Introduction to Move and Sui

Speaker: Henry Duong, Developer Relations Engineer

MystenLabs

Outline

Move

A safe, asset-oriented language

Sui Move

Combining Move with an object-oriented data model

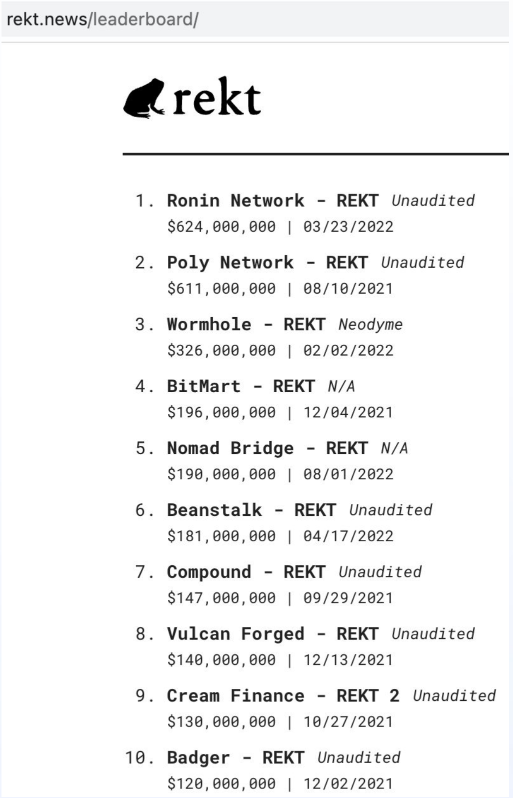
Sui System

Scaling with an object-oriented data model

Move

Smart Contract Safety

- Vulnerabilities are a continual and existential threat to broader adoption
- Human errors will always exist
- Safer languages by design, and advanced testing and verification tooling are the ways forward



rekt.news/leaderboard/

rekt

1. **Ronin Network - REKT** *Unaudited*
   $624,000,000 | 03/23/2022
2. **Poly Network - REKT** *Unaudited*
   $611,000,000 | 08/10/2021
3. **Wormhole - REKT** *Neodyme*
   $326,000,000 | 02/02/2022
4. **BitMart - REKT** *N/A*
   $196,000,000 | 12/04/2021
5. **Nomad Bridge - REKT** *N/A*
   $190,000,000 | 08/01/2022
6. **Beanstalk - REKT** *Unaudited*
   $181,000,000 | 04/17/2022
7. **Compound - REKT** *Unaudited*
   $147,000,000 | 09/29/2021
8. **Vulcan Forged - REKT** *Unaudited*
   $140,000,000 | 12/13/2021
9. **Cream Finance - REKT 2** *Unaudited*
   $130,000,000 | 10/27/2021
10. **Badger - REKT** *Unaudited*
    $120,000,000 | 12/02/2021

Smart Contract General Characteristics

- Specialized to do three main things:
  - Define new asset types
  - Read, write, and transfer assets
  - Check access control policies
- Thus SC languages should support:
  - Safe abstraction for custom assets, ownership and access
  - Strong Isolation - must be easy to write safe code

The structure of a language affects the speaker’s worldview or cognition.

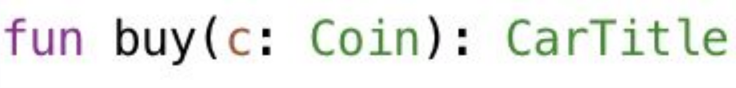
- Sapir-Whorf hypothesis

Limitations in Previous Smart Contract Languages

- Cannot pass or return an asset as a parameter
- Cannot store an asset in a data structure
- Allow a function caller to borrow an asset
- Declare an asset type in Contract A that’s used in Contract B
- Take an asset outside of the contract that created it

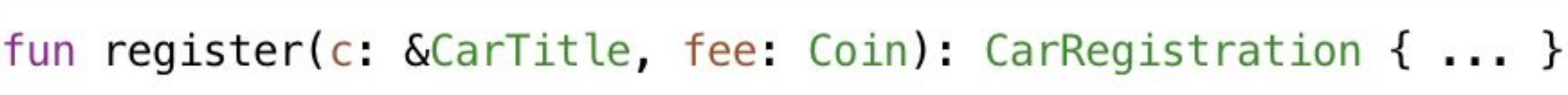
Example: Assets and Ownership

“If you **give** me a coin, I will **give** you a car title”



```
fun buy(c: Coin): CarTitle
```

“If you **show** me your title and **pay** a fee, I will **give** you a car registration”



```
fun register(c: &CarTitle, fee: Coin): CarRegistration { ... }
```

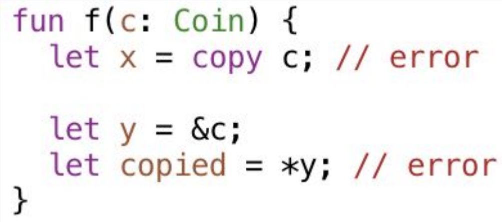
CarTitle, CarRegistration, Coin are user-defined types declared in different modules.

Can flow across trust boundaries without losing integrity

Example: Safe Type System

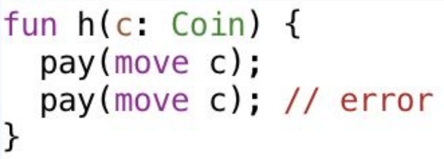
Protection against:

Duplication



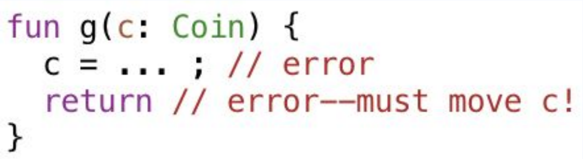
```
fun f(c: Coin) {
  let x = copy c; // error

  let y = &c;
  let copied = *y; // error
}
```

“Double-spending”



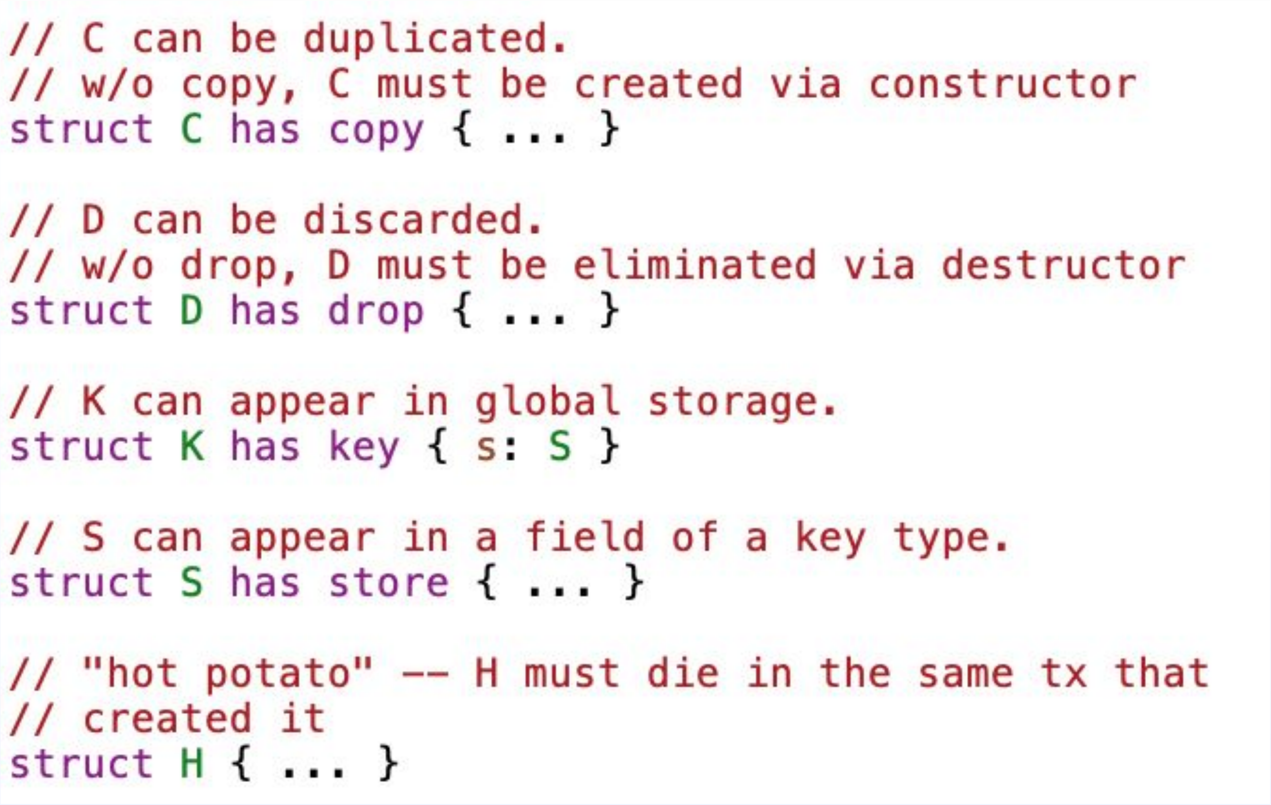
```
fun h(c: Coin) {
  pay(move c);
  pay(move c); // error
}
```

Destruction



```
fun g(c: Coin) {
  c = ... ; // error
  return // error--must move c!
}
```

Example: Fine-grained Control Over Assets



```
// C can be duplicated.
// w/o copy, C must be created via constructor
struct C has copy { ... }

// D can be discarded.
// w/o drop, D must be eliminated via destructor
struct D has drop { ... }

// K can appear in global storage.
struct K has key { s: S }

// S can appear in a field of a key type.
struct S has store { ... }

// "hot potato" -- H must die in the same tx that
// created it
struct H { ... }
```

Precise, Efficient, and Accessible Formal Verification

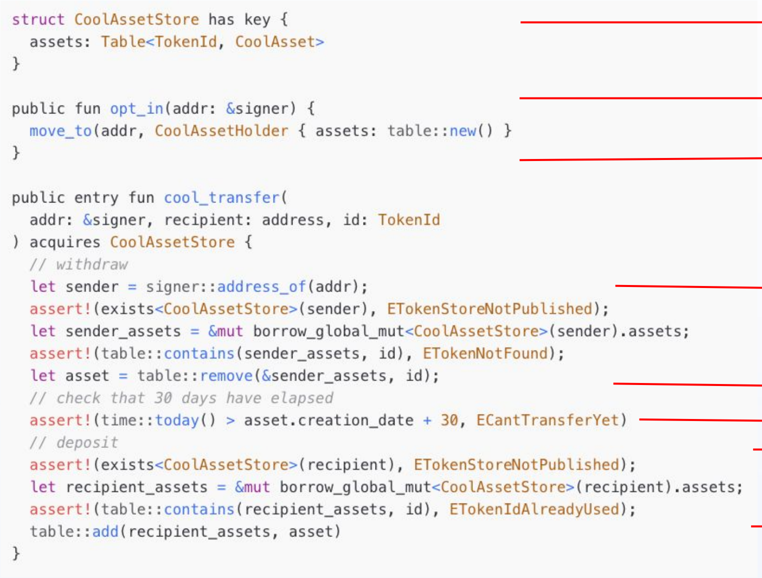
- Specification language integrated with compiler, Prover integrated with Move CLI
- Fast/stable enough to run in CI (and does for Move + Move-powered blockchain repos)
- Move stdlib is specified and verified
  - Community contributors are able to verify code, not just experts!
- https://github.com/move-language/move/tree/main/language/move-prover

Summary of Characteristics of Move

- No dynamic dispatch (no re-entrancy)
- No mixing of aliasing and mutability (similar to Rust)
- Type/memory/resource safety enforced by bytecode verifier
- Strong isolation aka “robust safety” by default (https://arxiv.org/abs/2110.05043)
- SafeMath by default
- Co-developed with the Move Prover formal verification tool
- **Platform agnostic and the JavaScript of Web3**

Sui Move

Diem-style Move: Transfer NFT with Lockup Period



```
struct CoolAssetStore has key {
  assets: Table<TokenId, CoolAsset>
}

public fun opt_in(addr: &signer) {
  move_to(addr, CoolAssetHolder { assets: table::new() }
}

public entry fun cool_transfer(
  addr: &signer, recipient: address, id: TokenId
) acquires CoolAssetStore {
  // withdraw
  let sender = signer::address_of(addr);
  assert!(exists<CoolAssetStore>(sender), ETokenStoreNotPublished);
  let sender_assets = &mut borrow_global_mut<CoolAssetStore>(sender).assets;
  assert!(table::contains(sender_assets, id), ETokenNotFound);
  let asset = table::remove(&sender_assets, id);
  // check that 30 days have elapsed
  assert!(time::today() > asset.creation_date + 30, ECantTransferYet)
  // deposit
  assert!(exists<CoolAssetStore>(recipient), ETokenStoreNotPublished);
  let recipient_assets = &mut borrow_global_mut<CoolAssetStore>(recipient).assets;
  assert!(table::contains(recipient_assets, id), ETokenIdAlreadyUsed);
  table::add(recipient_assets, asset)
}
```

1. Define table to hold multiple NFT’s of the same type

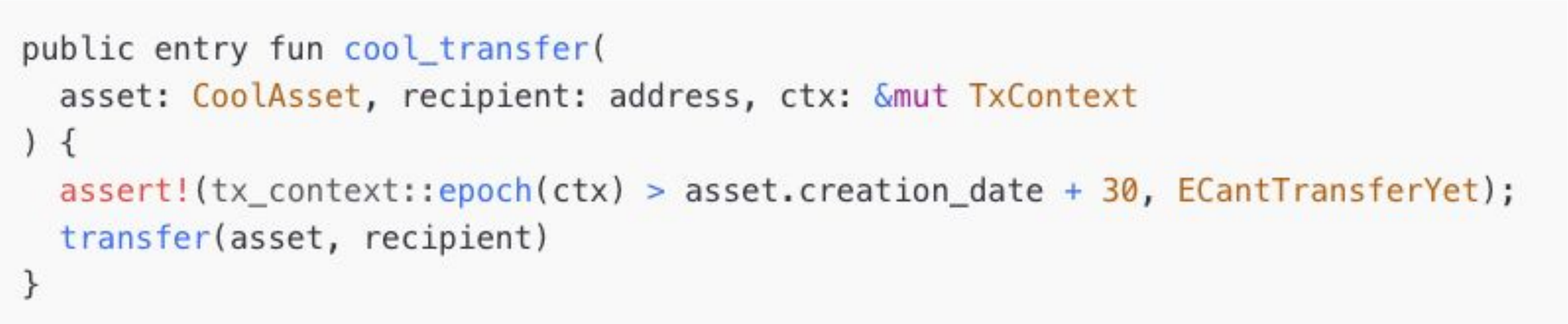
2. Recipient opts in by publishing table

3. Look up + remove asset from sender

4. Check lockup period

5. Transfer to recipient

Same Example in Sui Move



```
public entry fun cool_transfer(
  asset: CoolAsset, recipient: address, ctx: &mut TxContext
) {
  assert!(tx_context::epoch(ctx) > asset.creation_date + 30, ECantTransferYet);
  transfer(asset, recipient)
}
```

1. Define table to hold multiple NFT's of same type

No longer needed–Sui addresses distinguish objects by ID

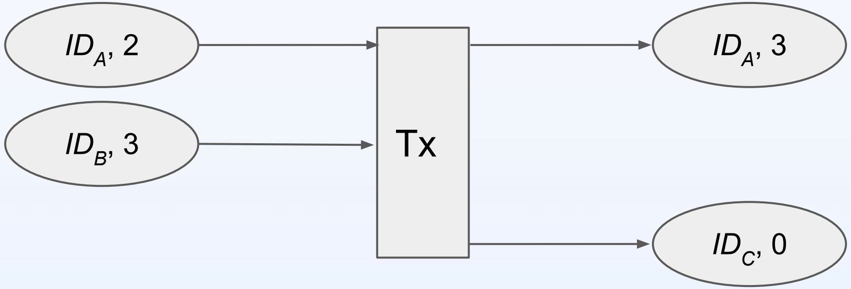
2. Recipient opts in by publishing table

No longer needed–any Sui address can receive any object type

3. Look up + remove asset from sender **Sui runtime does this for you**
4. Check lockup period
5. Transfer to recipient

Sui Move is Fully Asset-centric

- Global state is Map<ObjectID, Object>
- All objects have stable, globally unique ID’s: “everything is an NFT”
- All objects have ownership metadata checked by runtime
  - Saves programmers from many missing auth check bugs
- All tx inputs, outputs expressed in terms of objects
- Tx dependencies are explicit, statically known



Ownership and Access in Sui Move

- Types of Ownership

  If the object **O** is owned by:

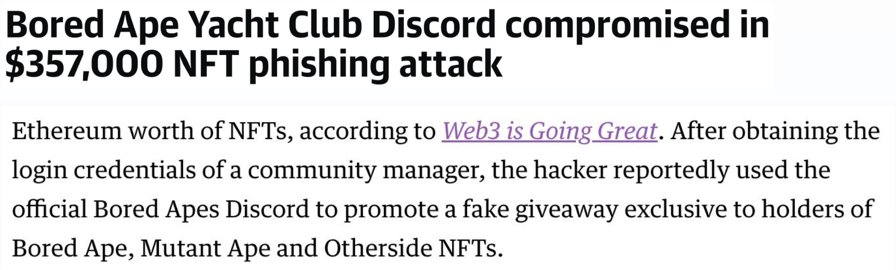
  - address **A**: only a transaction signed by **A** can use **O**
  - another object **P**: a transaction that includes **the entire ancestral chain of O** can use **O**
  - shared: anyone can use **O**

- Access expressed in Move syntax:

  function ***public entry fun f(consume: T, write: &mut T, read: &T)***

  - **T**: transfer, delete, write, read
  - **&mut T**: write, read
  - **&T**: read

Human Readable Signing Requests and Permissions



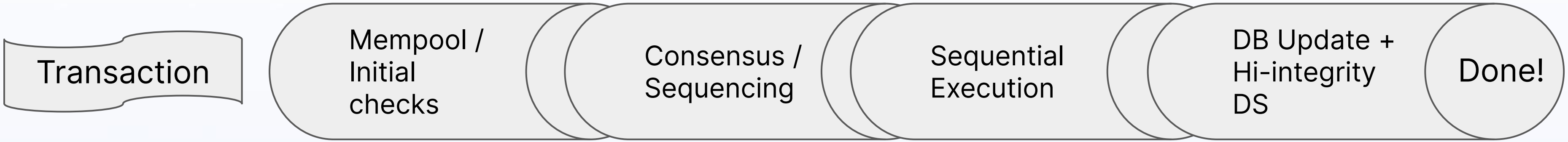
Bored Ape Yacht Club Discord compromised in $357,000 NFT phishing attack

Ethereum worth of NFTs, according to *Web3 is Going Great*. After obtaining the login credentials of a community manager, the hacker reportedly used the official Bored Apes Discord to promote a fake giveaway exclusive to holders of Bored Ape, Mutant Ape and Otherside NFTs.

- Sui tx format enables Android/iOS-style permission requests for wallets:
  - objects used
  - read/write/transfer permissions
  - in some cases, can predict tx effects via local pre-execution or static analysis
- fun safe_giveaway(ape: &BoredApe):
  - **“This app wants to read your ape–is that ok?”**
- fun malicious_giveaway(ape: BoredApe)
  - **“This app wants to transfer your ape–is that ok?”**

Sui System

The Classic Blockchain Architecture



	Transaction	Mempool / Initial checks	Consensus / Sequencing	Sequential Execution	DB Update + Hi-integrity DS	Done!
What a validator / miner does		p2p flood & selection on fee	Sequence all transactions in blocks	Execute each transaction (global lock)	Update DB, indexes, crypto (merkle trees)	
Issues:			BFT consensus: seconds latency, traditionally low throughput	Single core does all computations	Added latency of store, blocks, and crypto computations	

Sui System Design Goal

A **replicated** auditable **transaction processing** system, that allows writes constrained by **user-defined smart contracts** and public reads, and that is **robust** to byzantine failures.

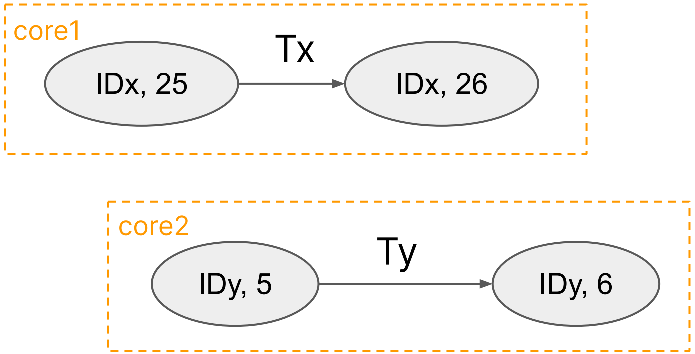
The Sui System Ingredients

Express computations on objects and allow parallelism

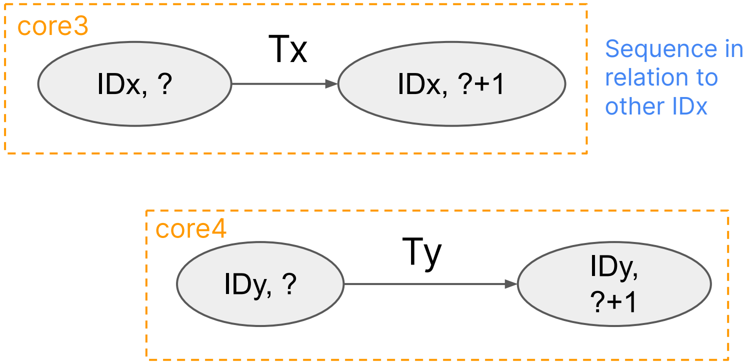
Avoid consensus when possible; very scalable consensus when necessary.

Do housekeeping after finality

Object versions ⇒ Parallel execution



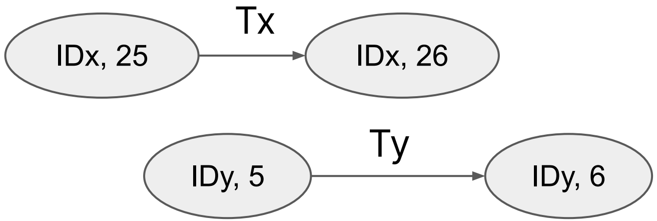
Owned object transactions can always be executed in parallel once their input object ID/version is known



Shared object also can be executed in parallel to each other (but sequentially for each shared object)

What Transactions can avoid consensus?

Owner can determine sequence

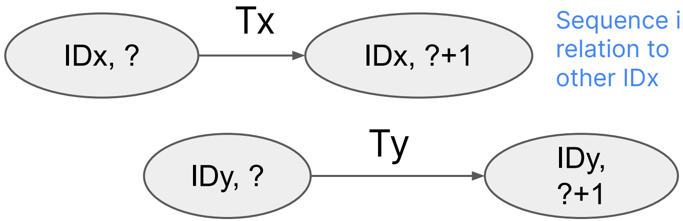


All inputs are owned objects, owned by the same address

⇒

Only need **reliable broadcast**
(check sequencing from owner)

System must determine sequence



Sequence in relation to other IDx

Mixture of owned objects (same address) and **shared object inputs**

⇒

Need consensus to sequence in relation to other transactions on same shared objects
(collaborative creation of sequence)

Narwhal
&
Bullshark

What Can Happen After Finality

What is on the latency critical path

Transaction Dissemination
Execution & finality certificate

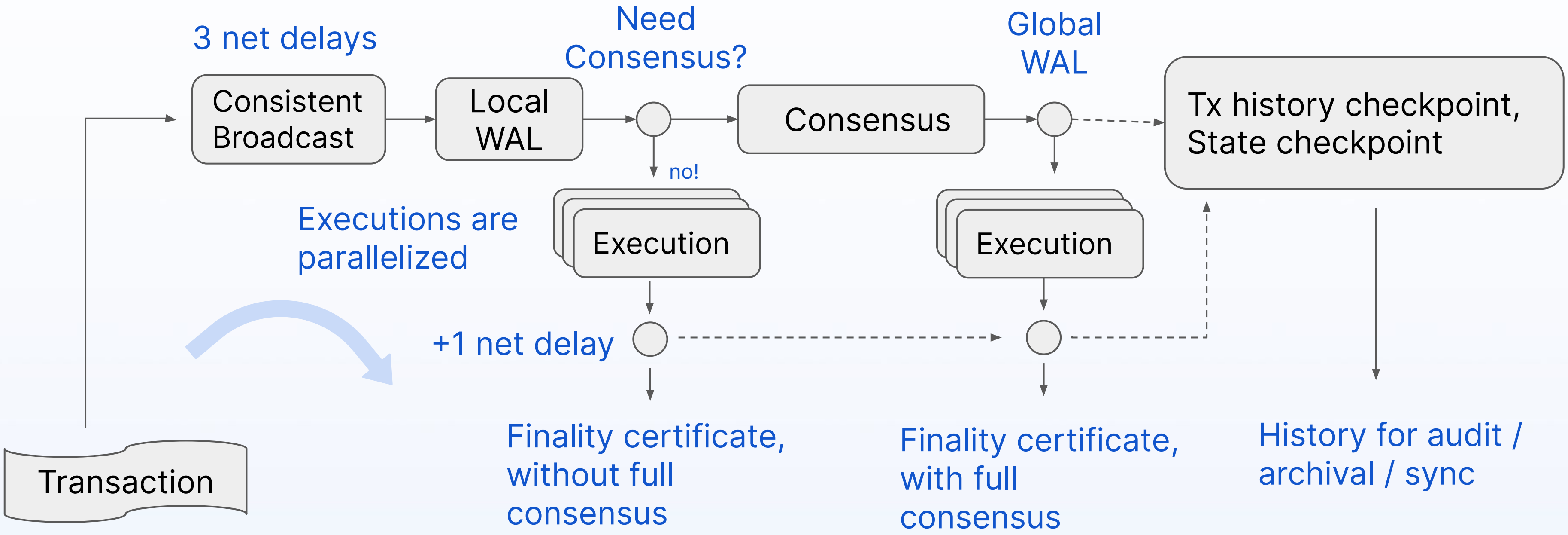
->

Finality, lets a user complete a flow, do another (dependent) transaction, go offline.

What can happen later

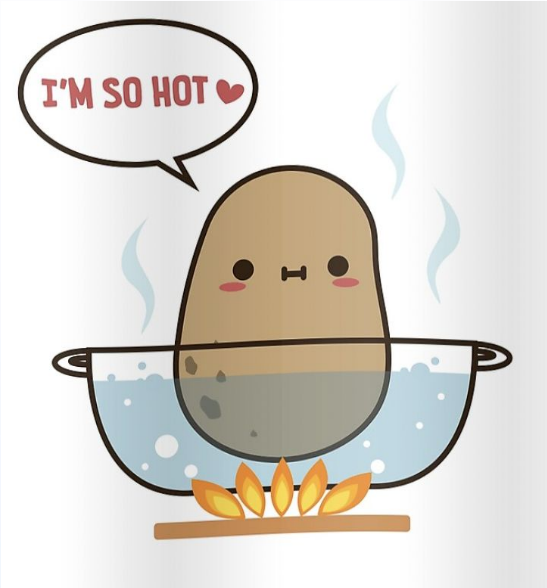
Gather Transactions in blocks
Gossip to update full nodes
Sync nodes that are behind
Maintain Merkle Trees
Determine exact full sequence
Determine exact bitstrings
Determine Checkpoints
Share checkpoints / sync
Reconfiguration management

Sui Validator Architecture


3 net delays
Need Consensus?
Global WAL
Consistent Broadcast
Local WAL
Consensus
Tx history checkpoint, State checkpoint
no!
Executions are parallelized
Execution
Execution
+1 net delay
Transaction
Finality certificate, without full consensus
Finality certificate, with full consensus
History for audit / archival / sync

Object-based Fee Markets

- Worst case scenario = every single tx touches the same shared object **O**, in which case, performance of Sui degrades to a traditional total order based blockchain
- Fee Market Design:
  - Goal: design pricing scheme such that maximizing fee revenue = maximizing throughput
  - Gas price for tx **T** is proportional to the "hotness" of the shared objects **T** touches, tracked by validators
  - Contention on shared object **O** *only* affects prices/QoS for tx's touching **O**
  - Other tx's are unaffected, eliminating noisy neighbors



Get in Touch

- Discord: https://discord.gg/GcFNX4WMrB
- Twitter: @Mysten_Labs
- Developer Docs: https://docs.sui.io/
- GitHub:
  - https://github.com/MystenLabs/sui
  - https://github.com/MystenLabs/awesome-move
  - https://github.com/move-language/move
- Contact: Email: henry@mystenlabs.com
  Twitter/Telegram: @henrydevrel