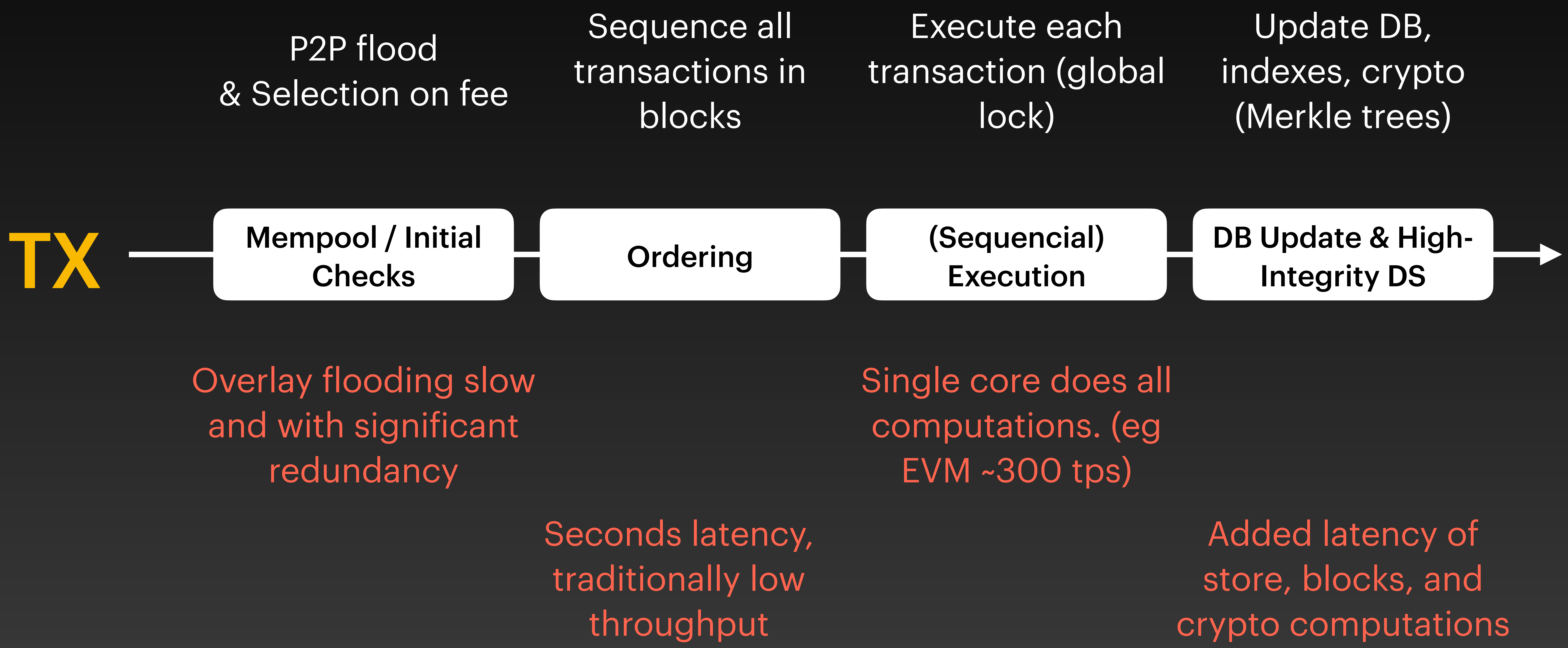
Sui Lutris: A Blockchain Combining Broadcast and Consensus

Alberto Sonnino

Byzantine Fault Tolerance

Byzantine Fault Tolerance

Typical Architecture



Typical Architecture

Mempool / Initial Checks	Ordering	(Sequencial) Execution	DB Update & High-Integrity DS
P2P flood & Selection on fee	Sequence all transactions in blocks	Execute each transaction (global lock)	Update DB, indexes, crypto (Merkle trees)
Overlay flooding slow and with significant redundancy	Seconds latency, traditionally low throughput	Single core does all computations. (eg EVM ~300 tps)	Added latency of store, blocks, and crypto computations

New Architecture

Secure Combination

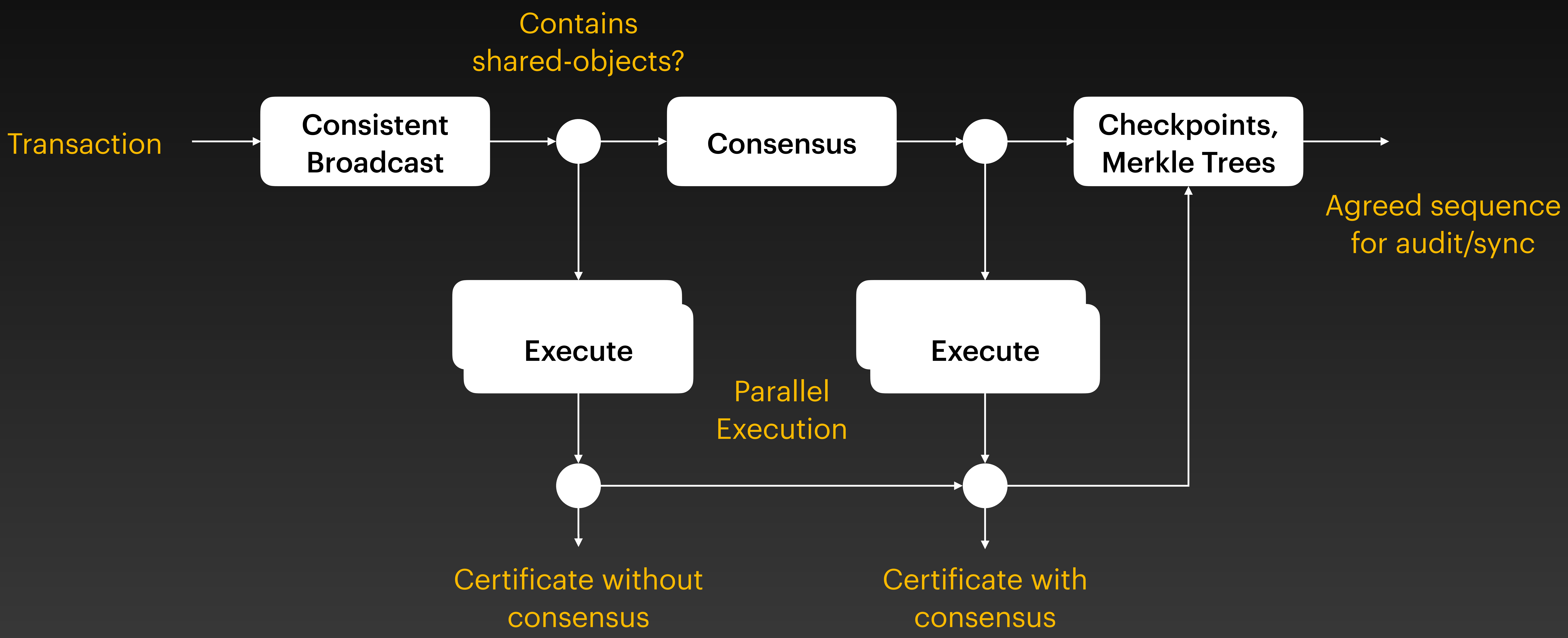
FastPay + Narwhal

Bullshark

= ❤

The Sui Lutris System

Architecture



New Data Model

Consensus is not required

- Coins, balances, and transfers
- NFTs creation and transfers
- Game logic allowing users to combine assets
- Inventory management for games / metaverse
- Auditable 3rd party services not trusted for safety
- ...

New Data Model

Consensus is required*

Increment a publicly-accessible counter

Auctions

Market places

Collaborative in-game assets

...

Consensus only when you need to

New Architecture

Architecture

Owned Objects

- Objects that can be mutated by a single entity
- e.g., My bank account
- **Do not need consensus**

Shared Objects

- Objects that can be mutated my multiple entities
- e.g., A global counter
- **Need consensus**

Sui Objects

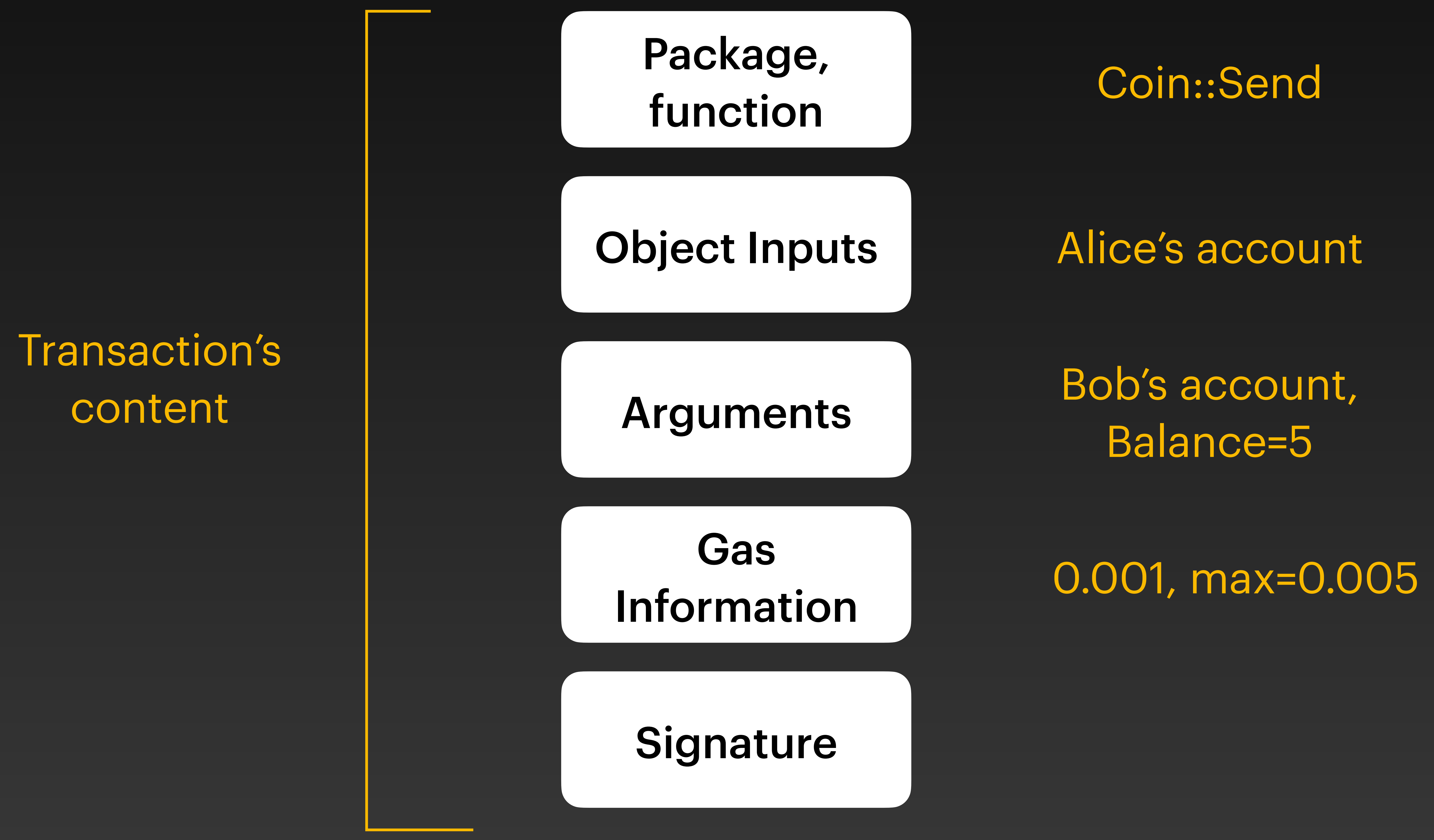
Objects:

- Unique ID
- Version number
- Ownership Information
- Type (shared, owned)

Sui Transaction

Objects:

- Unique ID
- Version number
- Ownership Information
- Type (shared, owned)



Consensus-less Path

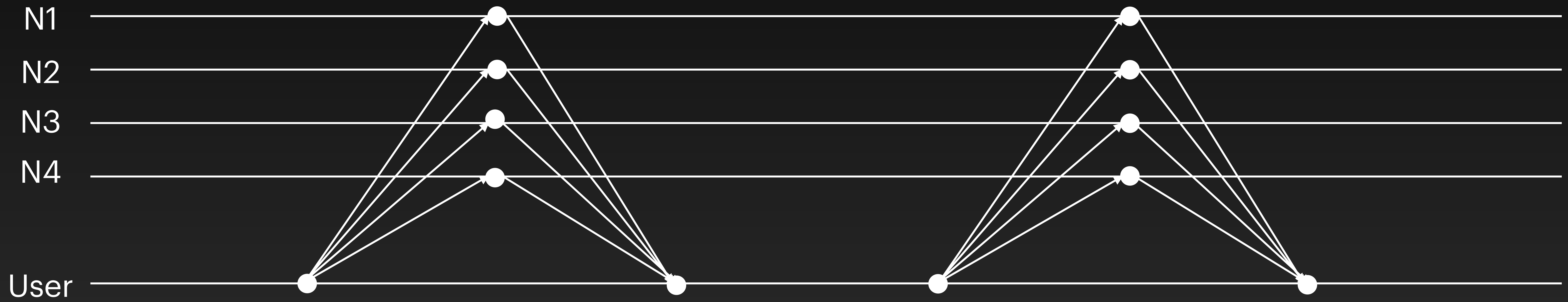
Example Transaction

T1

Inputs: O1 (v10), O2 (v27), O3 (v1001)

Output: Mutate O1, Transfer O2, Delete O3, Create O4

Consensus-less Path



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

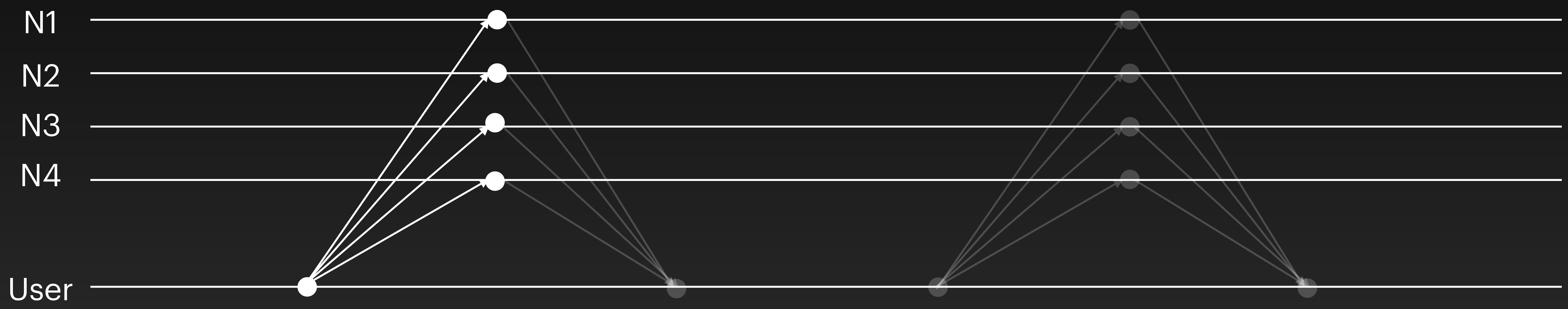
Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Consensus-less Path



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Consensus-less Path

Step 1: Owned object locks & version exist at validator

O1

L1 = (O1, 10)

Owner=X : None

O2

L2 = (O2, 27)

Owner=X : None

O3

L3 = (O3, 1001)

Owner=X : None

We call these “locks”, and are initialised to None.

Consensus-less Path

Step 2: Validator V checks / signs transactions

O1

L1 = (O1, 10)

Owner=X : ~~None~~ T1

O2

L2 = (O2, 27)

Owner=X : ~~None~~ T1

O3

L3 = (O3, 1001)

Owner=X : ~~None~~ T1

Transaction: T1

Inputs: (O1, 10), (O2, 27), (O3, 1001)

Move call details

Signature of X

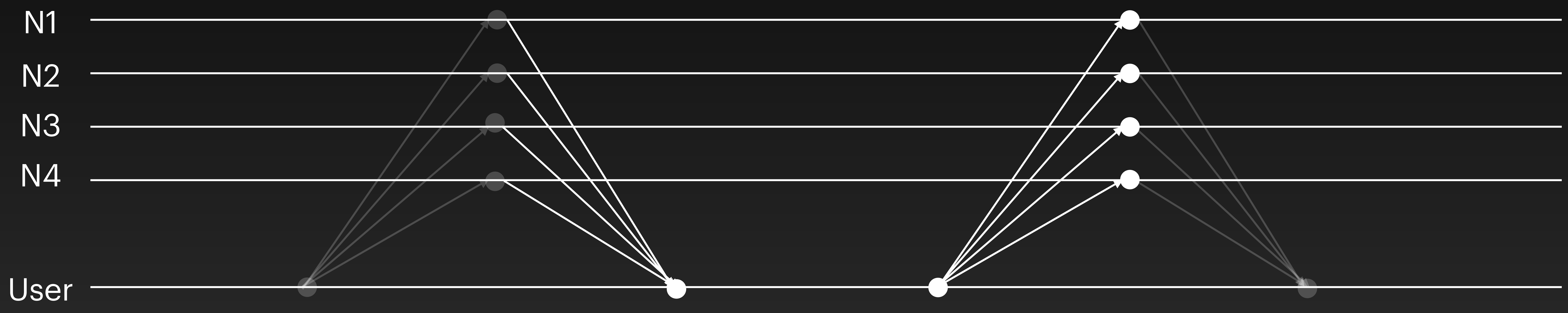
Checks T1 (Validity)

- Well-formed (syntactic)
- Valid Signature from X
- Valid execution function
- Version owned by X

Checks T1 (Broadcast)

- Objects exist and lock is None
- Set lock to T1

Consensus-less Path



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Consensus-less Path

Step 3: Validator V process certificate

O1

L1 = (O1, 10)

Owner=X : ~~None~~ T1

O2

L2 = (O2, 27)

Owner=X : ~~None~~ T1

O3

L3 = (O3, 1001)

Owner=X : ~~None~~ T1

Transaction: T1

Inputs: (O1, 10), (O2, 27), (O3, 1001)

Move call details

Signature of X

Signature (V1, ... V4)

Checks T1 (Validity)

- Again!

Checks T1 (Broadcast)

- Objects exist (with any lock)
- Certificate signed by quorum

Consensus-less Path

Step 4: Validator V executes / signs effect

O1

L1 = (O1, 11)

Owner=X : None

O2

L2 = (O2, 28)

Owner=Y : None

O4

L3 = (O4, 1)

Owner=X : None

Transaction: T1

Inputs: (O1, 10), (O2, 27), (O3, 1001)

Move call details

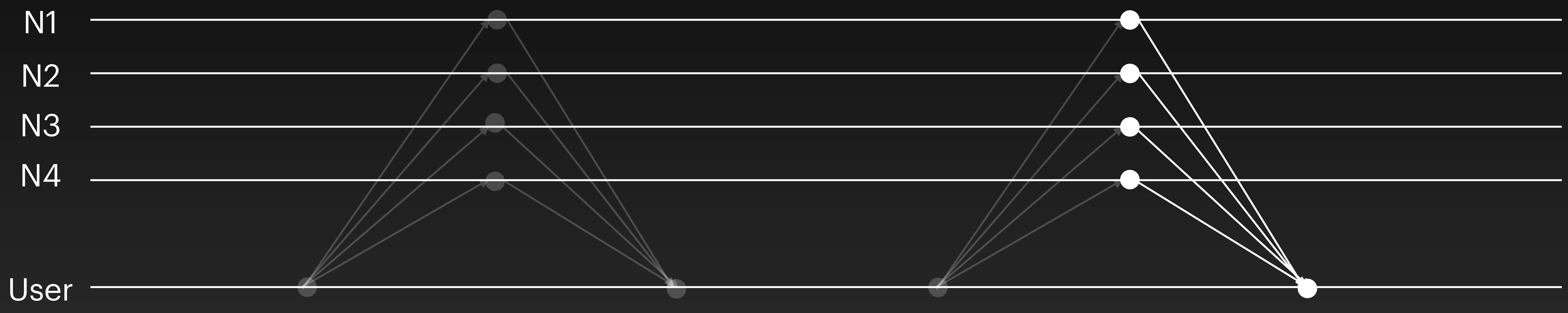
Signature of X

Signature (V1, ... V4)

Execute T1

- O1 mutated
- O2 transferred
- O3 deleted
- O4 created

Consensus-less Path



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

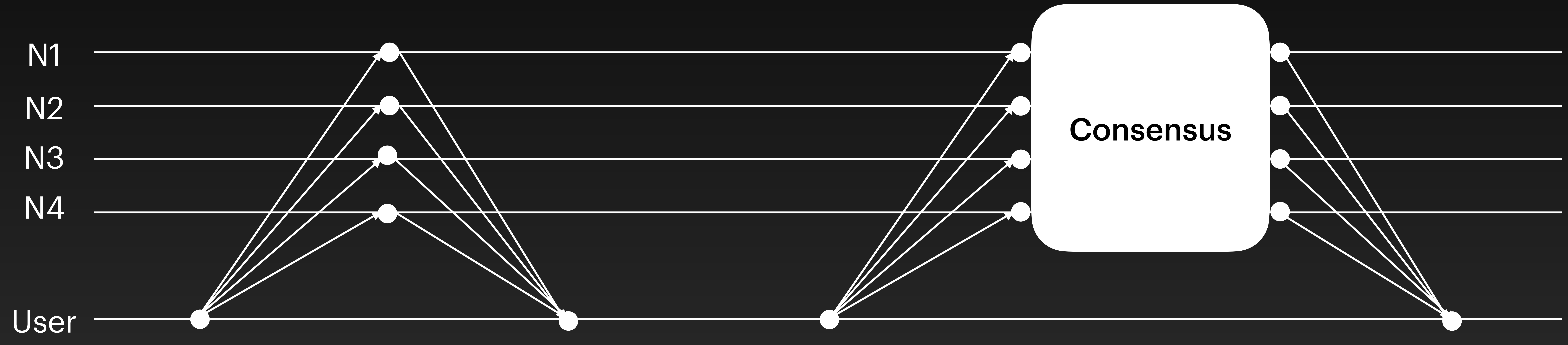
Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Integration with Consensus



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Integration with Consensus

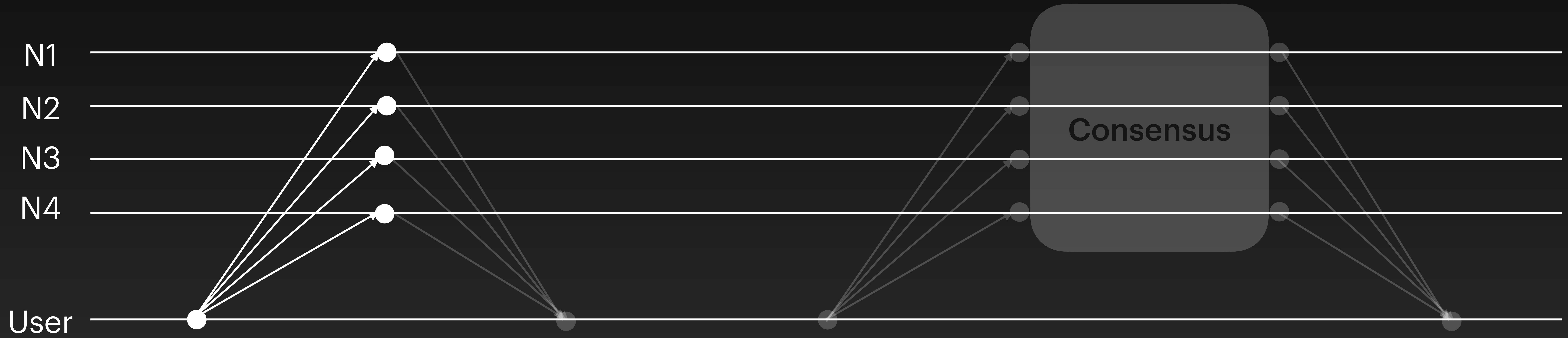
Example Transaction

T2

Inputs: O1 (v10), S2

Output: Mutate O1, Mutate S2, Create O4

Integration with Consensus



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Integration with Consensus

Step 1: Shared object locks exist at validator

O1

L1 = (O1, 10)

Owner=X : None

S2

L2 = (S2, *)

Do not check the version for shared objects

Integration with Consensus

Same as before

Step 2: Validator V checks / signs transactions

O1

L1 = (O1, 10)

Owner=X : ~~None~~ T2

S2

L2 = (S2, *)

Owner=X

Transaction: T2

Inputs: (O1, 10), (S2, *)

Move call details

Signature of X

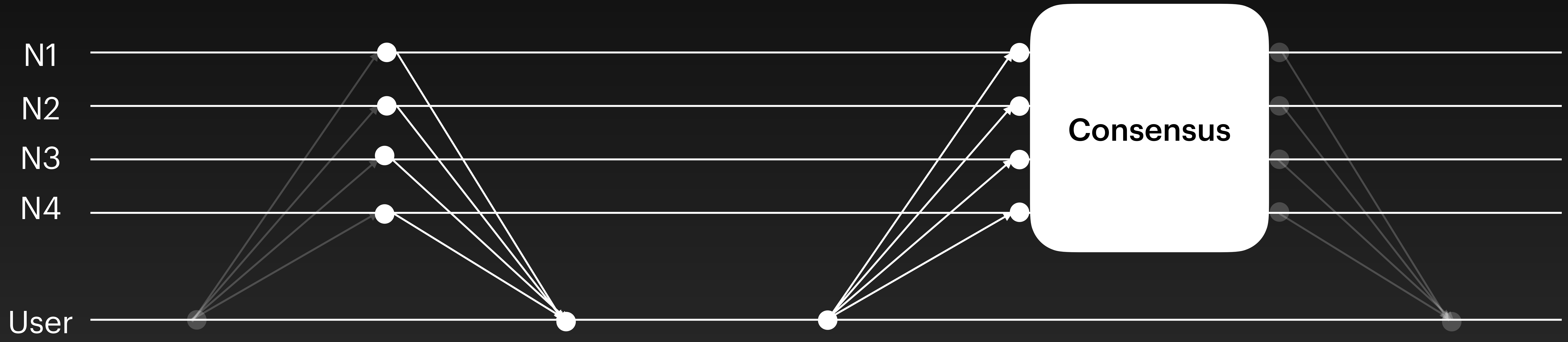
Checks T1 (Validity)

- Well-formed (syntactic)
- Valid Signature from X
- Valid execution function
- Version owned by X

Checks T1 (Broadcast)

- Objects exist and lock is None
- Set lock to T1

Integration with Consensus



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

User gather >2/3 effect signatures for finality

Integration with Consensus

Step 3: After consensus, assign shared objects locks

O1

L1 = (O1, 10)

Owner=X : ~~None~~ T2

S2

L2 = (S2, 4)

Transaction: T2

Inputs: (O1, 10), (S2, *)

Move call details

Signature of X

Assign Shared Locks

- Every node sees the same sequence out of consensus
- So they can all assign the same shared object locks

Integration with Consensus

Same as before

Step 3: Validator V process certificate

O1

L1 = (O1, 10)

Owner=X : ~~None~~ T2

S2

L2 = (S2, 4)

Transaction: T2

Inputs: (O1, 10), (S2, *)

Move call details

Signature of X

Checks T2 (Validity)

- Again!

Checks T2 (Broadcast)

- Objects exist (with any lock)
- Certificate signed by quorum

Integration with Consensus

Same as before

Step 4: Validator V Applies / Signs Effect

O1

L1 = (O1, 11)

Owner=X : None

S2

L2 = (S2, 4)

O4

L3 = (O4, 1)

Owner=X : None

Transaction: T2

Inputs: (O1, 10), (S2, *)

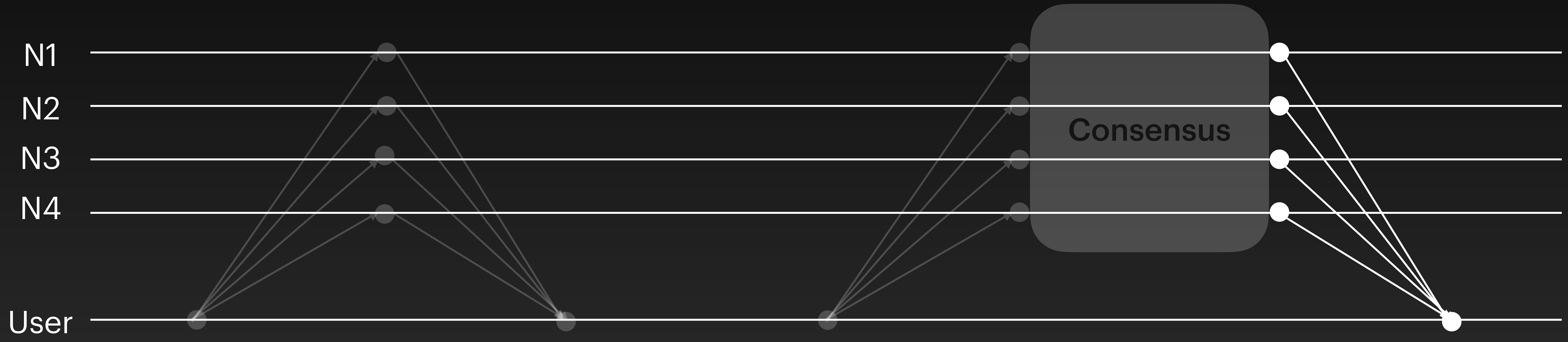
Move call details

Signature of X

Execute T1

- O1 mutated
- S2 mutated
- O4 created

Integration with Consensus



Send T1:

Disseminate the transaction

Echo T1:

Nodes check and sign T1

Cert T1:

User gather >2/3 signatures into a certificate and disseminate it

Effect T1:

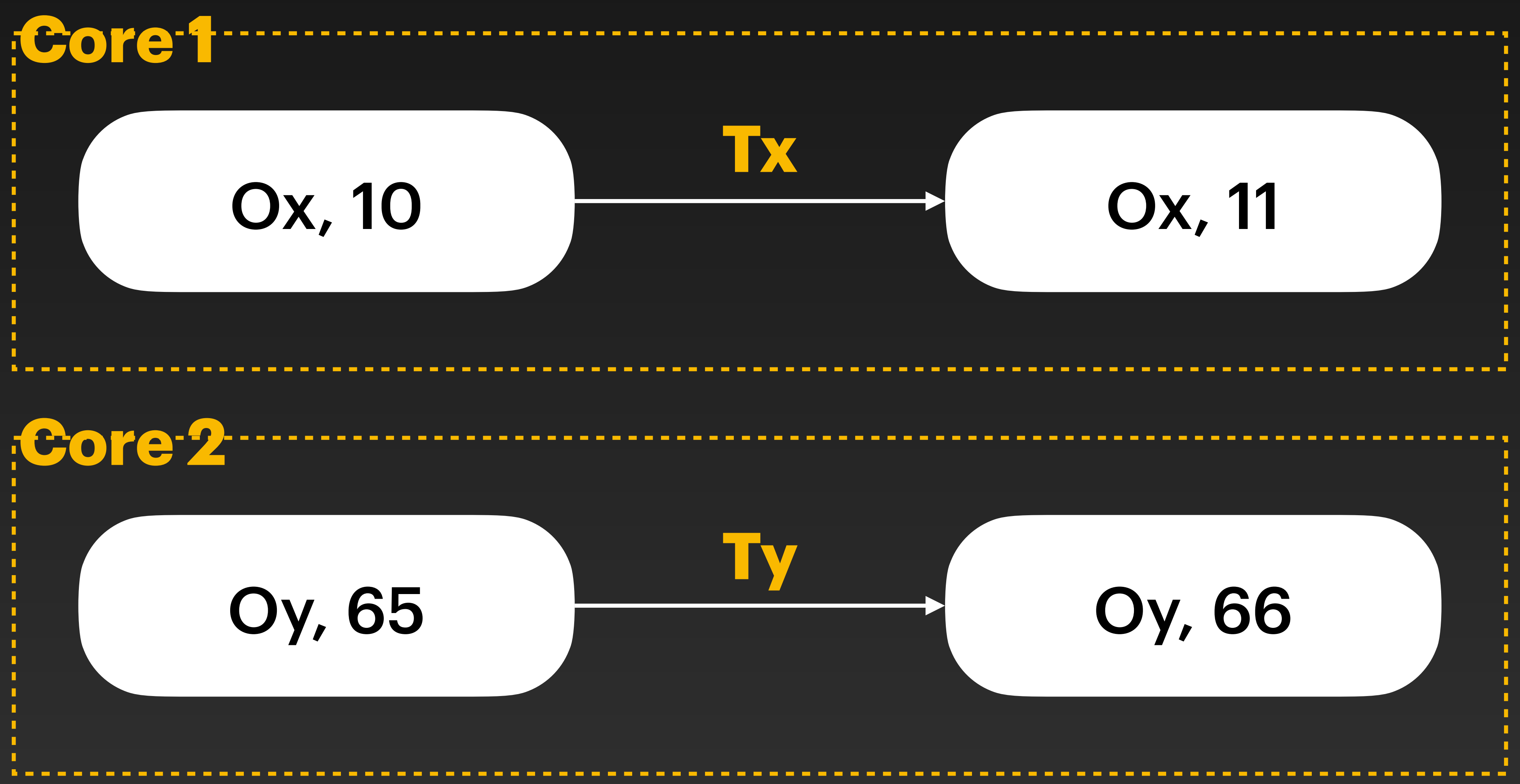
User gather >2/3 effect signatures for finality

Transaction Execution

- First, execute all precedent transactions
- Once there is a certificate, any validator can download Tx and execute

Transaction Execution

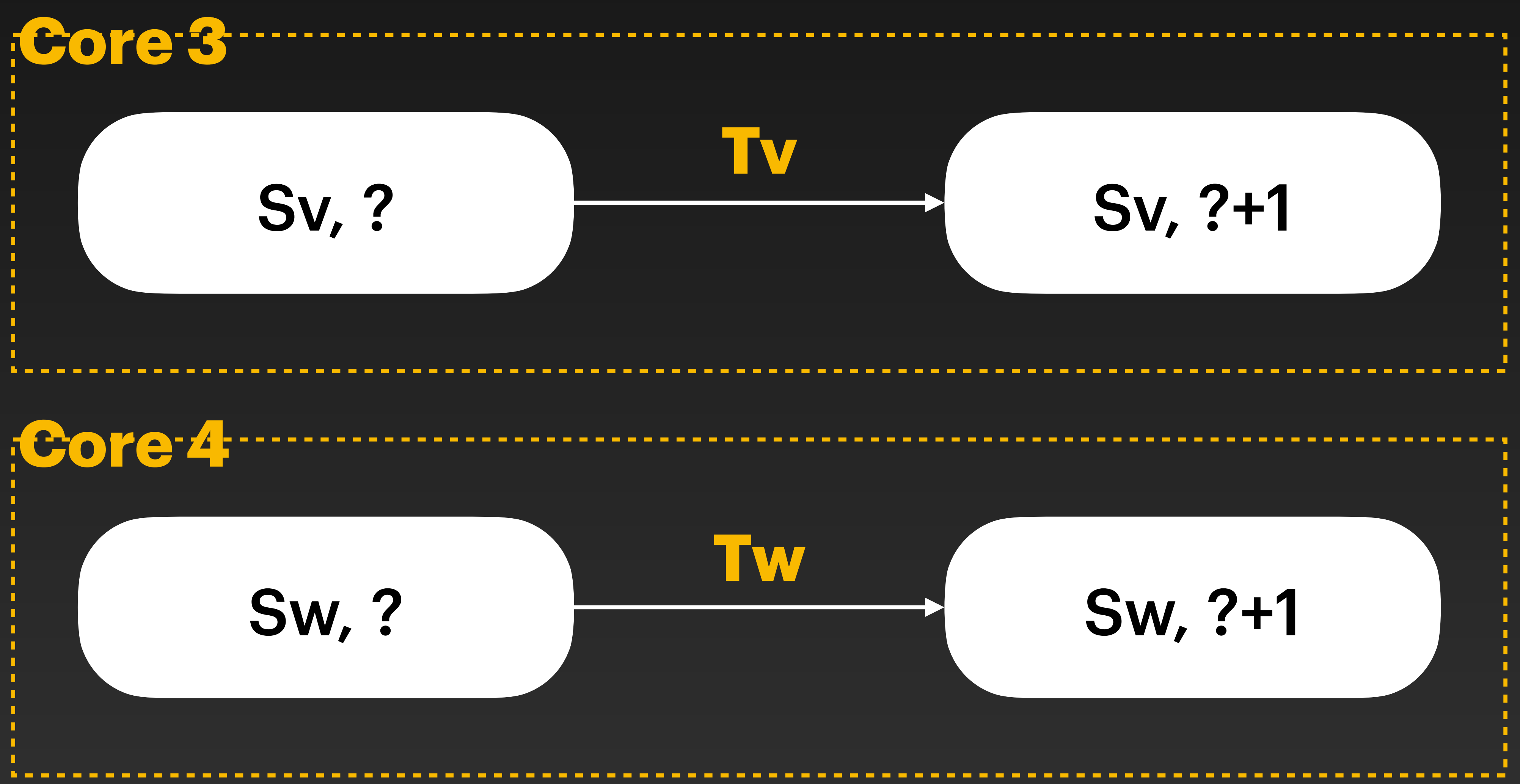
Owned-objects



Always executed in parallel

(once they inputs ID/version are known)

Shared-objects

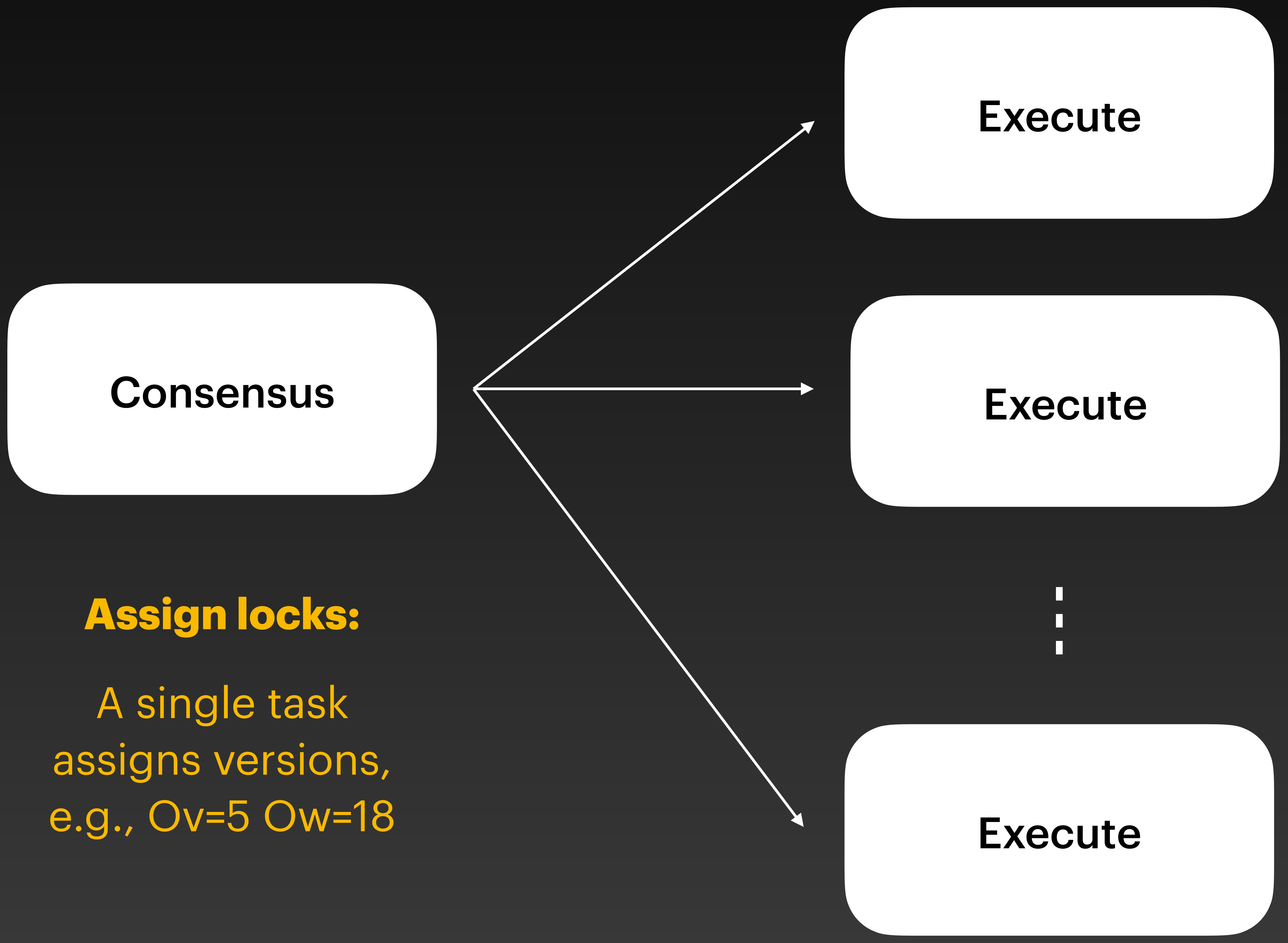


Often executed in parallel

(Sequentially for each shared object)

The Sui System

Shared objects



Assign locks:

A single task assigns versions, e.g., Ov=5 Ow=18

Execute:

Multiple tasks execute (if they can)

Transaction Execution

Schedule

Single task schedules transactions:

(Tx1, Sv) -> 5

(Tx1, Sw) -> 17

...

(Tx2, Sw) -> 6

Execute

Many tasks try to execute transactions:

(Tx1, Sv) == db[Sv]

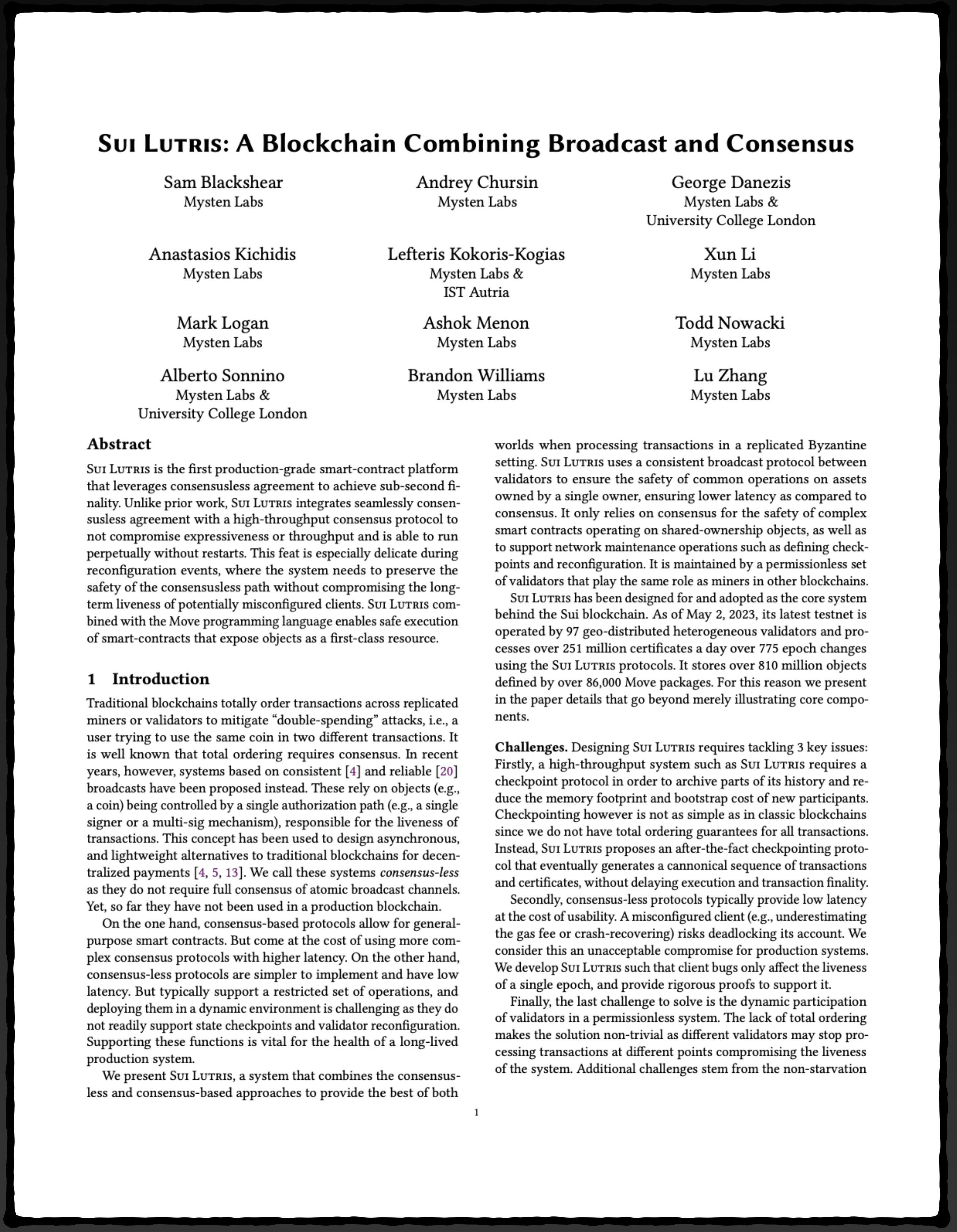
db[Sv] += 1

Missing owned-objects dependency?

- Tell the client
- Synchronise
- Retry

What we didn’t cover

- (Very) Detailed Algorithms
- Checkpointing
- Reconfiguration
- Proofs
- Production-readiness Insights
- …



Sui Lutris: A Blockchain Combining Broadcast and Consensus

Sam Blackshear, Mysten Labs
Andrey Chursin, Mysten Labs
George Danezis, Mysten Labs & University College London
Anastasios Kichidis, Mysten Labs
Lefteris Kokoris-Kogias, Mysten Labs & IST Autria
Xun Li, Mysten Labs
Mark Logan, Mysten Labs
Ashok Menon, Mysten Labs
Todd Nowacki, Mysten Labs
Alberto Sonnino, Mysten Labs & University College London
Brandon Williams, Mysten Labs
Lu Zhang, Mysten Labs

Abstract

Sui Lutris is the first production-grade smart-contract platform that leverages consensusless agreement to achieve sub-second finality. Unlike prior work, Sui Lutris integrates seamlessly consensusless agreement with a high-throughput consensus protocol to not compromise expressiveness or throughput and is able to run perpetually without restarts. This feat is especially delicate during reconfiguration events, where the system needs to preserve the safety of the consensusless path without compromising the long-term liveness of potentially misconfigured clients. Sui Lutris combined with the Move programming language enables safe execution of smart-contracts that expose objects as a first-class resource.

1 Introduction

Traditional blockchains totally order transactions across replicated miners or validators to mitigate “double-spending” attacks, i.e., a user trying to use the same coin in two different transactions. It is well known that total ordering requires consensus. In recent years, however, systems based on consistent [4] and reliable [20] broadcasts have been proposed instead. These rely on objects (e.g., a coin) being controlled by a single authorization path (e.g., a single signer or a multi-sig mechanism), responsible for the liveness of transactions. This concept has been used to design asynchronous, and lightweight alternatives to traditional blockchains for decentralized payments [4, 5, 13]. We call these systems *consensus-less* as they do not require full consensus of atomic broadcast channels. Yet, so far they have not been used in a production blockchain.

On the one hand, consensus-based protocols allow for general-purpose smart contracts. But come at the cost of using more complex consensus protocols with higher latency. On the other hand, consensus-less protocols are simpler to implement and have low latency. But typically support a restricted set of operations, and deploying them in a dynamic environment is challenging as they do not readily support state checkpoints and validator reconfiguration. Supporting these functions is vital for the health of a long-lived production system.

We present Sui Lutris, a system that combines the consensus-less and consensus-based approaches to provide the best of both worlds when processing transactions in a replicated Byzantine setting. Sui Lutris uses a consistent broadcast protocol between validators to ensure the safety of common operations on assets owned by a single owner, ensuring lower latency as compared to consensus. It only relies on consensus for the safety of complex smart contracts operating on shared-ownership objects, as well as to support network maintenance operations such as defining checkpoints and reconfiguration. It is maintained by a permissionless set of validators that play the same role as miners in other blockchains.

Sui Lutris has been designed for and adopted as the core system behind the Sui blockchain. As of May 2, 2023, its latest testnet is operated by 97 geo-distributed heterogeneous validators and processes over 251 million certificates a day over 775 epoch changes using the Sui Lutris protocols. It stores over 810 million objects defined by over 86,000 Move packages. For this reason we present in the paper details that go beyond merely illustrating core components.

Challenges. Designing Sui Lutris requires tackling 3 key issues: Firstly, a high-throughput system such as Sui Lutris requires a checkpoint protocol in order to archive parts of its history and reduce the memory footprint and bootstrap cost of new participants. Checkpointing however is not as simple as in classic blockchains since we do not have total ordering guarantees for all transactions. Instead, Sui Lutris proposes an after-the-fact checkpointing protocol that eventually generates a cannonical sequence of transactions and certificates, without delaying execution and transaction finality.

Secondly, consensus-less protocols typically provide low latency at the cost of usability. A misconfigured client (e.g., underestimating the gas fee or crash-recovering) risks deadlocking its account. We consider this an unacceptable compromise for production systems. We develop Sui Lutris such that client bugs only affect the liveness of a single epoch, and provide rigorous proofs to support it.

Finally, the last challenge to solve is the dynamic participation of validators in a permissionless system. The lack of total ordering makes the solution non-trivial as different validators may stop processing transactions at different points compromising the liveness of the system. Additional challenges stem from the non-starvation

Conclusion

The Sui Lutris System

- Separate owned and shared objects
- **Only use consensus when you need to**
- Execute in parallel whenever you can

- **Paper:** https://sonnino.com/papers/sui-lutris.pdf
- **Code:** https://github.com/mystenlabs/sui

alberto@mystenlabs.com