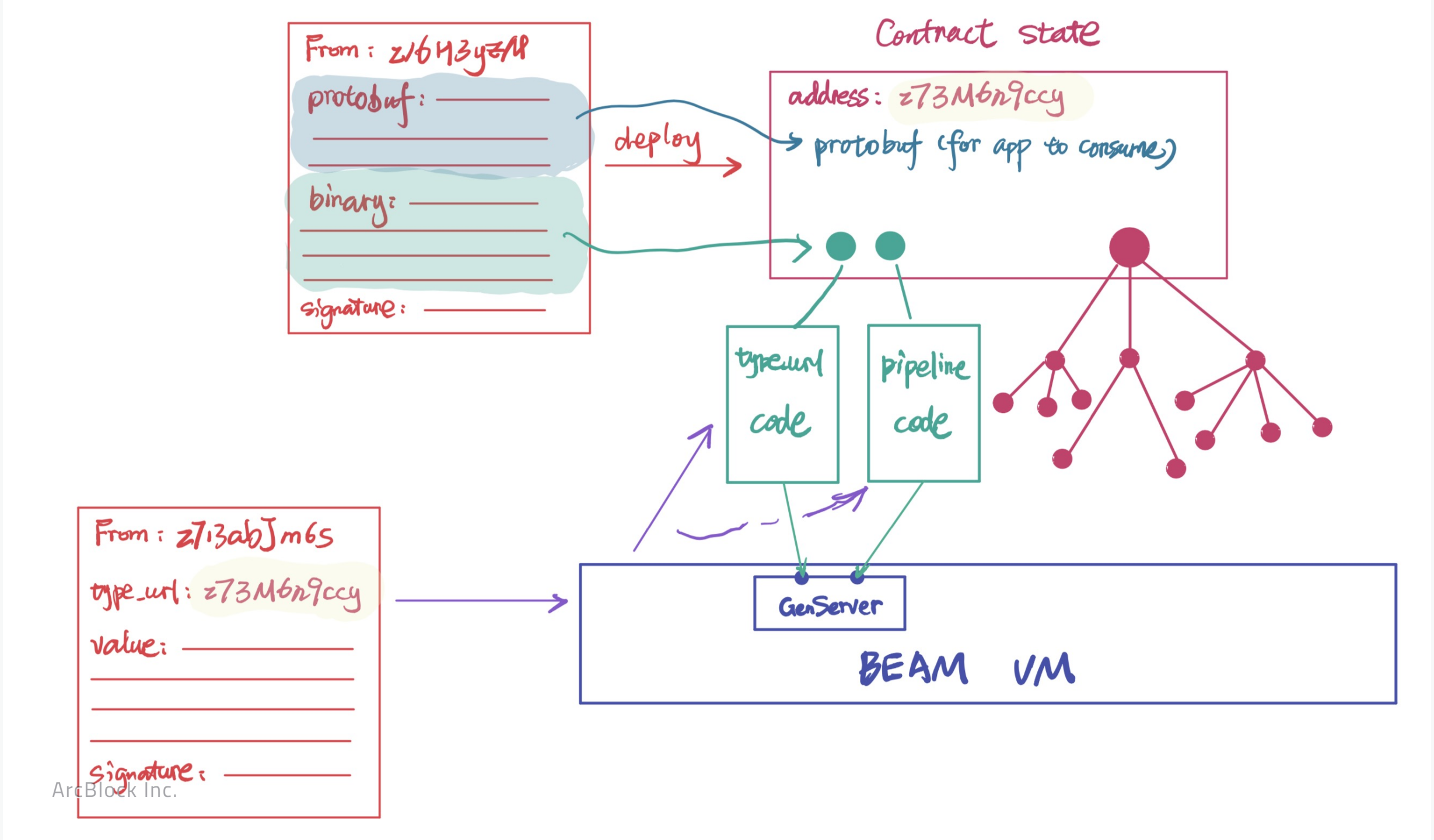
How TX is executed

A high-level view


Consensus Engine (TM)
internet
Alice → Bob 10 ABT
1
2
broadcast
Check Tx
3
TX hash
Smart Contract
Forge Core
Consensus Engine (TM)
5
6
Begin Block
7 deliver: Alice → Bob 10 ABT
update Alice
update Bob
add receipt
8
9 End Block
10 Commit Block
State DB
MPT
RocksDB

How TX is executed

A closer look



How TX is executed

Be an insider

Pre-pipeline: common paths

Think about TCP/IP that underneath the application layer

```yaml
---
check:
  - pipe: decode_tx
  - pipe: verify_tx
  - pipe: verify_blacklist
  - pipe: verify_replay
  - pipe: decode_itx
  - pipe: verify_protocol_state
  - pipe: verify_tx_size
  - pipe: verify_signature
  - pipe: verify_multisig

verify:
  - pipe: extract_state
    from: [:tx, :from]
    to: [:sender_state]
    status: :ok
  - pipe: extract_state
    from: [:tx, :delegator]
    to: [:priv, :delegator_state]
    status: :ok
  - pipe: verify_delegation
    delegation_state: [:priv, :delegation_state]
  - pipe: extract_signer
    to: [:priv, :signers]
    delegators: [:priv, :signer_delegators]
    status: :invalid_tx
  - pipe: verify_delegation
    type: :multisig
    delegation_states: [:priv, :signer_delegation_states]
  - pipe: verify_account_migration
    signers: [:priv, :signers]
  - pipe: verify_sender
```

TX specific pipeline: the core logic

```yaml
---
name: transfer
check:
  - pipe: extract_receiver
    from: [[:itx, :to]]
  - pipe: verify_info
    conditions:
      - expr: "info.itx.to !== \"\" and (info.itx.value !== nil or info.itx.assets !== [])"
        error: :insufficient_data
  - pipe: verify_itx_size
    value: [[:itx, :assets]]

verify:
  - pipe: verify_balance
    state: :sender_state
    value: [[:itx, :value]]
  - pipe: verify_receiver
  - pipe: extract_state
    from: :receiver
    to: :receiver_state
    status: :invalid_receiver_state
  - pipe: anti_land_attack
  - pipe: extract_state
    from: [:itx, :assets]
    to: [:priv, :assets]
    status: :invalid_asset
  - pipe: verify_transferrable
    assets: [:priv, :assets]
  - pipe: verify_owner
    assets: [:priv, :assets]
    state: :sender_state

update:
```

Post-pipeline: after TX is executed

```yaml
---
check: []

verify: []

update:
  - pipe: reset_halted
  - pipe: update_gas_balance
  - pipe: update_delegation_state
  - pipe: update_receipt_db
  - pipe: update_index_db
  - pipe: broadcast
```

Testing the chain and app

Why Forge Patron?

- make tests available to multi-node multi-chain environment
  - test for deploying new protocols
  - test for node upgrade
  - test for cross chain protocols
- make tests easily accessible to everybody
  - that's why tests are in yaml and descri
- make tests composable

The requirement on Patron

- able to define flexible topologies
  - user can easily copy & tune existing topology
- able to start / stop / cleanup on a specific topologies
  - which means in local environment the multiple running instances of forge must not impact each other
  - which means the keys / forge configuration shall be generated so validators can connect to each other
- able to run tests
  - parallel is important (for performance purpose)
  - tests shall be able to depend on other tests (so we can build complex solution-based testing)
- tests shall be defined as easy as possible without much coding work
  - better to be able to copy & tune existing tests to generate new ones

Topology

what is a topology?

- a topology defines how to mimic a real world deployment in a single node
- human-readable, easy to modify and reason about
- contains all the parameters to generate forge config
- isolate chains
- extensible

What a topology looks like?

```
type: topology
chains:
  - chain_id: asset_chain
    validators:
      - '[address: "127.0.0.1", p2p: 10000, proxy: 10010, grpc: 10020, tm_rpc: 10030, tm_grpc: 1
      - '[address: "127.0.0.1", p2p: 10001, proxy: 10011, grpc: 10021, tm_rpc: 10031, tm_grpc: 1
      - '[address: "127.0.0.1", p2p: 10002, proxy: 10012, grpc: 10022, tm_rpc: 10032, tm_grpc: 1
    genesis_config: forge.toml
    moderator:
      sk: tk3pcIjpDRzeUGutXL7mjf52jNfA_kztlQnIgYnBStiY2X5pZlsLGwfiMgFA6a8qLhCEgMjGmEBjcFROew9TXw
      pk: mNl-aWZbCxsH4jIBQOmvKi4QhIDIxphAY3BUTnsPU18
      address: z1TpjUv5ZVVpY854GVk9W9Zfnb4HKqQzRSg
    version: 0.29.1
    forge_bin: '~/.forge_cli/release/forge/<%= version %>/bin/forge'
    forge_web_bin: '~/.forge_cli/release/forge_web/<%= version %>/bin/forge_web'
    tm_bin: '~/.forge_cli/release/forge/<%= version %>/lib/consensus-<%= version %>/priv/tenderm
    forge_root: /tmp/asset_chain
```

What is a test flow?

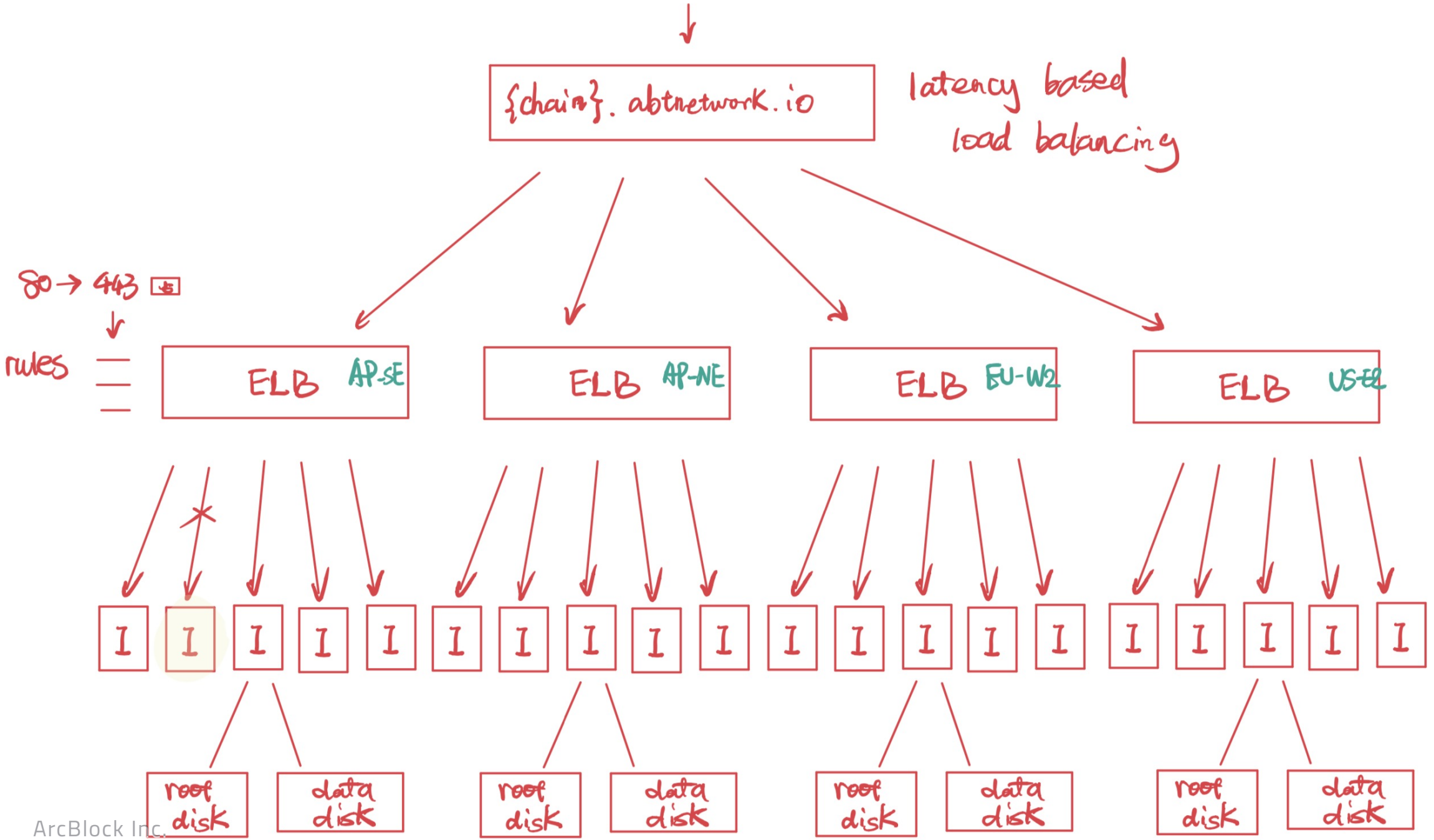
- flow defines the boundary of test cases
  - the boundary of unit tests is the test itself, so we can't have dependency and its stateless
  - the boundary of a solution test is the solution itself, we must have intermediate test states
- Execution model
  - run-to-completion?
  - concurrent?
- isolation (is alice in flow A the same alice in flow B?)
  - what information / state are shared?
  - what information / state are isolated?
- A flow is a set of ordered tests that runs sequentially until finished or terminated due to errors

What a flow looks like?

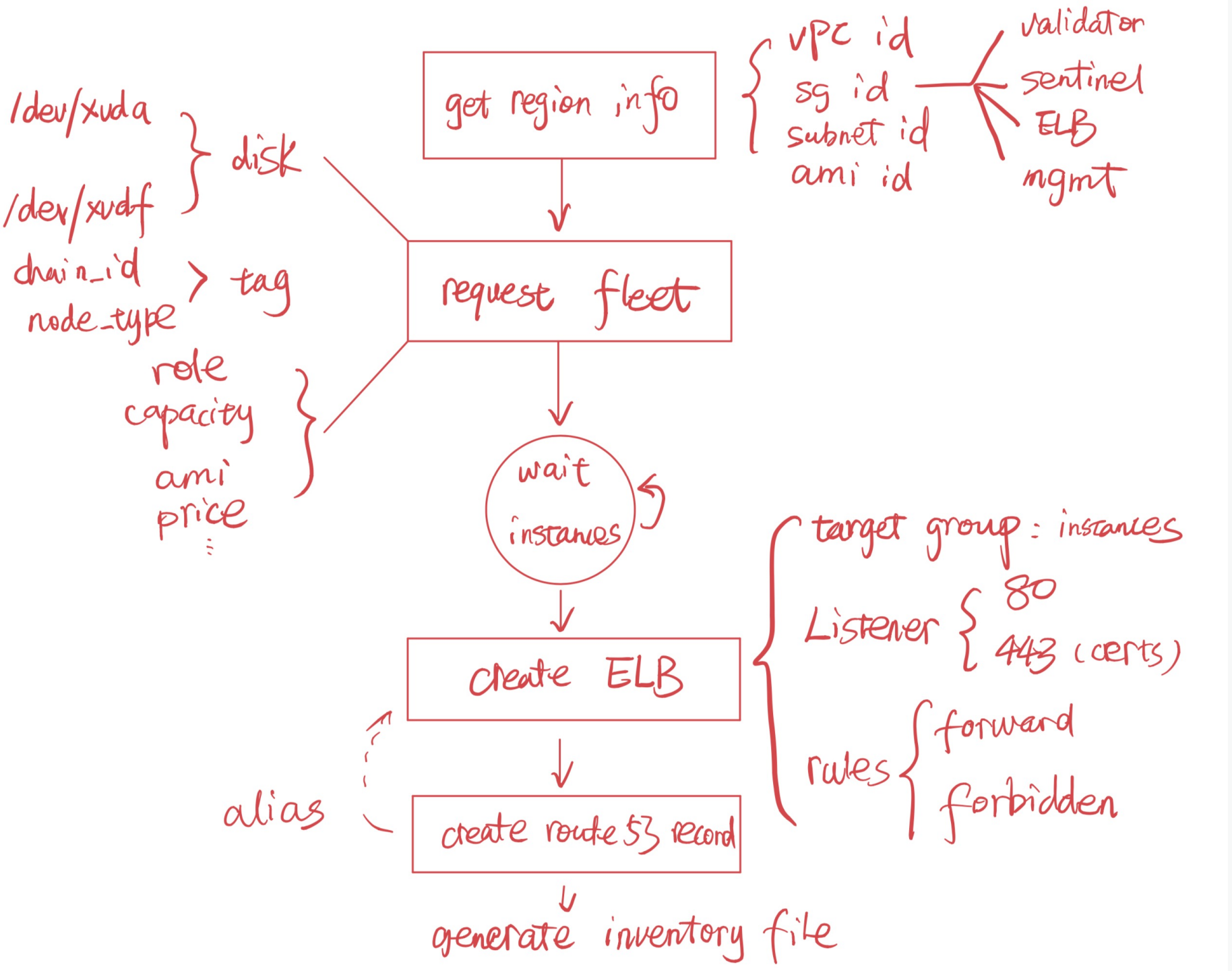
```
- name: declare users
  batch_declare:
    accounts: [alice, bella, charlie, dave, ethan, frank, gore]
  conn: ['asset_chain', 'app_chain']

# accessible as assets["alice_a1"]
- name: create asset for alice
  create_asset:
    sender_wallet: alice
    moniker: "King's treasure"
    data:
      type_url: 'fg:x:json'
      value: '{"title":"hello world!", "content": "a valuable asset"}'
  conn: ['asset_chain'] # if omitted, this would fall back to default
  state_name: alice_a1
  asserts:
    - tx_info.code = ok
    - ^alice_a1.owner = ^alice.address

- name: alice send tokens to bella
  transfer:
    sender_wallet: alice
    to: bella
    value: 10
```

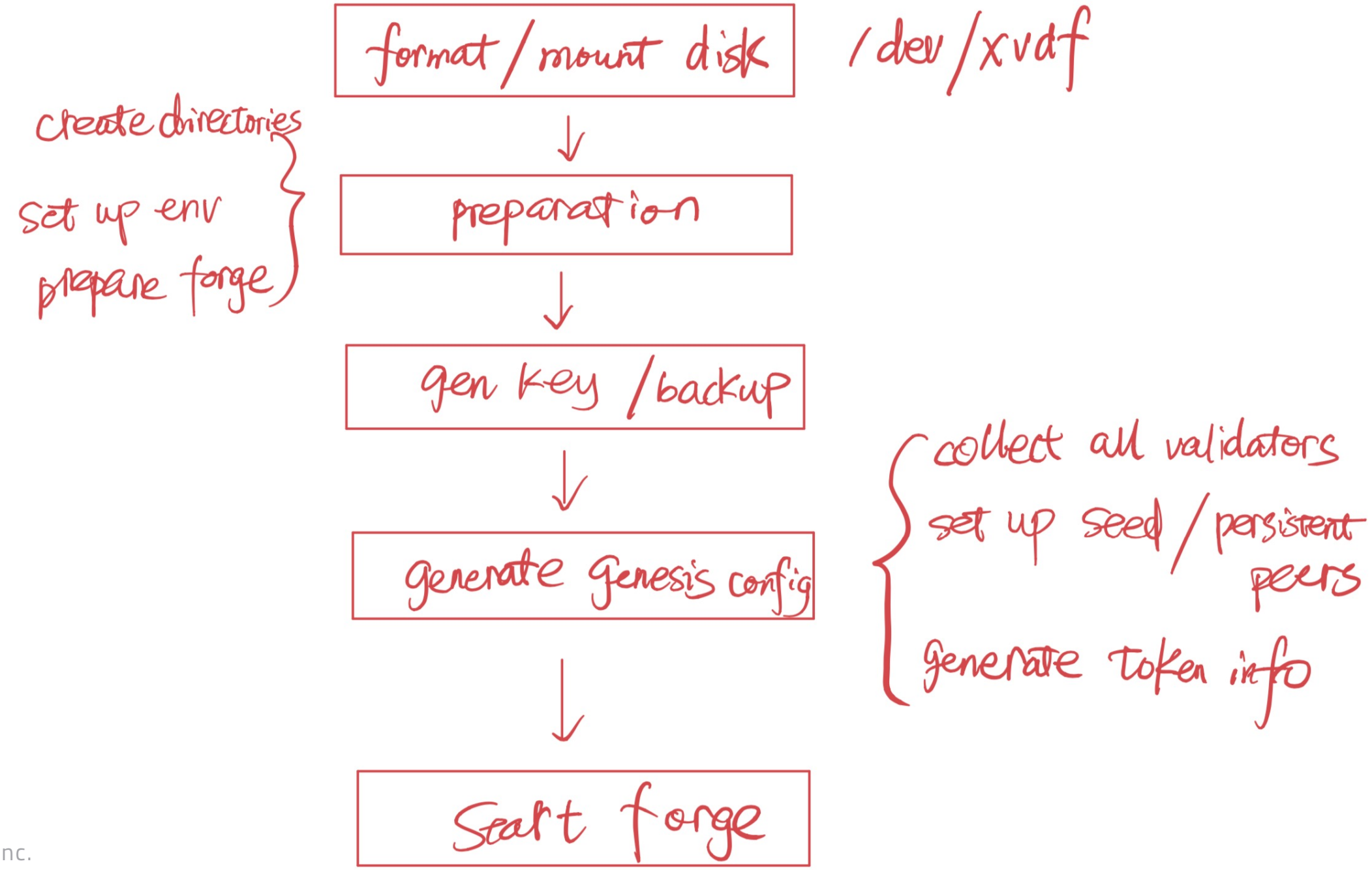
Deploy the chain

The picture in your head



How to achieve it?

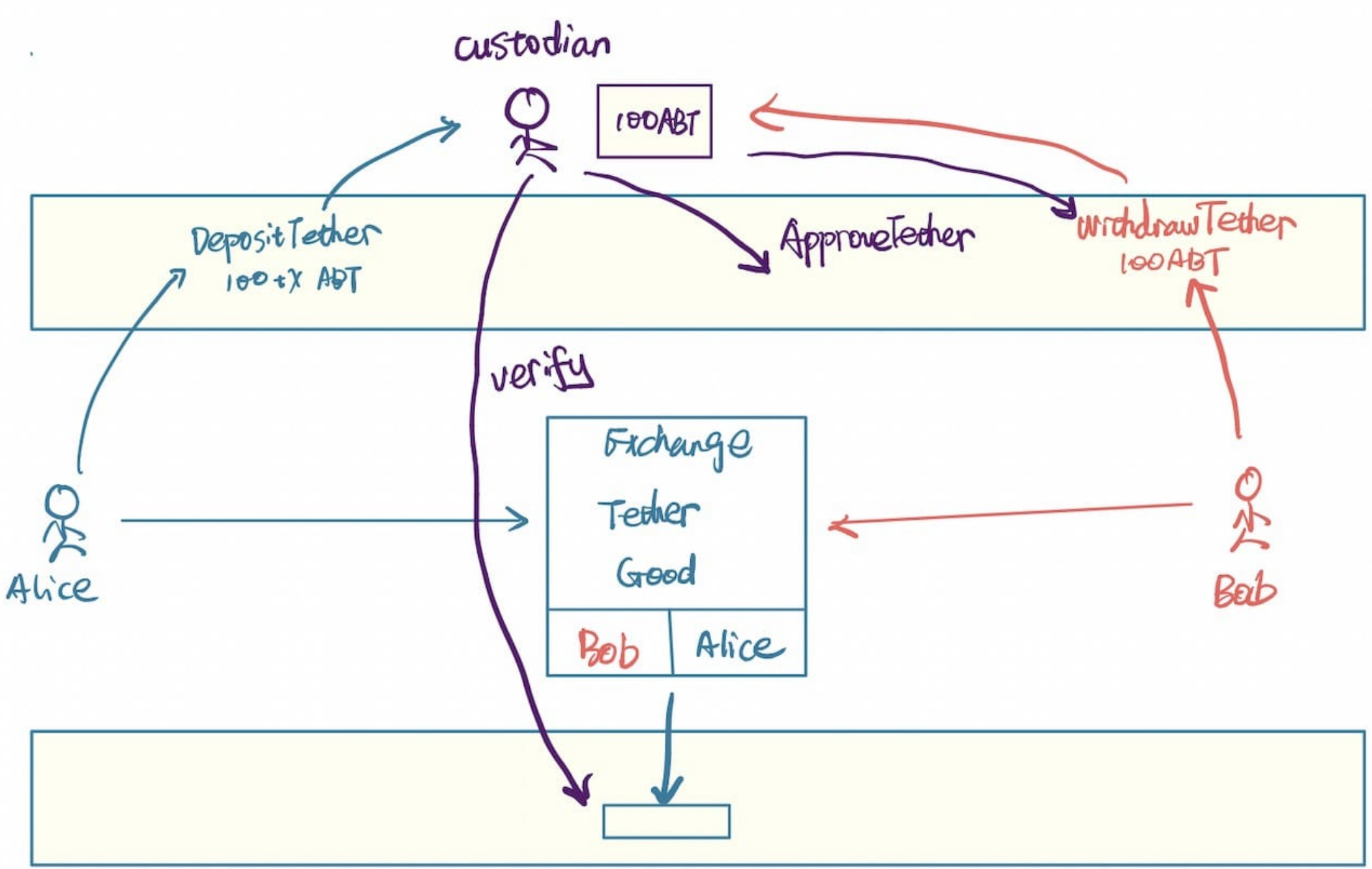



format / mount disk
/dev/xvdf
create directories
set up env
prepare forge
preparation
gen key / backup
generate Genesis config
collect all validators
set up seed / persistent peers
generate token info
Start forge

Inter-Chain communication

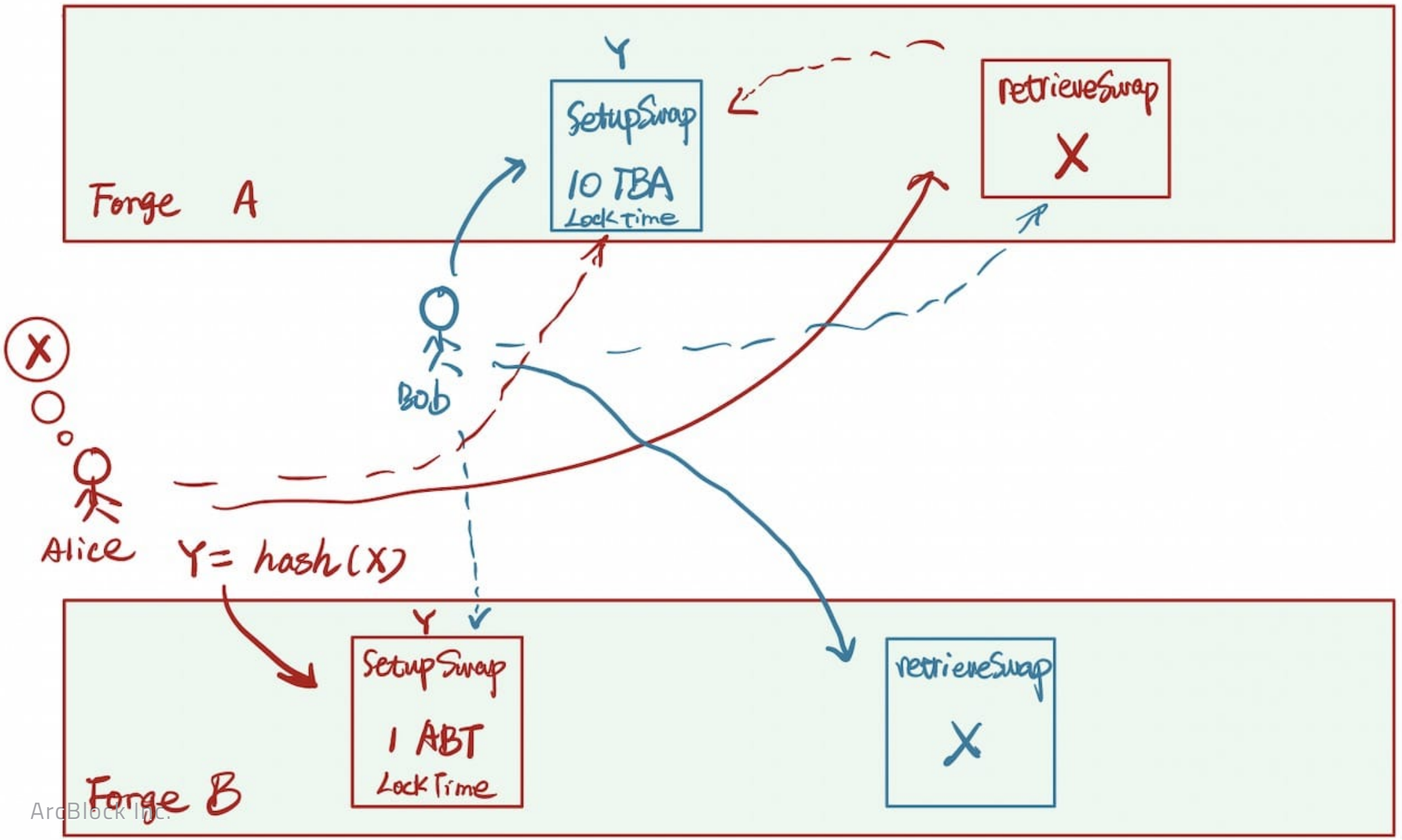
Solution 1: decentralized custodian

- Concepts:
  - custodian: an entity that locks user's tokens for cross-chain communication temporarily, and then release the tokens after the transaction is done
  - asset chain: the chain that locks the tokens
  - app chain: the chain that user wants to acquire assets by her tokens in asset chain
- Flow:
  - Alice: send `DepositTether` tx on asset chain to the custodian
  - Alice: send `Exchange` tx on app chain to buy asset from Bob
  - Bob: Take the signature of `DepositTether` from Alice to withdraw locked tokens from custodian
  - Custodian: verifies the validity of `Exchange` tx
  - Custodian: send `ApproveTether` to unlock tokens to Bob


custodian
100ABT
DepositTether
100+X ABT
ApproveTether
WithdrawTether
100ABT
verify
Exchange
Tether
Good
Bob
Alice
Alice
Bob

Atomic Swap

- Concepts:
  - use specific algorithm to make sure asset exchange is atomic without the help of custodian
- Flow: one key to unlock two locks
  - Alice: setup_swap in Chain A
  - Bob: setup_swap in Chain B
  - Alice: retrieve_swap in Chain B
  - Bob: retrieve_swap in Chain A



Other interesting questions

- How to build apps? (SDKs and starter projects)
- How apps could be discovered? ANS
- How could user interact with the apps? (scan QR code)
- How could my privacy be protected? (forge/ABT wallet has builtin DID support)

Resources

- Bitcoin paper
- Ethereum white paper
- Forge Docs
- arcblock.io
- ABT wallet
- ABT network


Thanks!
University of Victoria
IEEE
Advancing Technology for Humanity
ArcBlock