ArcBlock

Build decentralized, public-verifiable DB

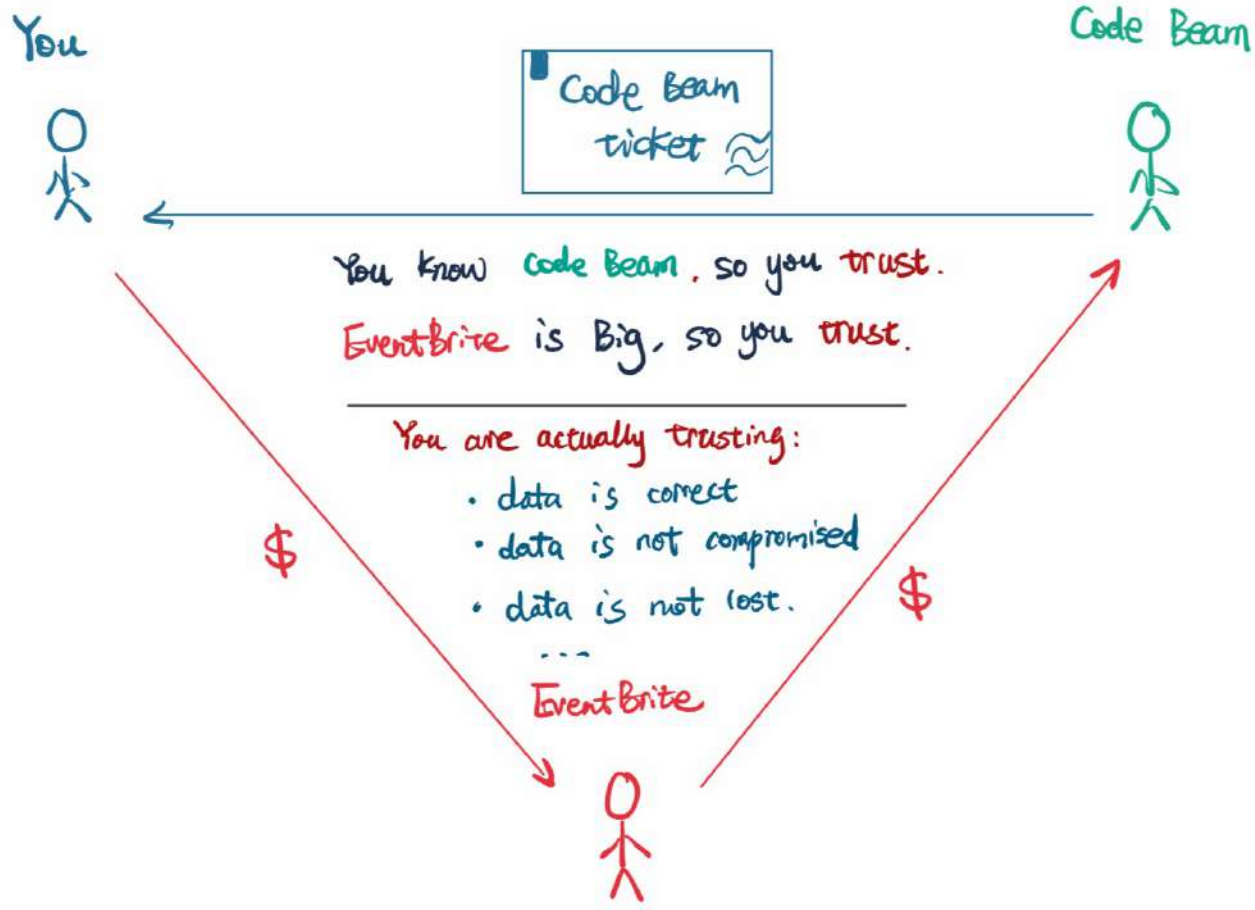
with ex_abci & tendermint

BROUGHT TO YOU BY

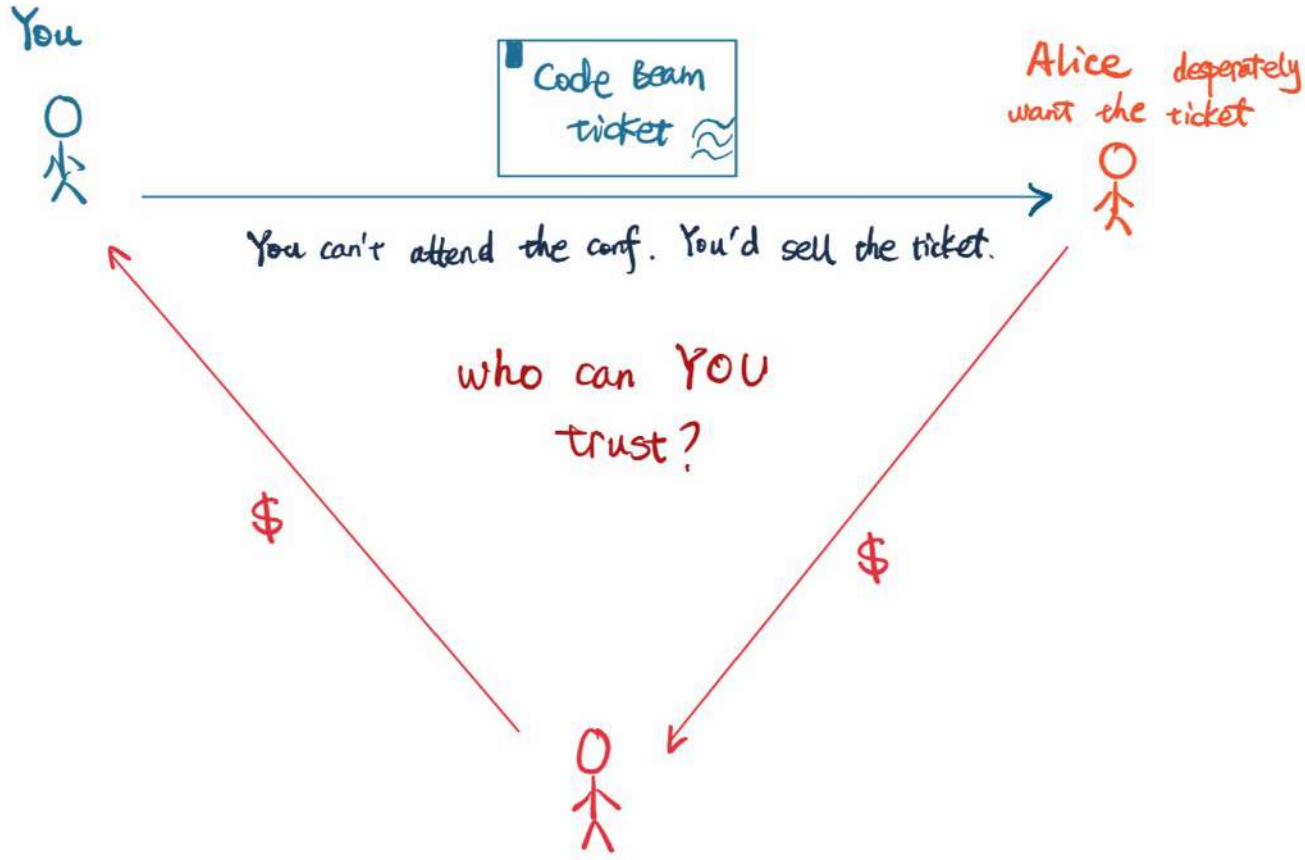
TIAN CHEN

Why decentralization & public-verifiable matters?

Take CODE BEAM ticket as an example



What if I want to resell it?



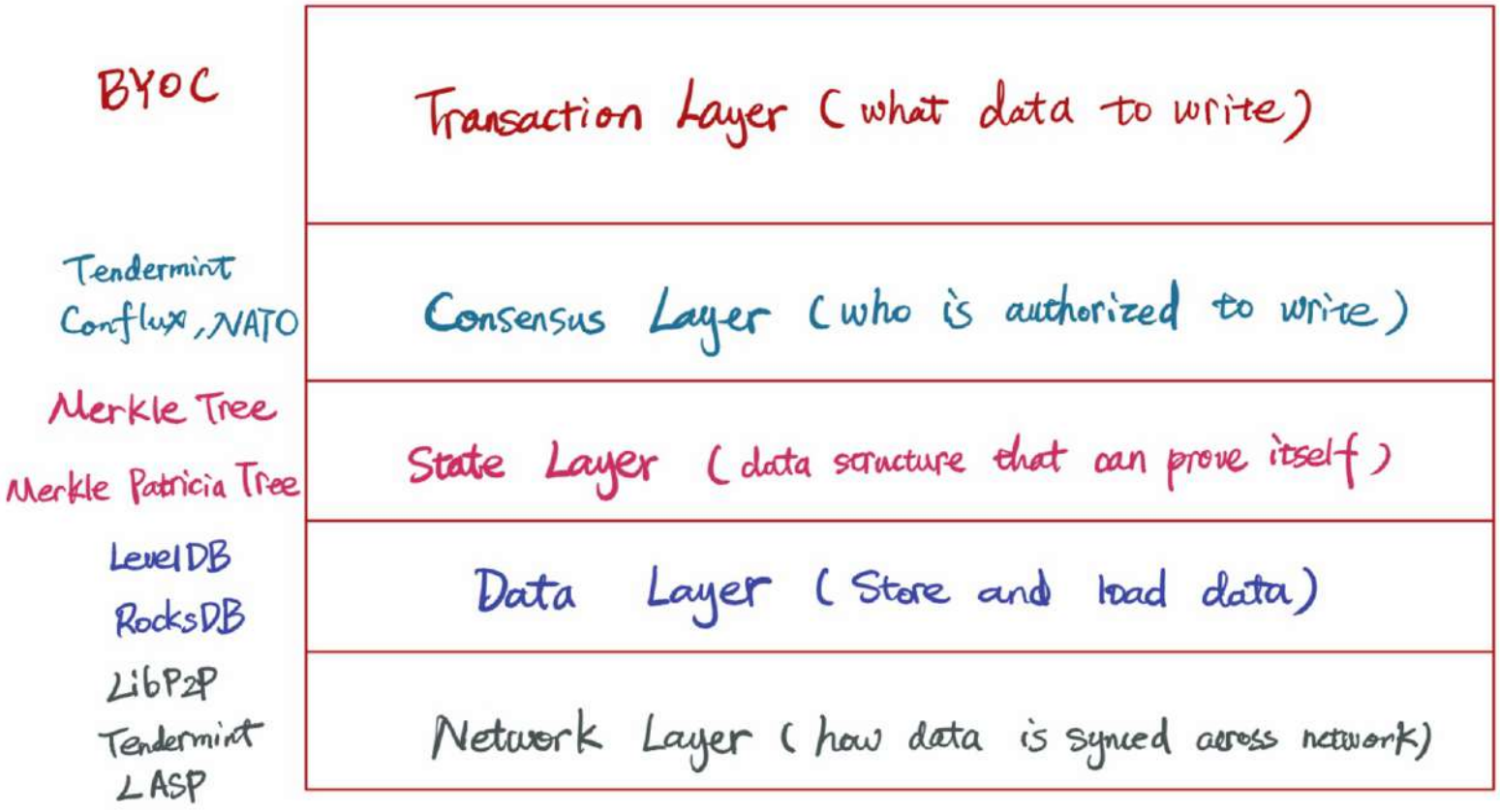
The problem

- Who owns the data? Eventbrite
- Who explains the data? Eventbrite
- How to verify the data? Eventbrite website / API
- Who owns the ticket? Conceptually, **we**

What if the ticket data is public verifiable?

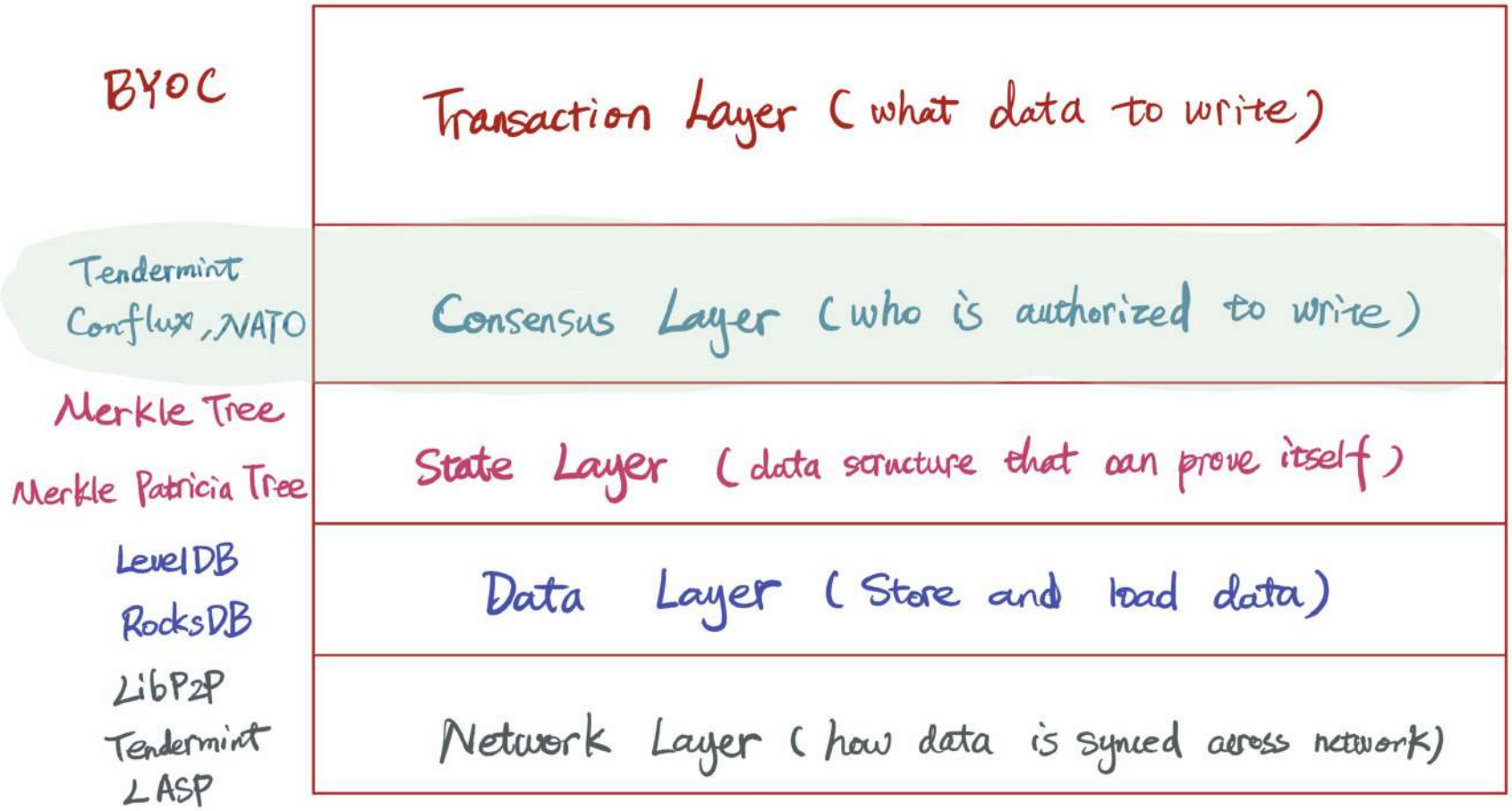
anyone can build service on top of it and everyone is confident to use the service

But how?


BYOC
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)

Some concepts before we go further

Consensus related concepts



What is consensus (and block / transaction)?

- consensus: **who** can write **what**, in **which order**, into the database
  - Distributed: leader election / two phase commit for Crash Fault Tolerance
  - Decentralized: consensus algorithms (NATO / pBFT / etc.) for Byzantine Fault Tolerance
- block: a **container** to decide,
  - how time is synchronized
    - Distributed: not needed. Master / leader provides the global timer
    - Decentralized: block acts as a tick to move the whole network forward.
  - how transactions are ordered
    - Distributed: not needed (master decides the order)
    - Decentralized: block acts as a container of transactions to make sure everyone process the data in the same order
- transaction: a **FACT** that potentially triggers state transit (how state transit)
  - Distributed: whatever client told the DB server is treated as a fact
  - Decentralized: data that signed by one or more entities is fact (self provable)

TRUST from authority to MATH/ALGO

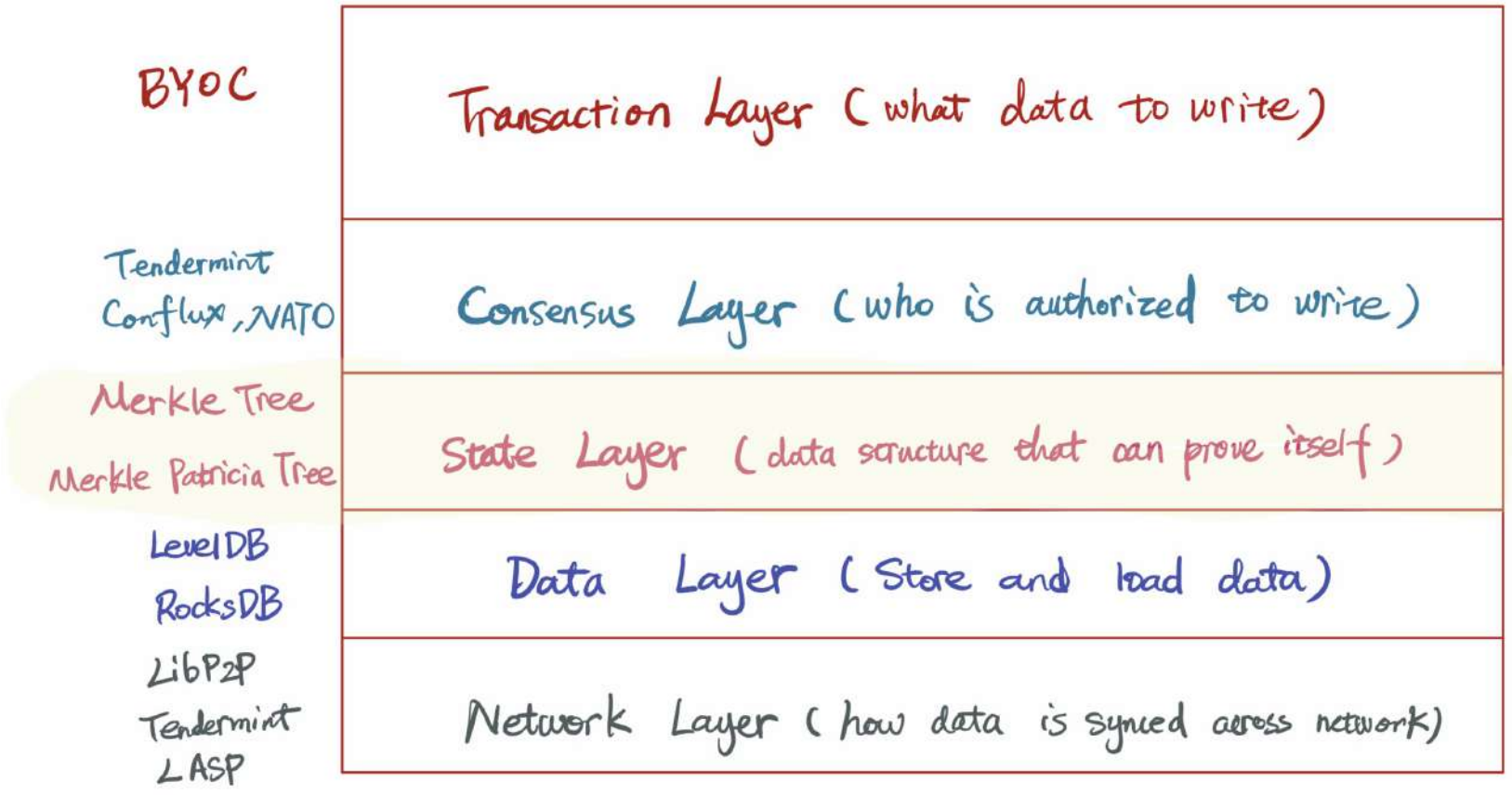
What is tendermint?

- an open source project writes on golang to provide a PBFT based consensus layer
- has a nice abstraction separating consensus/p2p from application
- application can listen to these events:
  - begin block
  - deliver tx
  - end block
  - commit block

What is ex_abci?

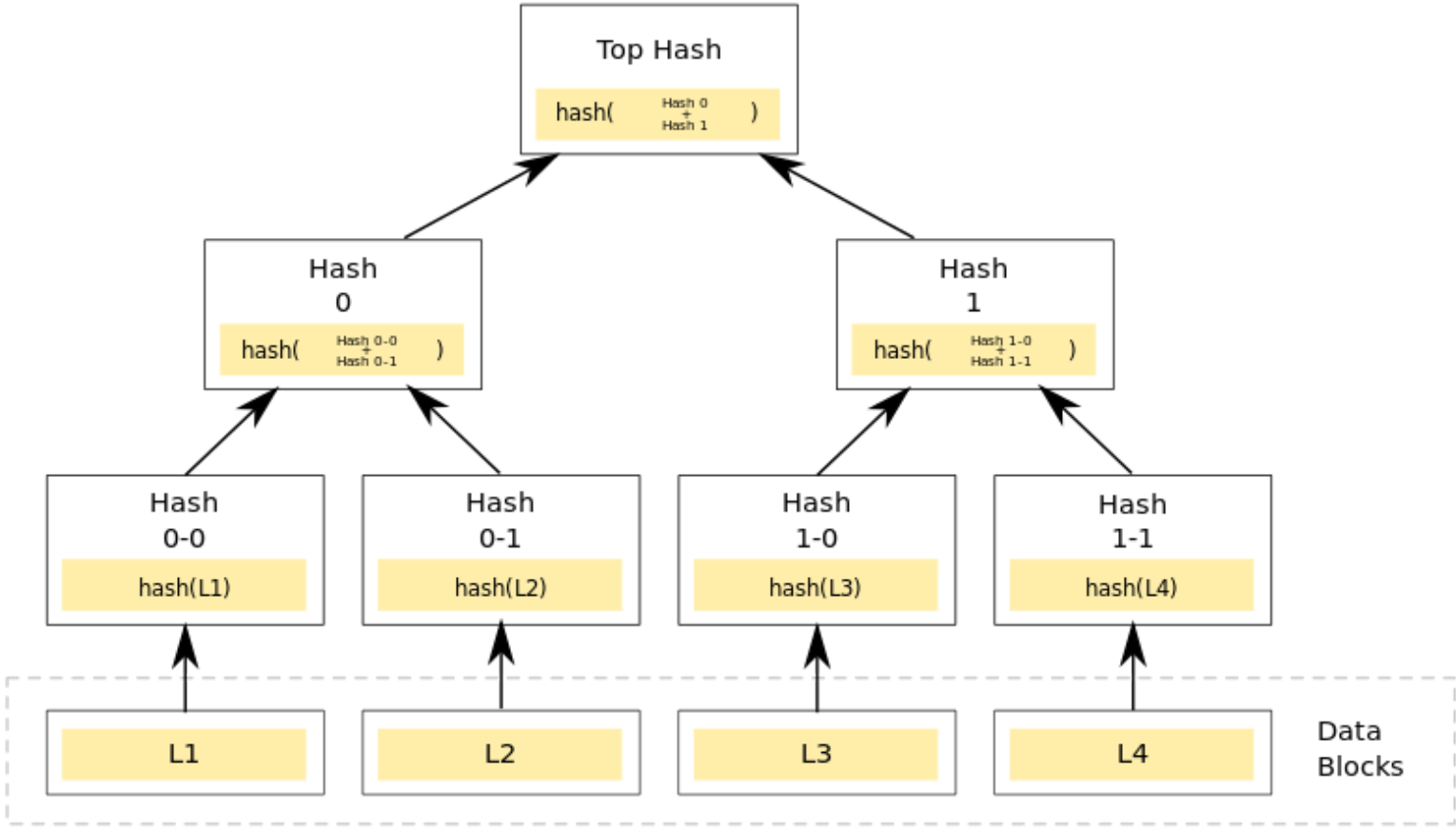
- an elixir implementation of tendermint ABCI protocol (by ArcBlock)
  - https://github.com/ArcBlock/ex-abci-proto
- uses ranch listener to accept connections
- inspired by abci_server and js-abci
- application just need to process the decoded events, e.g.
  - handle_begine_block
  - handle_deliver_tx
  - handle_end_block
  - handle_commit_block

Verifiable data related concepts

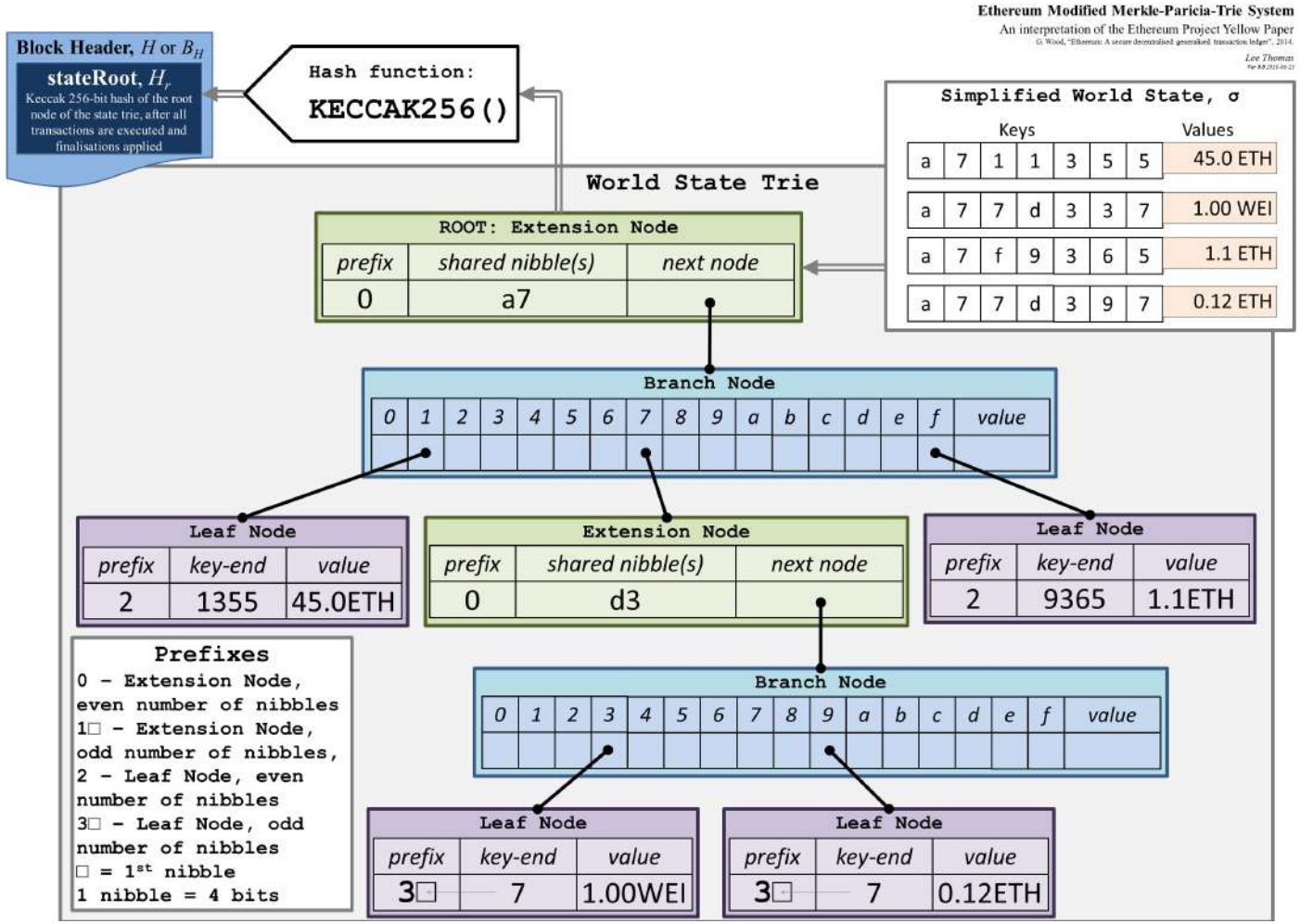


The fact is verifiable, what about the state?

What is merkle tree?



What is merkle patricia tree (MPT)?



MPT vs Erlang Map

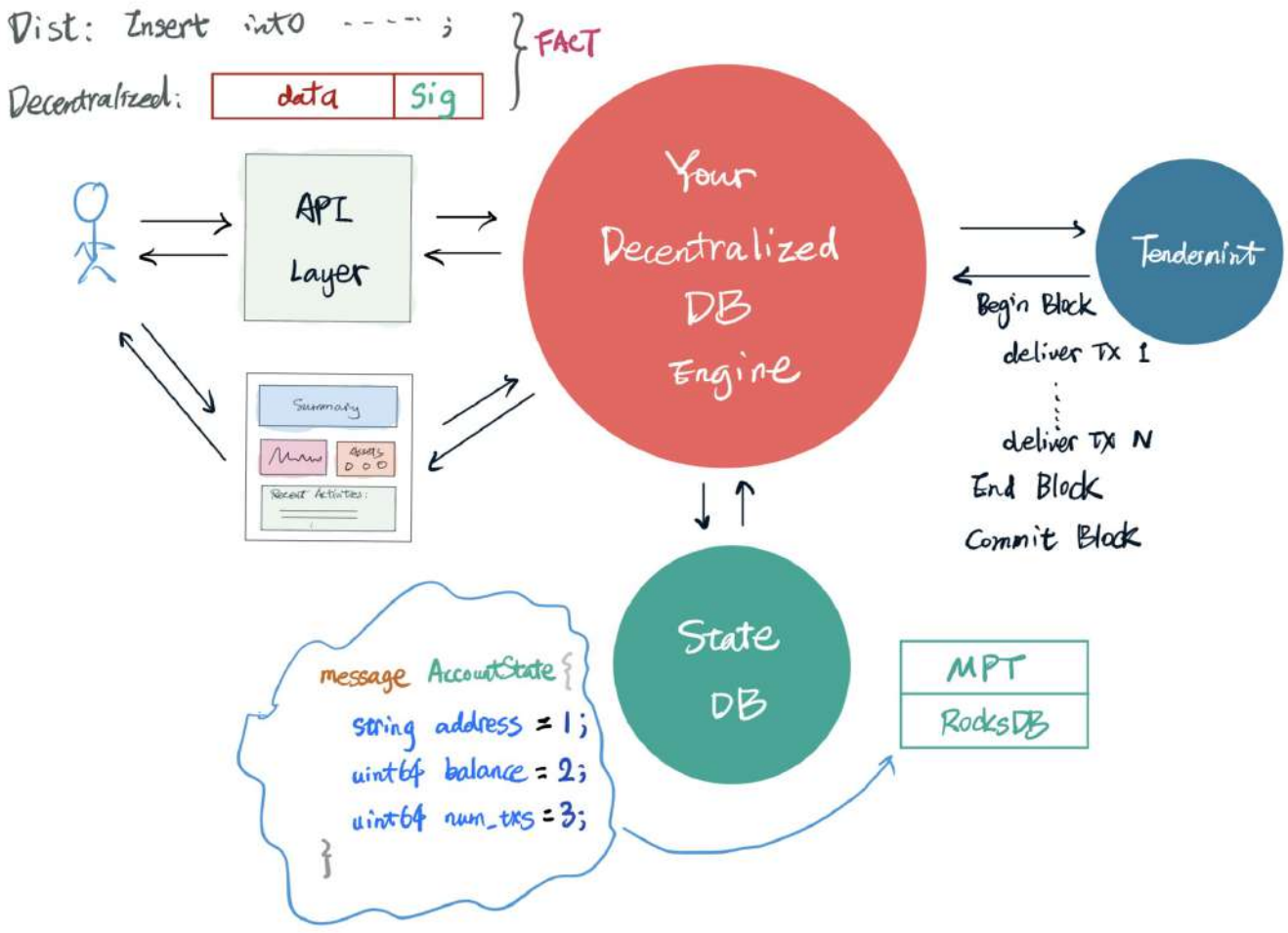

```
# `m` is actually a persistent data structure that internally a tree is used to store it
m = Map.put(%{"data" => "unchanged data"}, "greeting", "hello")

# `m1` is new tree that inherit most of the parts of the old tree `m`
m1 = Map.put(m, "greeting", "nihao")

# normally if no one reference `m` any more it is garbage collected
```

- MPT made sure all the references along the lifecycle will be stored
- MPT made sure the data

  `%{"data" => "unchanged data", "greeting" => "nihao"}` would only:

  - converge to the same `m1` through crypto-graphical functions,
  - and through `m1` one can surely get “nihao” with key “greeting”
  - this is very important for decentralized DB - we can’t rely on Map since it is a local reference

Now let's build a decentralized, public-verifiable DB



Defining transaction and state


```
message Transaction {
  bytes from = 1;
  bytes to = 2;
  uint64 nonce = 3;
  uint64 total = 4;
  bytes pub_key = 5;
  bytes signature = 6;
}

message AccountState {
  string address = 1;
  uint64 balance = 2;
  uint64 nonce = 3;
  uint64 num_txs = 4;
}
```

Note: we use protobuf here because it is upgradable, small footprint, designed for building protocols across the wire.

Use MPT to store verifiable data

- use merkle_patricia_tree lib
  - leveldb version: exthereum/merkle_patricia_tree
  - rocksdb version: tyrchen/merkle_patricia_tree

```
iex(4)> trie = MerklePatriciaTree.Trie.update(trie, "hello", "world")
%MerklePatriciaTree.Trie{
  db: {MerklePatriciaTree.DB.RocksDB,
   {#Reference<0.2842935440.3690070038.178302>,
    #Reference<0.2842935440.3690070038.178303>}},
  root_hash: <<10, 145, 86, 89, 184, 143, 128, 191, 162, 0, 87, 11, 210, 118,
    122, 106, 184, 203, 10, 78, 68, 253, 36, 12, 197, 237, 122, 39, 114, 140,
    69, 49>>
}
iex(5)> MerklePatriciaTree.Trie.get(trie, "hello")
"world"
```

History of data (time travel)


```
iex(6)> trie1 = MerklePatriciaTree.Trie.update(trie, "hello", "world1")
%MerklePatriciaTree.Trie{
  db: {MerklePatriciaTree.DB.RocksDB,
   {#Reference<0.2842935440.3690070038.178302>,
    #Reference<0.2842935440.3690070038.178303>}},
  root_hash: <<168, 6, 249, 16, 118, 215, 5, 41, 16, 207, 47, 57, 237, 186, 153,
    106, 117, 176, 236, 26, 151, 226, 16, 68, 222, 47, 230, 102, 131, 237, 82,
    19>>
}
iex(7)> MerklePatriciaTree.Trie.get(trie1, "hello")
"world1"
iex(8)> root_hash = <<10, 145, 86, 89, 184, 143, 128, 191, 162, 0, 87, 11, 210, 118,
...(8)>     122, 106, 184, 203, 10, 78, 68, 253, 36, 12, 197, 237, 122, 39, 114, 140,
...(8)>     69, 49>>
<<10, 145, 86, 89, 184, 143, 128, 191, 162, 0, 87, 11, 210, 118, 122, 106, 184,
  203, 10, 78, 68, 253, 36, 12, 197, 237, 122, 39, 114, 140, 69, 49>>
iex(9)> MerklePatriciaTree.Trie.get(%{trie1 | root_hash: root_hash}, "hello")
"world"
```

Update state upon deliver tx


```elixir
def handle_call({:handle_deliver_tx, request}, _from, %{trie: trie} = state) do
    %AbciVendor.RequestDeliverTx{tx: data} = request
    tx = Transaction.decode(data)

    Logger.debug(fn -> "Deliver tx: #{inspect(tx)}" end)

    response = struct(ExAbci.ResponseDeliverTx, verify(tx, trie, data))

    trie =
      case response.code == 0 do
        true -> update_state(tx, trie)
        _ -> trie
      end

    {:reply, response, %{state | trie: trie}}
  end
```

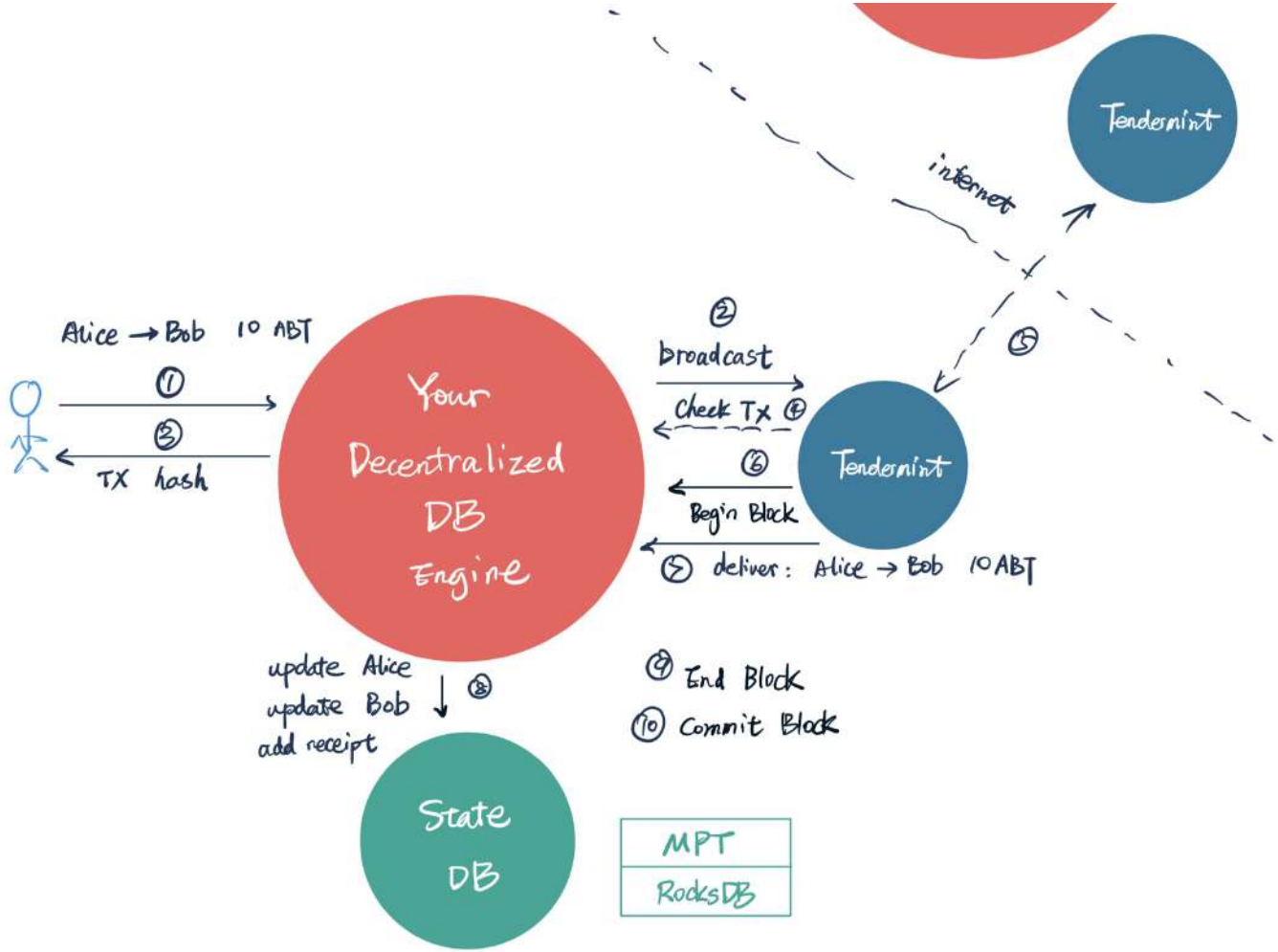
Update state upon deliver tx


```elixir
defp update_state(%Transaction{from: from, to: to, nonce: n1, total: total}, trie) do
  acc1 = Account.get(trie, from)

  trie =
    Account.put(trie, from, %AccountState{
      acc1
      | nonce: n1 + 1,
        balance: acc1.balance - total,
        num_txs: acc1.num_txs + 1
    })

  acc2 =
    case Account.get(trie, to) do
      nil -> %AccountState{nonce: 0, balance: total, num_txs: 1}
      v -> %AccountState{v | balance: v.balance + total, num_txs: v.num_txs + 1}
    end

  Account.put(trie, to, acc2)
end
```

Broadcast a transaction


```
def send(tx) do
  data = Transaction.encode(tx)
  url = "http://localhost:26657/broadcast_tx_commit?tx=\"#{Base.encode64(data)}\""
  HTTPoison.get(url)
end
```

The whole flow revisited



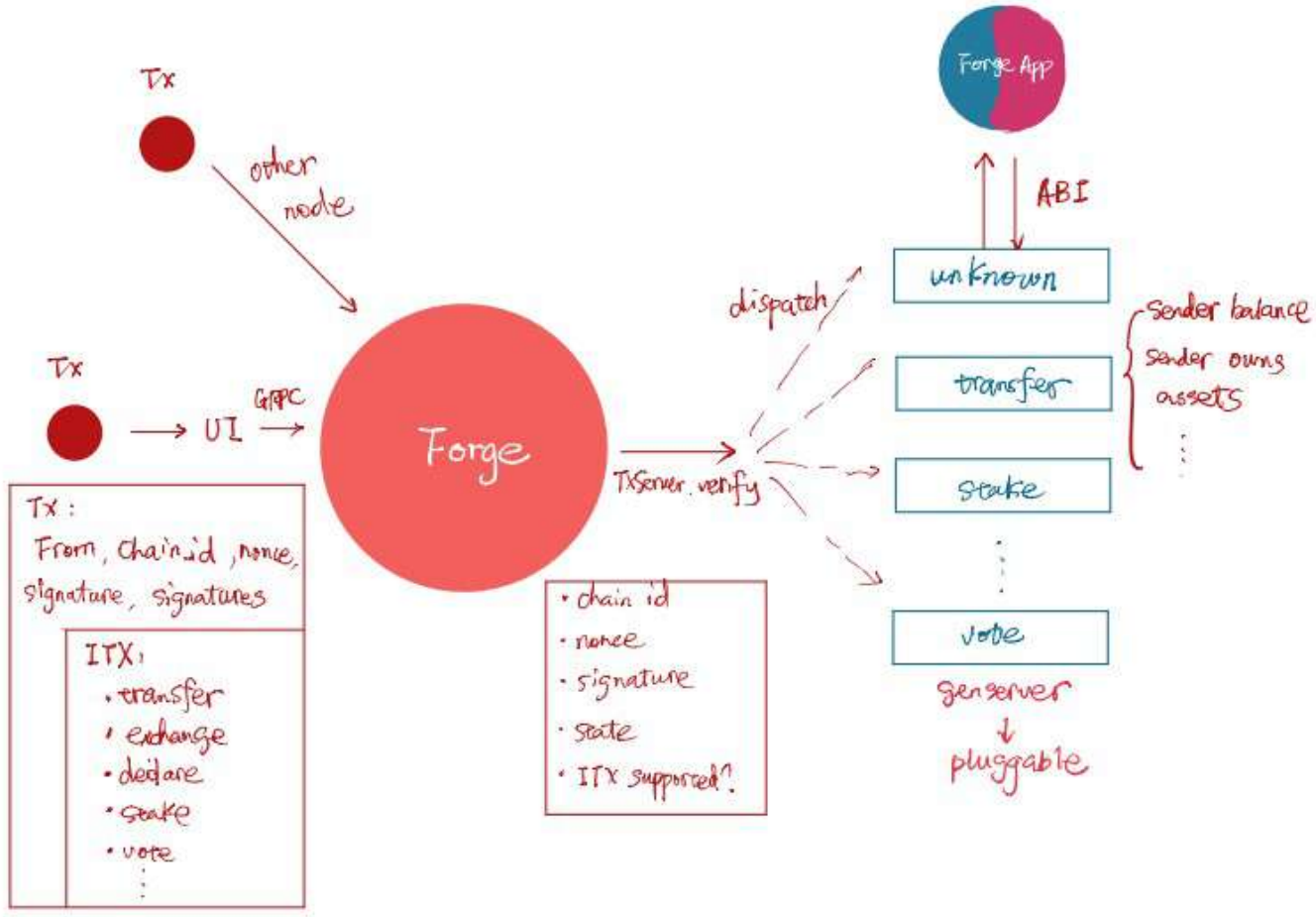
Faking a demo is easy but build a real world DB is HARD

- state consistency across the world
- crash recovery
- complicated transaction processing
- performance
- logging / monitoring / alerting
- robust / extensible

ArcBlock Forge: the Ruby on Rails (RoR) dream

- Full-stack development framework for building apps backed by decentralized DB
  - Build a dApp on its own chain should be easy enough
- developers can use many languages to build dApps based on Forge
  - currently erlang / elixir (first class citizen), nodejs and python
- A wholistic framework that all the batteries are included
  - transfer, exchange, create_asset, stake, vote,
- A powerful CLI that alleviate most day to day work
  - e.g. `forge init`, `forge start`, etc.
- A ready-to-use UI to explore the chain and manage node / accounts / states
- It could be used for applications that requires decentralized, public verifiable database, not limited to cryptocurrency world

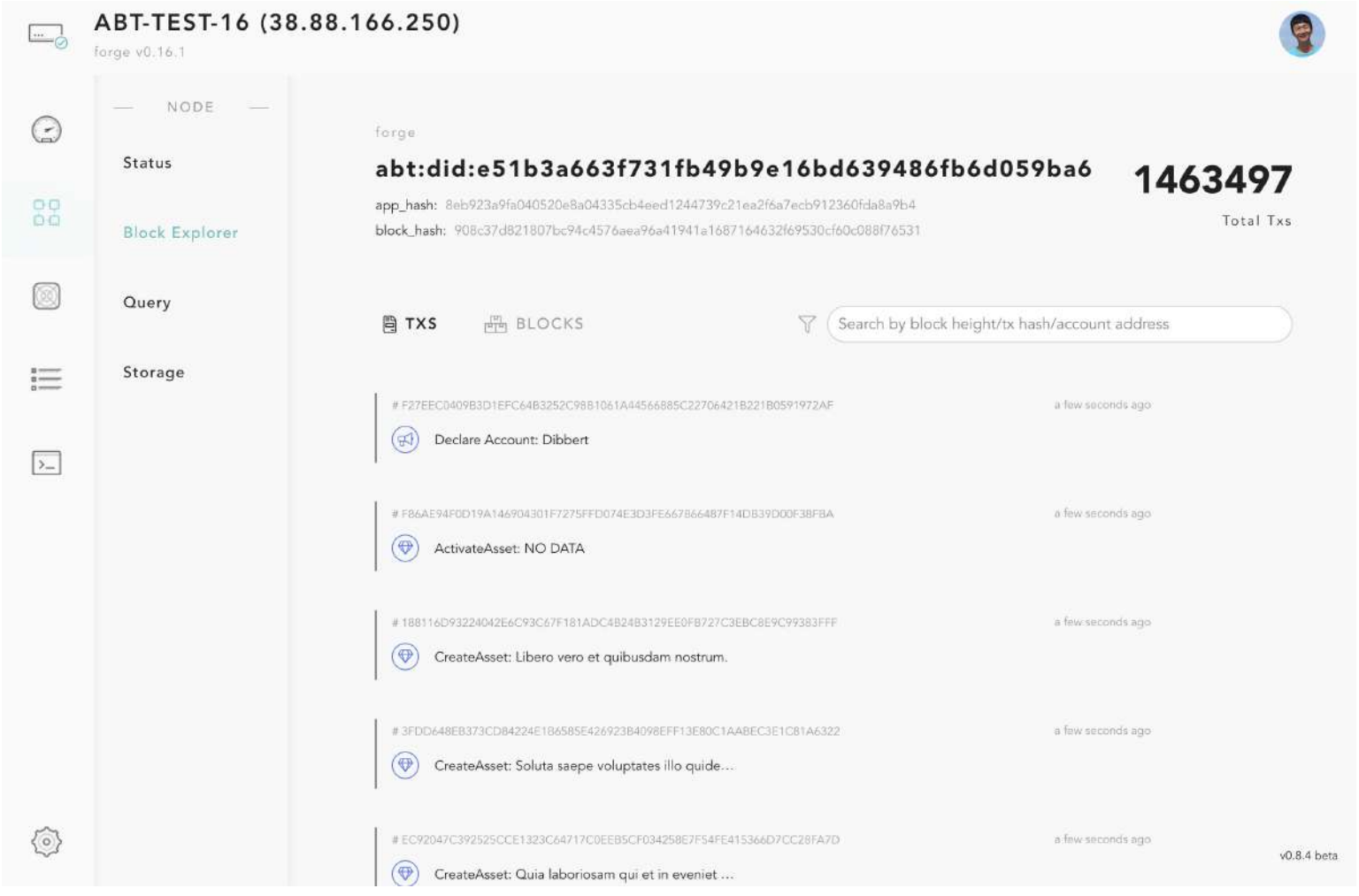
Forge Tx Flow



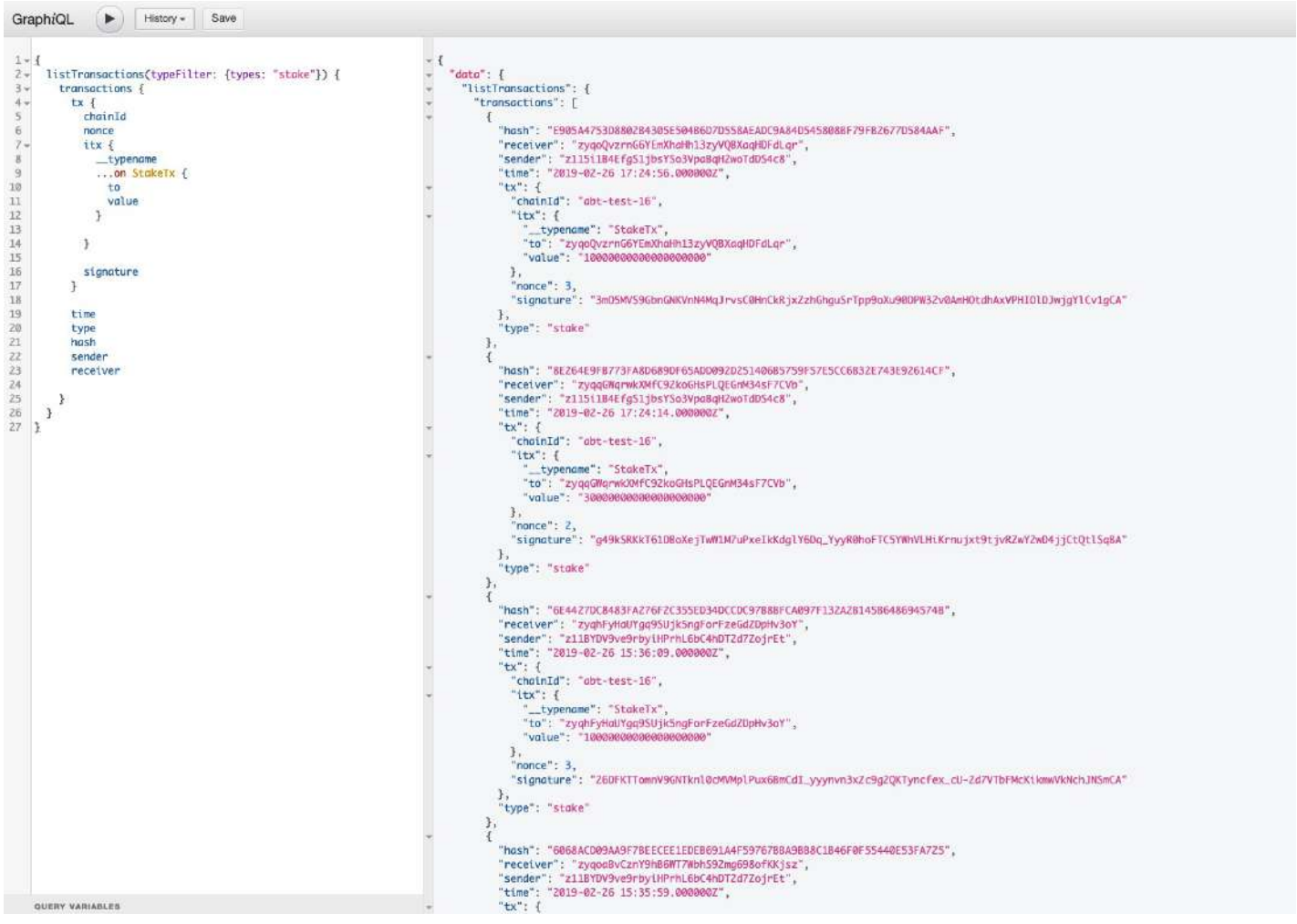
Forge Tx Pipeline


```
use ForgePipe.Builder

pipe Forge.Pipe.VerifyInfo,
  conditions: [
    [
      "info.itx.to !== \"\" and (info.itx.value !== nil or not Enum.empty?(info.itx.assets))",
      :insufficient_data
    ],
    ["info.sender_state !== nil", :invalid_sender_state]
  ]

pipe Forge.Pipe.VerifyItxSize, value: [[:itx, :assets]]
pipe Forge.Pipe.VerifyBalance, state: :sender_state, value: [:itx, :value]
pipe Forge.Pipe.ExtractReceiver, from: [[:itx, :to]]

pipe Forge.Pipe.ExtractState,
  from: :receiver,
  status: :invalid_receiver_state,
  to: :receiver_state

pipe Forge.Pipe.ExtractState,
  from: [:itx, :assets],
  status: :invalid_asset,
  to: [:priv, :assets]


pipe Forge.Pipe.VerifyTransferrable, assets: [:priv, :assets]
pipe Forge.Pipe.VerifyOwner, assets: [:priv, :assets], state: :sender_state
```

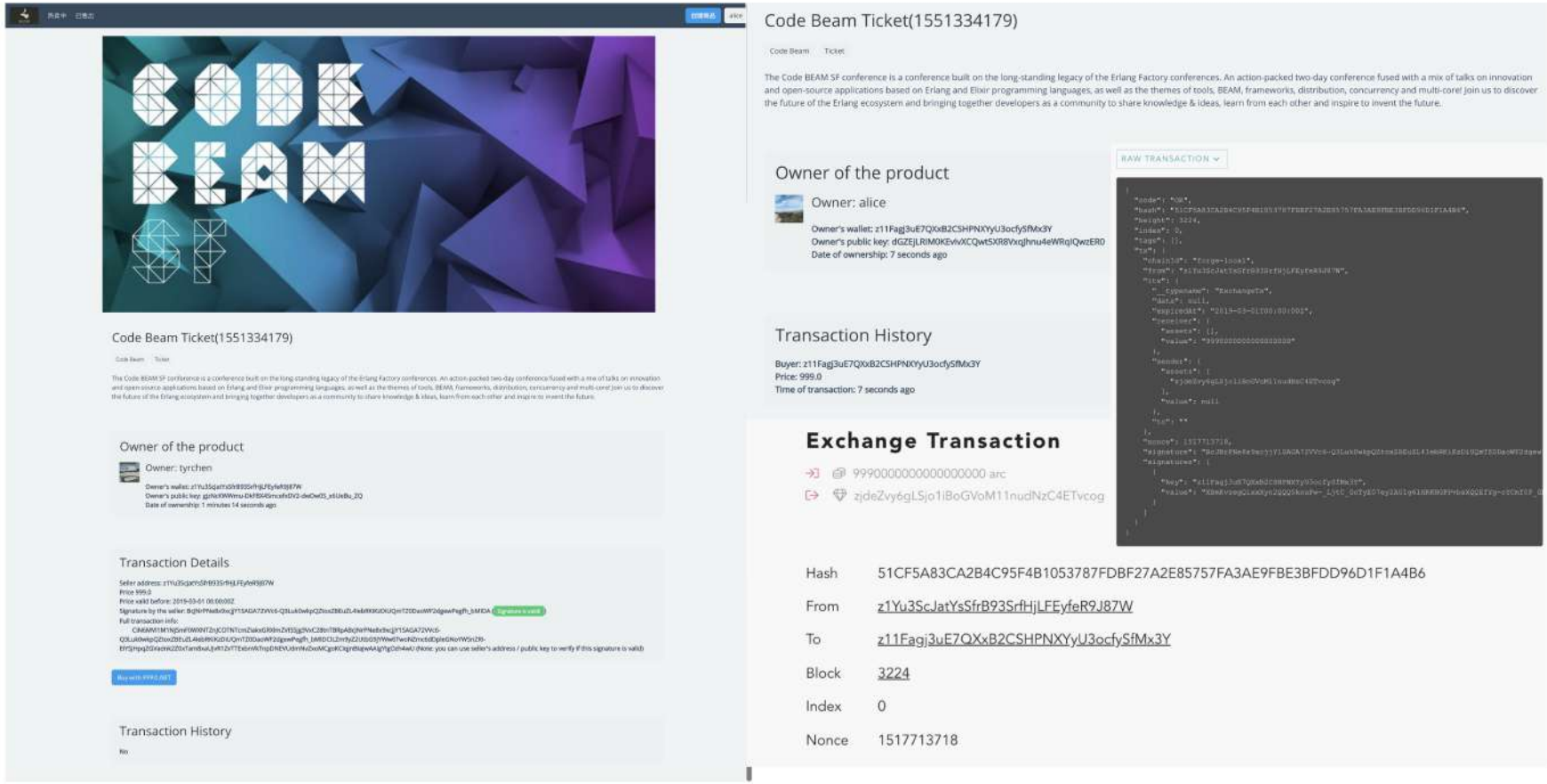
The Forge Block / DB explorer



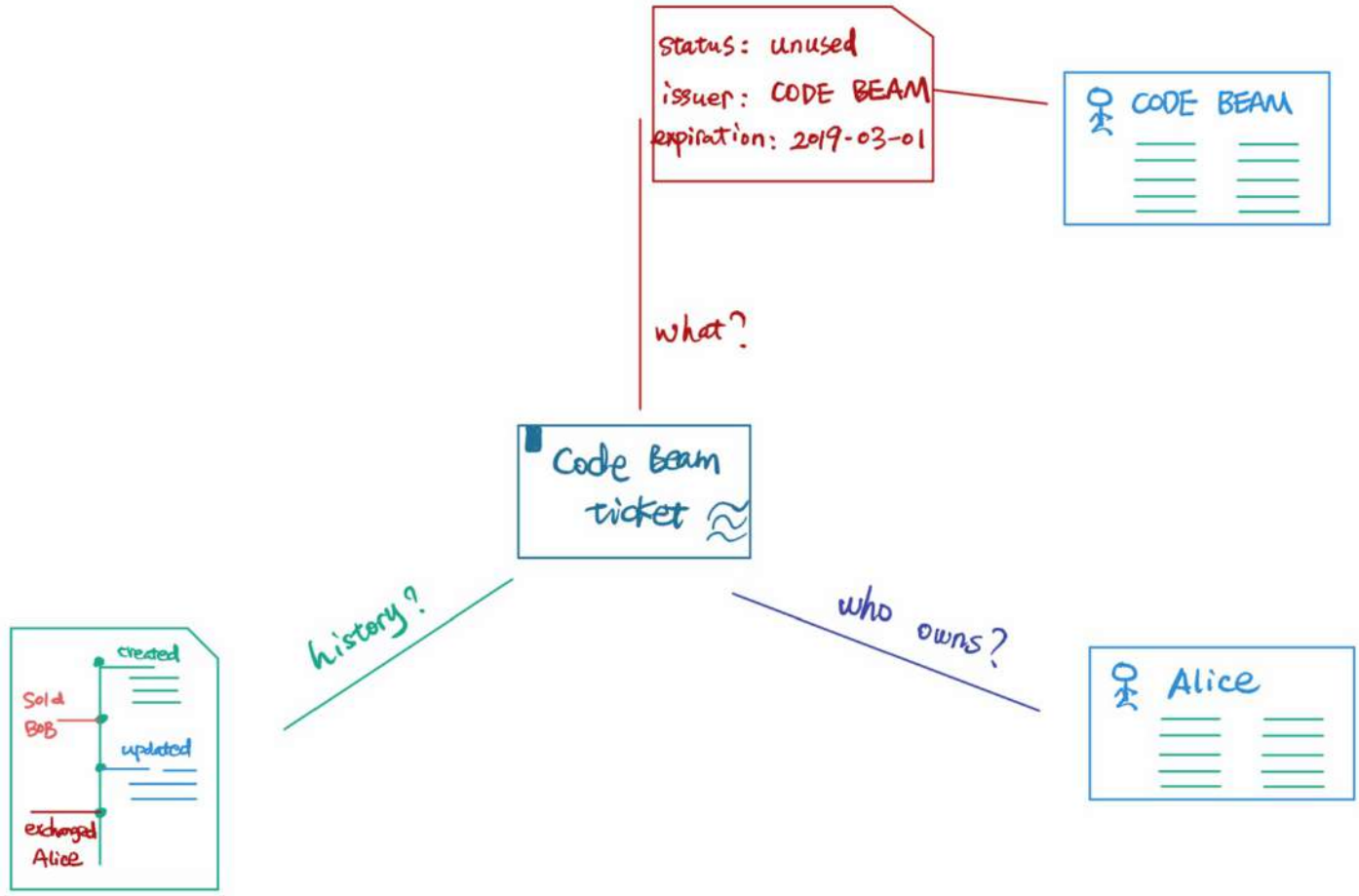
The query language



Applications built on Forge



Verifiable ticket



Status of Forge

- will be open sourced soon (a few weeks ~ a few months)
  - documentation / getting started guide
  - example applications
  - make SDK for nodejs / python / elixir mature
- features / tasks on the framework
  - performance tuning
  - stress / security testing
  - more common transactions
  - make statedb indexable
  - better deployment experience
- more iterations on testnet

The future is on decentralization

- data shall be owned by the owner, not the application
- building decentralized DB shall be as easy as what we're doing now
- public-verifiable data/code will be the most important factor in next generation web

Thank You

ArcBlock

www.arcblock.io