The Sui Smart Contracts Platform

The MystenLabs Team
hello@mystenlabs.com

1 INTRODUCTION

Sui is a decentralized permissionless smart contract platform biased towards low-latency management of assets. It uses the Move programming language to define assets as objects that may be owned by an address. Move programs define operations on these typed objects including custom rules for their creation, the transfer of these assets to new owners, and operations that mutate assets.

Sui is maintained by a permissionless set of authorities that play a role similar to validators or miners in other blockchain systems. It uses a Byzantine consistent broadcast protocol between authorities to ensure the safety of common operations on assets, ensuring lower latency and better scalability as compared to Byzantine agreement. It only relies on Byzantine agreement for the safety of shared objects. As well as governance operations and check-pointing, performed off the critical latency path. Execution of smart contracts is also naturally parallelized when possible. Sui supports light clients that can authenticate reads as well as full clients that may audit all transitions for integrity. These facilities allow for trust-minimized bridges to other blockchains.

A native asset SUI is used to pay for gas for all operations. It is also used by its owners to delegate stake to authorities to operate Sui within epochs, and periodically, authorities are reconfigured according to the stake delegated to them. Used gas is distributed to authorities and their delegates according to their stake and their contribution to the operation of Sui.

This whitepaper is organized in two parts, with Sect. 2 describing the Sui programming model using the Move language, and Sect. 4 describing the operations of the permissionless decentralized system that ensures safety, liveness and performance for Sui.

2 SUI SMART CONTRACT PROGRAMMING

Sui smart contracts are written in the Move[4] language. Move is safe and expressive, and its type system and data model naturally support the parallel agreement/execution strategies that make Sui scalable. Move is an open-source programming language for building smart contracts originally developed at Facebook for the Diem blockchain. The language is platform-agnostic, and in addition to being adopted by Sui, it has been gaining popularity on other platforms (e.g., 0L, StarCoin).

In this section we will discuss the main features of the Move language and explain how it is used to create and manage assets on Sui. A more thorough explanation of Move's features can be found in the Move Programming Language book[1] and more Sui-specific Move content can be found in the Sui Developer Portal[2], and a more formal description of Move in the context of Sui can be found in Section 3.

2.1 Overview

Sui's global state includes a pool of programmable objects created and managed by Move *packages* that are collections of Move modules (see Section 2.1.1 for details) containing Move functions and types. Move packages themselves are also objects. Thus, Sui objects can be partitioned into two categories:

- **Struct data values**: Typed data governed by Move modules. Each object is a struct value with fields that can contain primitive types (e.g. integers, addresses), other objects, and non-object structs.
- **Package code values**: a set of related Move bytecode modules published as an atomic unit. Each module in a package can depend both on other modules in that package and on modules in previously published packages.

Objects can encode assets (e.g., fungible or non-fungible tokens), *capabilities* granting the permission to call certain functions or create other objects, "smart contracts" that manage other assets, and so on–it's up to the programmer to decide. The Move code to declare a custom Sui object type looks like this:

```
struct Obj has key {
  id: VersionedID, // globally unique ID and version
  f: u64 // objects can have primitive fields
  g: OtherObj // fields can also store other objects
}
```

All structs representing Sui objects (but not all Move struct values) must have the `id` field and the `key` ability[3] indicating that the value can be stored in Sui's global object pool.

2.1.1 Modules. A Move program is organized as a set of modules, each consisting of a list of struct declarations and function declarations. A module can import struct types from other modules and invoke functions declared by other modules.

Values declared in one Move module can flow into another–e.g., module `OtherObj` in the example above could be defined in a different module than the module defining `Obj`. This is different from most smart contract languages, which allow only unstructured bytes to flow across contract boundaries. However, Move is able to support this because it provides encapsulation features to help programmers write *robustly safe* [14] code. Specifically, Move's type system ensures that a type like `Obj` above can only be created, destroyed, copied, read, and written by functions inside the module that declares the type. This allows a module to enforce strong invariants on its declared types that continue to hold even when they flow across smart contract trust boundaries.

2.1.2 Transactions and Entrypoints. The global object pool is updated via transactions that can create, destroy, read, and write objects. A transaction must take each existing object it wishes to operate on as an input. In addition, a transaction must include the

[1] https://diem.github.io/move/
[2] https://github.com/MystenLabs/fastnft/blob/main/doc/SUMMARY.md
[3] https://diem.github.io/move/abilities.html

versioned ID of a package object, the name of a module and function inside that package, and arguments to the function (including input objects). For example, to call the function

```
public fun entrypoint(
  o1: Obj, o2: &mut Obj, o3: &Obj, x: u64, ctx: &mut TxContext
) { ... }
```

a transaction must supply ID's for three distinct objects whose type is `Obj` and an integer to bind to `x`. The `TxContext` is a special parameter filled in by the runtime that contains the sender address and information required to create new objects.

Inputs to an entrypoint (and more generally, to any Move function) can be passed with different mutability permissions encoded in the type. An `Obj` input can be read, written, transferred, or destroyed. A `&mut Obj` input can only be read or written, and a `&Obj` can only be read. The transaction sender must be authorized to use each of the input objects with the specified mutability permissions–see Section 4.4 for more detail.

2.1.3 Creating and Transferring Objects. Programmers can create objects by using the `TxContext` passed into the entrypoint to generate a fresh ID for the object:

```
public fun create_then_transfer(
  f: u64, g: OtherObj, o1: Obj, ctx: &mut TxContext
) {
  let o2 = Obj { id: TxContext::fresh_id(ctx), f, g };
  Transfer::transfer(o1, TxContext:sender());
  Transfer::transfer(o2, TxContext:sender());
}
```

This code takes two objects of type `OtherObj` and `Obj` as input, uses the first one and the generated ID to create a new `Obj`, and then transfers both `Obj` objects to the transaction sender. Once an object has been transferred, it flows into the global object pool and cannot be accessed by code in the remainder of the transaction. The `Transfer` module is part of the `Sui` standard library, which includes functions for transferring objects to user addresses and to other objects.

We note that if the programmer code neglected to include one of the `transfer` calls, this code would be rejected by the Move type system. Move enforces *resource safety* [5] protections to ensure that objects cannot be created without permission, copied, or accidentally destroyed. Another example of resource safety would be an attempt to transfer the same object twice, which would also be rejected by the Move type system.

3 THE SUI PROGRAMMING MODEL

In this section, we expand on the informal description of the Sui programming model from Section 2 by presenting detailed semantic definitions. The previous section showed examples of Move source code; here we define the structure of Move bytecode. Developers write, test, and formally verify [10, 16] Move source code locally, then compile it to Move bytecode before publishing it to the blockchain. Any Move bytecode be published on-chain must pass through a *bytecode verifier*[4, 5] to ensure that it satisfies key properties such as type, memory, and resource safety.

As mentioned in Section 2, Move is a platform-agnostic language which can be adapted to fit specific needs of different systems without forking the core language. In the following description, we define both concepts from core Move language (denoted in black text) and Sui-specific features extending the core Move language (denoted with orange text).

3.1 Modules

Module =	ModuleName×
	(StructName ⇀ StructDecl)×
	(FunName ⇀ FunDecl) × FunDecl
GenericParam =	[Ability]
StructDecl =	(FieldName ⇀ StorableType)×
	[Ability] × [GenericParam]
FunDecl =	[Type][Type] × [Instr] × [GenericParam]
Instr =	TransferToAddr | TransferToObj | ShareMut |
	ShareImmut | ...

Table 1: Module

Move code is organized into *modules* whose structure is defined in Table 1. A module consists of a collection of named *struct* declarations and a collection of named *function* declarations (examples of these declaration are provided in Section 2.1). A module also contains a special function declaration serving as the module *initializer*. This function is invoked exactly once at the time the module is published on-chain.

A struct declaration is a collection of named fields, where a field name is mapped to a storeable type. Its declaration also includes an optional list of abilities (see Section 2 for a description of storeable types and abilities). A struct declaration may also include a list of *generic parameters* with ability constraints, in which case we call it a *generic struct* declaration, for example `struct Wrapper<T: copy>{ t: T }`. A generic parameter represents a type to be used when declaring struct fields – it is unknown at the time of struct declaration, with a *concrete* type provided when the struct is instantiated (i.e., as struct value is created).

A function declaration includes a list of parameter types, a list of return types, and a list of instructions forming the function's body. A function declaration may also include a list of generic parameters with ability constraints, in which case we call it a *generic function* declaration, for example `fun unwrap<T: copy>(p: Wrapper<T>){}`. Similarly to struct declarations, a generic parameter represents a type unknown at function declaration time, but which is nevertheless used when declaring function parameters, return values and a function body (concrete type is provided when a function is called).

Instructions that can appear in a function body include all ordinary Move instructions with the exception of global storage instructions (e.g., `move_to`, `move_from`, `borrow_global`). See [14] for a complete list of core Move's instructions and their semantics. In Sui persistent storage is supported via Sui's global object pool rather than the account-based global storage of core Move.

There are four Sui-specific object operations. Each of these operations changes the ownership metadata of the object (see Section 3.3) and returns it to the global object pool. Most simply, a Sui object can be transferred to the address of a Sui end-user. An object can also be transferred to another *parent* object–this operation requires the caller to supply a mutable reference to the parent object in

addition to the child object. An object can be mutably *shared* so it can be read/written by anyone in the Sui system. Finally, an object can be immutably shared so it can be read by anyone in the Sui system, but not written by anyone.

The ability to distinguish between different kinds of ownership is a unique feature of Sui. In other blockchain platforms we are aware of, every contract and object is mutably shared. As we will explain in Section 4, Sui leverages this information for parallel transaction execution (for all transactions) and parallel agreement (for transactions involving objects without shared mutability).

3.2 Types and Abilities

PrimType =	{`address`, `id`, `bool`, `u8`, `u64`, ...}
StructType =	ModuleName × StructName× [StorableType]
StorableType =	PrimType ⊎ StructType⊎ GenericType ⊎ VectorType
VectorType =	StorableType
GenericType =	ℕ
MutabilityQual =	{`mut`, `immut`}
ReferenceType =	StorableType × MutabilityQual
Type =	ReferenceType ⊎ StorableType
Ability =	{`key`, `store`, `copy`, `drop`}

Table 2: Types and Abilities

A Move program manipulates both data stored in Sui global object pool and transient data created when the Move program executes. Both objects and transient data are Move *values* at the language level. However, not all values are created equal – they may have different properties and different structure as prescribed by their types.

The types used in Move are defined in Table 2. Move supports many of the same *primitive types* supported in other programming languages, such as a boolean type or unsigned integer types of various sizes. In addition, core Move has an `address` type representing an end-user in the system that is also used to identify the sender of a transaction and (in Sui) the owner of an object. Finally, Sui defines an `id` type representing an identity of a Sui object– see Section 3.3 for details.

A *struct type* describes an instance (i.e., a value) of a struct declared in a given module (see Section 3.1 for information on struct declarations). A struct type representing a generic struct declaration (i.e., *generic struct* type) includes a list of *storeable types* – this list is the counterpart of the generic parameter list in the struct declaration. A storeable type can be either a *concrete type* (a primitive or a struct) or a *generic type*. We call such types storeable because they can appear as fields of structs and in objects stored persistently on-chain, whereas reference types cannot.

For example, the `Wrapper<u64>` struct type is a generic struct type parameterized with a concrete (primitive) storeable type `u64` – this kind of type can be used to create a struct instance (i.e.,value). On the other hand, the same generic struct type can be parameterized with a generic type (e.g., `struct Parent<T> { w: Wrapper<T> }`) coming from a generic parameter of the enclosing struct or function declaration – this kind of type can be used to declare struct fields, function params, etc. Structurally, a generic type is an integer index (defined as ℕ in Table 5) into the list of generic parameters in the enclosing struct or function declaration.

A *vector type* in Move describes a variable length collection of homogenous values. A Move vector can only contain storeable types, and it is also a storeable type itself.

A Move program can operate directly on values or access them indirectly via references. A *reference type* includes both the storeable type referenced and a *mutability qualifier* used to determine (and enforce) whether a value of a given type can be read and written (`mut`) or only read (`immut`). Consequently, the most general form of a Move value type (`Type` in Table 2) can be either a storeable type or a reference type.

Finally, *abilities* in Move control what actions are permissible for values of a given type, such as whether a value of a given type can be copied (duplicated). Abilities constraint struct declarations and generic type parameters. The Move bytecode verifier is responsible for ensuring that sensitive operations like copies can only be performed on types with the corresponding ability.

3.3 Objects and Ownership

TxDigest =	Com(Tx)
ObjID =	Com(TxDigest × ℕ)
SingleOwner =	Addr ⊎ ObjID
Shared =	{`shared_mut`, `shared_immut`}
Ownership =	SingleOwner ⊎ Shared
StructObj =	StructType × Struct
ObjContents =	StructObj ⊎ Package
Obj =	ObjContents × ObjID × Ownership × Version

Table 3: Objects and Ownership

Each Sui object has a globally unique identifier (ObjID in Table 3) that serves as the persistent identity of the object as it flows between owners and into and out of other objects. This ID is assigned to the object by the transaction that creates it. An object ID is created by applying a collision-resistant hash function to the contents of the current transaction and to a counter recording how many objects the transaction has created. A transaction (and thus its digest) is guaranteed to be unique due to constraints on the input objects of the transaction, as we will explain subsequently.

In addition to an ID, each object carries metadata about its ownership. An object is either uniquely owned by an address or another object, shared with write/read permissions, or shared with only read permissions. The ownership of an object determines whether and how a transaction can use it as an input. Broadly, a uniquely owned object can only be used in a transaction initiated by its owner or including its parent object as an input, whereas a shared object can be used by any transaction, but only with the specified mutability permissions. See Section 4.4 for a full explanation.

There are two types of objects: package code objects, and struct data objects. A package object contains of a list of modules. A struct object contains a Move struct value and the Move type of that value.

The contents of an object may change, but its ID, object type (package vs struct) and Move struct type are immutable. This ensures that objects are strongly typed and have a persistent identity.

Finally, an object contains a version. Freshly created objects have version 0, and an object's version is incremented each time a transaction takes the object as an input.

3.4 Addresses and Authenticators

Authenticator =	Ed25519PubKey ⊎ ECDSAPubKey ⊎ ...
Addr =	*Com*(Authenticator)

Table 4: Addresses and Authenticators

An address is the persistent identity of a Sui end-user (although note that a single user can have an arbitrary number of addresses). To transfer an object to another user, the sender must know the address of the recipient.

As we will discuss shortly, a Sui transaction must contain the address of the user sending (i.e., initiating) the transaction and an *authenticator* whose digest matches the address. The separation between addresses and authenticators enables *cryptographic agility*. An authenticator can be a public key from any signature scheme, even if the schemes use different key lengths (e.g., to support post-quantum signatures). In addition, an authenticator need not be a single public key–it could also be (e.g.) a K-of-N multisig key.

3.5 Transactions

ObjRef =	ObjID × Version × *Com*(Obj)
CallTarget =	ObjRef × ModuleName × FunName
CallArg =	ObjRef ⊎ ObjID ⊎ PrimType
Package =	[Module]
Publish =	Package × [ObjRef]
Call =	CallTarget × [StorableType] × [CallArg]
GasInfo =	ObjRef × MaxGas × BaseFee × Tip
Tx =	(Call ⊎ Publish) × GasInfo × Addr × Authenticator

Table 5: Transactions

Sui has two different transaction types: publishing a new Move package, and calling a previously published Move package. A publish transaction contains a *package*–a set of modules that will be published together as a single object, as well as the dependencies of all the modules in this package (encoded as a list of object references that must refer to already-published package objects). To execute a publish transaction, the Sui runtime will run the Move bytecode verifier on each package, link the package against its dependencies, and run the module initializer of each module. Module initializers are useful for bootstrapping the initial state of an application implemented by the package.

A call transaction's most important arguments are object inputs. Object arguments are either specified via an object reference (for single-owner and shared immutable objects) or an object ID (for shared mutable objects). An object reference consists of an object ID, an object version, and the hash of the object value. The Sui runtime will resolve both object ID's and object references to object values stored in the global object pool. For object references, the runtime will check the version of the reference against the version of the object in the pool, as well as checking that the reference's hash matches the pool object. This ensures that the runtime's view of the object matches the transaction sender's view of the object.

In addition, a call transaction accepts type arguments and pure value arguments. Type arguments instantiate generic type parameters of the entrypoint function to be invoked (e.g., if the entrypoint function is `send_coin<T>(c: Coin<T>, ...)`, the generic type parameter `T` could be instantiated with the type argument `SUI` to send the Sui native token). Pure values can include primitive types and vectors of primitive types, but not struct types.

The function to be invoked by the call is specified via an object reference (which must refer to a package object), a name of a module in that package, and a name of a function in that package. To execute a call transaction, the Sui runtime will resolve the function, bind the type, object, and value arguments to the function parameters, and use the Move VM to execute the function.

Both call and publish transactions are subject to gas metering and gas fees. The metering limit is expressed by a maximum gas budget. The runtime will execute the transaction until the budget is reached, and will abort with no effects (other than deducting fees and reporting the abort code) if the budget is exhausted.

The fees are deducted from a *gas object* specified as an object reference. This object must be a Sui native token (i.e., its type must be `Coin<SUI>`). Sui uses EIP1559[4]-style fees: the protocol defines a base fee (denominated in gas units per Sui token) that is algorithmically adjusted at epoch boundaries, and the transaction sender can also include an optional tip (denominated in Sui tokens). Under normal system load, transactions will be processed promptly even with no tip. However, if the system is congested, transactions with a larger tip will be prioritized. The total fee deduced from the gas object is (GasUsed ∗ BaseFee) + Tip.

3.6 Transaction Effects

Event =	StructType × Struct
Create =	Obj
Update =	Obj
Wrap =	ObjID × Version
Delete =	ObjID × Version
ObjEffect =	Create ⊎ Update ⊎ Wrap ⊎ Delete
AbortCode =	ℕ × ModuleName
SuccessEffects =	[ObjEffect] × [Event]
AbortEffects =	AbortCode
TxEffects =	SuccessEffects ⊎ AbortEffects

Table 6: Transaction Effects

Transaction execution generates transaction effects which are different in the case when execution of a transaction is successful (`SuccessEffects` in Table 6) and when it is not (`AbortEffects` in Table 6).

[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md

Upon successful transaction execution, transaction effects include information about changes made to Sui's global object pool (including both updates to existing objects and freshly created objects) and *events* generated during transaction execution. Another effect of successful transaction execution could be object removal (i.e., deletion) from the global pool and also wrapping (i.e., embedding) one object into another, which has a similar effect to removal – a wrapped object disappears from the global pool and exists only as a part of the object that wraps it. Since deleted and wrapped objects are no longer accessible in the global pool, these effects are represented by the ID and version of the object.

Events encode side effects of successful transaction execution beyond updates to the global object pool. Structurally, an event consists of a Move struct and its type. Events are intended to be consumed by actors outside the blockchain, but cannot be read by Move programs.

Transactions in Move have an all-or-nothing semantics – if execution of a transaction aborts at some point (e.g., due to an unexpected failure), even if some changes to objects had happened (or some events had been generated) prior to this point, none of these effects persist in an aborted transaction. Instead, an aborted transaction effect includes a numeric abort code and the name of a module where the transaction abort occurred. Gas fees are still charged for aborted transactions.

4 THE SUI SYSTEM

In this section we describe Sui from a systems' perspective, including the mechanisms to ensure safety and liveness across authorities despite Byzantine failures. We also explain the operation of clients, including light clients that need some assurance about the system state without validating its full state.

Brief background. At a systems level Sui is an evolution of the FastPay [3] low-latency settlement system, extended to operate on arbitrary objects through user-defined smart contracts, and with a permissionless delegated proof of stake committee composition [2]. Basic asset management by object owners is based on a variant of Byzantine consistent broadcast [6] that has lower latency and is easier to scale across many machines as compared to traditional implementations of Byzantine consensus [8, 11, 12].When full agreement is required we use a high-throughput DAG-based consensus, e.g. [9] to manage locks, while execution on different shared objects is parallelized.

Protocol outline. Figure 1 illustrates the high-level interactions between a client and Sui authorities to commit a transaction. We describe them here briefly:

- A user with a private signing key creates and signs a *user transaction* to mutate objects they own, or shared objects, within Sui. Subsequently, user signature keys are not needed, and the remaining of the process may be performed by the user client, or a gateway on behalf of the user (denoted as *keyless operation* in the diagram).
- The user transaction is sent to the Sui authorities, that each check it for validity, and upon success sign it and return the *signed transaction* to the client. The client collects the responses from a quorum of authorities to form a *transaction certificate*.
- The transaction certificate is then sent back to all authorities, and if the transaction involves shared objects it is also sent to a *Byzantine agreement protocol* operated by the Sui authorities. Authorities check the certificate, and in case shared objects are involved also wait for the agreement protocol to sequence it in relation to other shared object transactions, and then execute the transaction and summarize its effects into a *signed effects* response.
- Once a quorum of authorities has executed the certificate its effects are final (denoted as *finality* in the diagram). Clients can collect a quorum of authority responses and create an *effects certificate* and use it as a proof of the finality of the transactions effects.

This section describes each of these operations in detail, as well as operations to reconfigure and manage state across authorities.

4.1 System Model

Sui operates in a sequence of epochs denoted by $e \in \{0, \ldots\}$. Each epoch is managed by a committee $C_e = (V_e, S_e(\cdot))$, where V_e is a set of authorities with known public verification keys and network end-points. The function $S_e(v)$ maps each authority $v \in V_e$ to a number of units of delegated stake. We assume that C_e for each epoch is signed by a quorum (see below) of authority stake at epoch $e-1$. (Sect. 4.7 discusses the formation and management of committees). Within an epoch, some authorities are correct (they follow the protocol faithfully and are live), while others are Byzantine (they deviate arbitrarily from the protocol). The security assumption is that the set of honest authorities $H_e \subseteq V_e$ is assigned a *quorum* of stake within the epoch, i.e. $\sum_{h \in H_e} S_e(h) > 2/3 \sum_{v \in V_e} S_e(v)$ (and refer to any set of authorities with over two-thirds stake as a quorum).

There exists at least one live and correct party that acts as a *relay* for each certificate (see Sect. 4.3) between honest authorities. This ensures liveness, and provides an eventual delivery property to the Byzantine broadcast (see totality of reliable broadcast in [6]). Each authority operates such a relay, either individually or through a collective dissemination protocol. External entities, including Sui light clients, replicas and services may also take on this role. The distinction between the passive authority core, and an internal or external active relay component that is less reliable or trusted, ensures a clear demarcation and minimization of the Trusted Computing Base [15] on which Sui's safety and liveness relies.

4.2 Authority & Replica Data Structures

Sui authorities rely on a number of data structures to represent state. We define these structures based on the operations they support. They all have a deterministic byte representation.

An *Object* (Obj) stores user smart contracts and data within Sui. They are the Sui system-level encoding of the Move objects introduced in Sect. 2. They support the following set of operations:

- *ref*(Obj) returns the *reference* (ObjRef) of the object, namely a triplet (ObjID, Version, ObjDigest). ObjID is practically

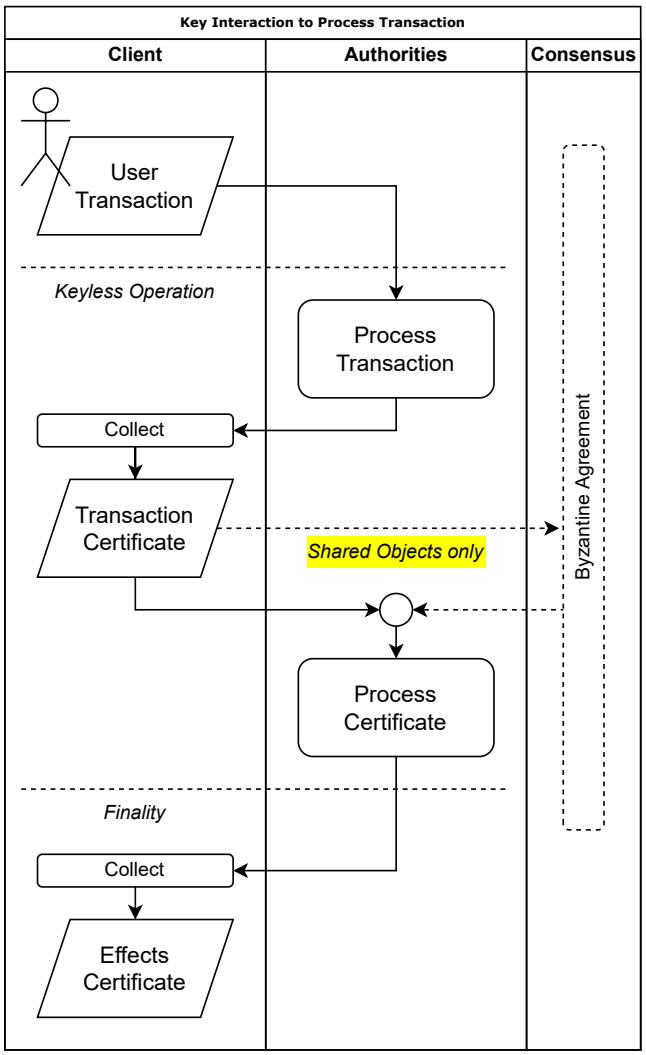

Figure 1: Outline of interactions to commit a transaction.

unique for all new objects created, and Version is an increasing positive integer representing the object version as it is being mutated.

- *owner*(Obj) returns the authenticator Auth of the owner of the object. In the simplest case, Auth is an address, representing a public key that may use this object. More complex authenticators are also available (see Sect. 4.4).
- *read-only*(Obj) returns true if the object is read-only. Read-only objects may never be mutated, wrapped or deleted. They may also be used by anyone, not just their owners.
- *parent*(Obj) returns the transaction digest (TxDigest) that last mutated or created the object.
- *contents*(Obj) returns the object type Type and data Data that can be used to check the validity of transactions and carry the application-specific information of the object.

The object reference (ObjRef) is used to index objects. It is also used to authenticate objects since ObjDigest is a commitment to their full contents.

A *transaction* (Tx) is a structure representing a state transition for one or more objects. They support the following set of operations:

- *digest*(Tx) returns the TxDigest, which is a binding cryptographic commitment to the transaction.
- *epoch*(Tx) returns the EpochID during which this transaction may be executed.
- *inputs*(Tx) returns a sequence of object [ObjRef] the transaction needs to execute.
- *payment*(Tx) returns a reference to an ObjRef to be used to pay for gas, as well as the maximum gas limit, and a conversion rate between a unit of gas and the unit of value in the gas payment object.
- *valid*(Tx, [Obj]) returns true if the transaction is valid, given the requested input objects provided. Validity is discussed in Sect. 4.4, and relates to the transactions being authorized to act on the input objects, as well as sufficient gas being available to cover the costs of its execution.
- *exec*(Tx, [Obj]) executes the transaction and returns a structure Effects representing its effects. A valid transaction execution is infallible, and its output is deterministic.

A transaction is indexed by its TxDigest, which may also be used to authenticate its full contents. All valid transactions (except the special hard-coded genesis transaction) have at least one owned input, namely the objects used to pay for gas.

A *transaction effects* (Effects) structure summarizes the outcome of a transaction execution. It supports the following operations:

- *digest*(Effects) is a commitment EffDigest to the Effects structure, that may be used to index or authenticate it.
- *transaction*(Effects) returns the TxDigest of the executed transaction yielding the effects.
- *dependencies*(Effects) returns a sequence of dependencies [TxDigest] that should be executed before the transaction with these effects may execute.
- *contents*(Effects) returns a summary of the execution. Status reports the outcome of the smart contract execution. The lists Created, Mutated, Wrapped, Unwrapped and Deleted, list the object references that underwent the respective operations. And Events lists the events emitted by the execution.

A *transaction certificate* TxCert on a transaction contains the transaction itself as well as the identifiers and signatures from a quorum of authorities. Note that a certificate may not be unique, in that the same logical certificate may be represented by a different set of authorities forming a quorum. Additionally, a certificate might not strictly be signed by exactly a 2/3 quorum, but possibly more if more authorities are responsive. However, two different valid certificates on the same transaction should be treated as representing semantically the same certificate. A *partial certificate* (TxSign) contains the same information, but signatures from a set of authorities representing stake lower than the required quorum, usually a single authority. The identifiers of signers are included in the certificate (i.e., accountable signatures [**?**]) to identify authorities

ready to process the certificate, or that may be used to download past information required to process the certificate (see Sect. 4.8).

Similarly, an *effects certificate* EffCert on an effects structure contains the effects structure itself, and signatures from authorities[5] that represent a quorum for the epoch in which the transaction is valid. The same caveats, about non-uniqueness and identity apply as for transaction certificates. A partial effects certificate, usually containing a single authority signature and the effects structure is denoted as EffSign.

Persistent Stores. Each authority and replica maintains a set of persistent stores. The stores implement persistent map semantics and can be represented as a set of key-value pairs (denoted $map[key] \rightarrow value$), such that only one pair has a given key. Before a pair is inserted a contains(key) call returns false, and get(key) returns an error. After a pair is inserted contains(key) calls returns true, and get(key) return the value. An authority maintains the following persistent stores:

- The *order lock map* Lock_v[ObjRef] → TxSignOption records the first valid transaction Tx seen and signed by the authority for an owned object version ObjRef, or None if the object version exists but no valid transaction using as an input it has been seen. It may also record the first certificate seen with this object as an input. This table, and its update rules, represents the state of the distributed locks on objects across Sui authorities, and ensures safety under concurrent processing of transactions.
- The *certificate map* Ct_v[TxDigest] → (TxCert, EffSign) records all full certificates TxCert, which also includes Tx, processed by the authority within their validity epoch, along with their signed effects EffSign. They are indexed by transaction digest TxDigest
- The *object map* Obj_v[ObjRef] → Obj records all objects Obj created by transactions included in certificates within Ct_v indexed by ObjRef. This store can be completely derived by re-executing all certificates in Ct_v. A secondary index is maintained that maps ObjID to the latest object with this ID. This is the only information necessary to process new transactions, and older versions are only maintained to facilitate reads and audit.
- The *synchronization map* Sync_v[ObjRef] → TxDigest indexes all certificates within Ct_v by the objects they create, mutate or delete as tuples ObjRef. This structure can be fully re-created by processing all certificates in Ct_v, and is used to help client synchronize transactions affecting objects they care about.

Authorities maintain all four structures, and also provide access to local checkpoints of their certificate map to allow other authorities and replicas to download their full set of processed certificates. A replica does not process transactions but only certificates, and re-executes them to update the other tables as authorities do. It also maintains an order lock map to audit non-equivocation.

[5]Note that if the signature algorithm permits it, authority signatures can be compressed, but always using accountable signature aggregation, because tracking who signed is important for gas profit distribution and other network health measurements.

An authority may be designed as a full replica maintaining all four stores (and checkpoints) to facilitate reads and synchronization, combined with a minimal authority core that only maintains object locks and objects for the latest version of objects used to process new transactions and certificates. This minimizes the Trusted Computing Base relied upon for safety.

Only the order lock map requires strong key self-consistency, namely a read on a key should always return whether a value or None is present for a key that exists, and such a check should be atomic with an update that sets a lock to a non-None value. This is a weaker property than strong consistency across keys, and allows for efficient sharding of the store for scaling. The other stores may be eventually consistent without affecting safety.

4.3 Authority Base Operation

Process Transaction. Upon receiving a transaction Tx an authority performs a number of checks:

(1) It ensures *epoch*(Tx) is the current epoch.
(2) It ensures all object references *inputs*(Tx) and the gas object reference in *payment*(Tx) exist within Obj_v and loads them into [Obj]. For owned objects the exact reference should be available; for read-only or shared objects the object ID should exist.
(3) Ensures sufficient gas can be made available in the gas object to cover the cost of executing the transaction.
(4) It checks *valid*(Tx, [Obj]) is true. This step ensures the authentication information in the transaction allows access to the owned objects.
(5) It checks that Lock_v[ObjRef] for all owned *inputs*(Tx) objects exist, and it is either None or set to *the same* Tx, and atomically sets it to TxSign. (We call these the 'locks checks').

If any of the checks fail processing ends, and an error is returned. However, it is safe for a partial update of Lock_v to persist (although our current implementation does not do partial updates, but atomic updates of all locks).

If all checks are successful then the authority returns a signature on the transaction, ie. a partial certificate TxSign. Processing an order is idempotent upon success, and returns a partial certificate (TxSign), or a full certificate (TxCert) if one is available.

Any party may collate a transaction and signatures (TxSign) for a set of authorities forming a quorum for epoch e, to form a transaction certificate TxCert.

Process Certificate. Upon receiving a certificate an authority checks all validity conditions for the transaction, except those relating to locks (the so-called 'locks checks'). Instead it performs the following checks: for each *owned* input object in *inputs*(Tx) it checks that the lock exists, and that it is either None, set to *any* TxSign, or set to a certificate for the same transaction as the current certificate. If this modified locks check fails, the authority has detected an unrecoverable Byzantine failure, halts normal operations, and starts a disaster recovery process. For *shared objects* (see Sect. 4.4) authorities check that the locks have been set through the certificate being sequenced in a consensus, to determine the

version of the share object to use. If so, the transaction may be executed; otherwise it needs to wait for such sequencing first.

If the check succeeds, the authority adds the certificate to its certificate map, along with the effects resulting from its execution, ie. $\mathsf{Ct}_v[\mathsf{TxDigest}] \rightarrow (\mathsf{TxCert}, \mathsf{EffSign})$; it updates the locks map to record the certificate $\mathsf{Lock}_v[\mathsf{ObjRef}] \rightarrow \mathsf{TxCert}$ for all owned input objects that have locks not set to a certificate. As soon as all objects in $\mathsf{Input}(\mathsf{Tx})$ is inserted in Obj_v, then all effects in EffSign are also materialized by adding their ObjRef and contents to Obj_v. Finally for all created or mutated in EffSign the synchronization map is updated to map them to Tx.

Remarks. The logic for handling transactions and certificates leads to a number of important properties:

- **Causality & parallelism.** The processing conditions for both transactions and certificates ensure causal execution: an authority only 'votes' by signing a transaction if it has processed all certificates creating the objects the transaction depends upon, both owned, shared and read-only. Similarly, an authority only processes a certificate if all input objects upon which it depends exist in its local objects map. This imposes a causal execution order, but also enables transactions not causally dependent on each other to be executed in parallel on different cores or machines.
- **Sign once, and safety.** All owned input objects locks in $\mathsf{Lock}_v[\cdot]$ are set to the first transaction Tx that passes the checks using them, and then the first certificate that uses the object as an input. We call this *locking the object to this transaction*, and there is no unlocking within an epoch. As a result an authority only signs a single transaction per lock, which is an essential component of consistent broadcast [6], and thus the safety of Sui.
- **Disaster recovery.** An authority detecting two contradictory certificates for the same lock, has proof of irrecoverable Byzantine behaviour – namely proof that the quorum honest authority assumption does not hold. The two contradictory certificates are a fraud proof [1], that may be shared with all authorities and replicas to trigger disaster recovery processes. Authorities may also get other forms of proof of unrecoverable byzantine behaviour such as >1/3 signatures on effects (EffSign) that represent an incorrect execution of a certificate. Or a certificate with input objects that do not represent the correct outputs of previously processed certificates. These also can be packaged as a fraud proof and shared with all authorities and replicas. Note these are distinct from proofs that a tolerable minority of authorities ($\leq 1/3$ by stake) or object owners (any number) is byzantine or equivocating, which can be tolerated without any service interruption.
- **Finality.** Authorities return a certificate (TxCert) and the signed effects (EffSign) for any read requests for an index in Lock_v, Ct_v and Obj_v, Sync_v. A transaction is considered final if over a quorum of authorities reports Tx as included in their Ct_v store. This means that an effects certificate (EffCert) is a transferable proof of finality. However, a certificate using an object is also proof that all dependent certificates in its causal path are also final. Providing a certificate to any party, that may then submit it to a super majority of authorities for processing also ensures finality for the effects of the certificate. Note that finality is later than fastpay [3] to ensure safety under re-configuration. However, an authority can apply the effect of a transaction upon seeing a certificate rather than waiting for a commit.

4.4 Owners, Authorization, and Shared Objects

Transaction validity (see Sect. 4.3) ensures a transaction is authorized to include all specified input objects in a transaction. This check depends on the nature of the object, as well as the owner field.

Read-only objects cannot be mutated or deleted, and can be used in transactions concurrently and by all users. Move modules for example are read-only. Such objects do have an owner that might be used as part of the smart contract, but that does not affect authorization to use them. They can be included in any transaction.

Owned objects have an owner field. The owner can be set to an address representing a public key. In that case, a transaction is authorized to use the object, and mutate it, if it is signed by that address. A transaction is signed by a single address, and therefore can use one or more objects owned by that address. However, a single transaction cannot use objects owned by more than one address. The owner of an object, called a child object, can be set to the ObjID of another object, called the parent object. In that case the child object may only be used if the parent object is included in the transaction, and the transaction is authorized to use the object. This facility may be used by contracts to construct efficient collections and other complex data structures.

Shared objects are mutable, but do not have a specific owner. They can instead be included in transactions by different parties, and do not require any authorization. Instead they perform their own authorization logic. Such objects, by virtue of having to support multiple writers while ensuring safety and liveness, do require a full agreement protocol to be used safely. Therefore they require additional logic before execution. Authorities process transactions as specified in Sect. 4.3 for owned objects and read-only objects to manage their locks. However, authorities do not rely on consistent broadcast to manage the locks of shared objects. Instead, the creators of transactions that involve shared objects insert the certificate on the transaction into a high-throughput consensus system, e.g. [9]. All authorities observe a consistent sequence of such certificates, and assign the version of shared objects used by each transaction according to this sequence. Then execution can proceed and is guaranteed to be consistent across all authorities. Authorities include the version of shared objects used in a transaction execution within the Effects certificate.

The above rules ensure that execution for transactions involving read-only and owned objects requires only consistent broadcast and a single certificate to proceed; and Byzantine agreement is only required for transactions involving shared objects. Smart contract authors can therefore design their types and their operations to optimize transfers and other operations on objects of a single user to have lower latency, while enjoying the flexibility of using shared

objects to implement logic that needs to be accessed by multiple users.

4.5 Clients

Full Clients & Replicas. Replicas, also sometimes called *full clients*, do not validate new transactions, but maintain a consistent copy of the valid state of the system for the purposes of audit, as well as to construct transactions or operate services incl. read infrastructures for light client queries.

Light Clients. Both object references and transactions contain information that allows the authentication of the full causal chain of transactions that leading up to their creation or execution. Specifically, an object reference (ObjRef) contains an ObjDigest that is an authenticator for the full state of the object, including the facility to get *parent*(Obj), namely the TxDigest that created the object. Similarly, a TxDigest authenticates a transaction, including the facility to extract through *inputs*(Tx) the object references of the input objects. Therefore the set of objects and certificates form a bipartite graph that is self-authenticating. Furthermore, effects structures are also signed, and may be collated into effects certificates that directly certify the results of transaction executions.

These facilities may be used to support *light clients* that can perform high-integrity reads into the state of Sui, without maintaining a full replica node. Specifically an authority or full node may provide a succinct bundle of evidence, comprising a certificate TxCert on a transaction Tx and the input objects [Obj] corresponding to *inputs*(Tx) to convince a light client that a transition can take place within Sui. A light client may then submit this certificate, or check whether it has been seen by a quorum or sample of authorities to ensure finality. Or it may craft a transaction using the objects resulting from the execution, and observe whether it is successful.

More directly, a service may provide an effects certificate to a client to convince them of the existence and finality of a transition within Sui, with no further action or interaction within the system. If a checkpoint of finalized certificates is available, at an epoch boundary or otherwise, a bundle of evidence including the input objects and certificate, alongside a proof of inclusion of the certificate in the checkpoint is also a proof of finality.

Authorities may use a periodic checkpointing mechanism to create collective checkpoints of finalized transactions, as well as the state of Sui over time. A certificate with a quorum of stake over a checkpoint can be used by light clients to efficiently validate the recent state of objects and emitted events. A check pointing mechanism is necessary for committee reconfiguration between epochs. More frequent checkpoints are useful to light clients, and may also be used by authorities to compress their internal data structures as well as synchronize their state with other authorities more efficiently.

4.6 Bridges

Native support for light clients and shared objects managed by Byzantine agreement allows Sui to support two-way bridges to other blockchains [13]. The trust assumption of such bridges reflect the trust assumptions of Sui and the other blockchain, and do not have to rely on trusted oracles or hardware if the other blockchain also supports light clients [7].

Bridges are used to import an asset issued on another blockchain, to represent it and use it as a wrapped asset within the Sui system. Eventually, the wrapped asset can be unlocked and transferred back to a user on the native blockchain. Bridges can also allow assets issued on Sui to be locked, and used as wrapped assets on other blockchains. Eventually, the wrapped object on the other system can be destroyed, and the object on Sui updated to reflect any changes of state or ownership, and unlocked.

The semantics of bridged assets are of some importance to ensure wrapped assets are useful. Fungible assets bridged across blockchains can provide a richer wrapped representation that allows them to be divisible and transferable when wrapped. Non-fungible assets are not divisible, but only transferable. They may also support other operations that mutates their state in a controlled manner when wrapped, which may necessitate custom smart contract logic to be executed when they are bridged back and unwrapped. Sui is flexible and allows smart contract authors to define such experiences, since bridges are just smart contracts implemented in Move rather than native Sui concepts – and therefore can be extended using the composability and safety guarantees Move provides.

4.7 Committee Reconfiguration

Reconfiguration occurs between epochs when a committee C_e is replaced by a committee $C_{e'}$, where $e' = e + 1$. Reconfiguration safety ensures that if a transaction Tx was committed at e or before, no conflicting transaction can be committed after e. Liveness ensures that if Tx was committed at or before e, then it must also be committed after e.

We leverage the Sui smart contract system to perform a lot of the work necessary for reconfiguration. Within Sui a system smart contract allows users to lock and delegate stake to candidate authorities. During an epoch, owners of coins are free to delegate by locking tokens, undelegate by unlocking tokens or change their delegation to one or more authorities.

Once a quorum of stake for epoch e vote to end the epoch, authorities exchange information to commit to a checkpoint, determine the next committee, and change the epoch. First, authorities run a check pointing protocol, with the help of an agreement protocol [9], to agree on a certified checkpoint for the end of epoch e. The checkpoint contains the union of all transactions, and potentially resulting objects, that have been processed by a quorum of authorities. As a result if a transaction has been processed by a quorum of authorities, then at least one honest authorities that processed it will have its processed transactions included in the end-of-epoch checkpoint, ensuring the transaction and its effects are durable across epochs. Furthermore, such a certified checkpoint guarantees that all transactions are available to honest authorities of epoch e.

The stake delegation at the end-of-epoch checkpoint is then used to determine the new set of authorities for epoch $e + 1$. Both a quorum of the old authorities stake and a quorum of the new authority stake signs the new committee $C_{e'}$, and checkpoint at which the new epoch commences. Once both set of signatures are

available the new set of authorities start processing transactions for the new epoch, and old authorities may delete their epoch signing keys.

Recovery. It is possible due to client error or client equivocation for an owned object to become 'locked' within an epoch, preventing any transaction concerning it from being certified (or finalized). For example, a client signing two different transactions using the same owned object version, with half of authorities signing each, would be unable to form a certificate requiring a quorum of signatures on any of the two certificates. Recovery ensures that once epochs change such objects are again in a state that allows them to be used in transactions. Since, no certificate can be formed, the original object is available at the start of the next epoch to be operated on. Since transactions contain an epoch number, the old equivocating transactions will not lock the object again, giving its owner a chance to use it.

Rewards & cryptoeconomics. Sui has a native token SUI, with a fixed supply. SUI is used to pay for gas, and is also be used as delegated stake on authorities within an epoch. The voting power of authorities within this epoch is a function of this delegated stake. At the end of the epoch fees collected through all transactions processed are distributed to authorities according to their contribution to the operation of Sui, and in turn they share some of the fees as rewards to addresses that delegated stake to them. We postpone a full description of the token economics of Sui to a dedicated paper.

4.8 Authority & Replica Updating

Client-driven. Due to client failures or non-byzantine authority failures, some authorities may not have processed all certificates. As a result causally related transactions depending on missing objects generated by these certificates would be rejected. However, a client can always update an honest authority to the point where it is able to process a correct transaction. It may do this using its own store of past certificates, or using one or more other honest authorities as a source for past certificates.

Given a certificate c and a Ct_v store that includes c and its causal history, a client can update an honest authority v' to the point where c would also be applied. This involves finding the smallest set of certificates not in v' such that when applied the Objects in v' include all inputs of c. Updating a lagging authority B using a store Ct_v including the certificate TxCert involves:

- The client maintains a list of certificates to sync, initially set to contain just TxCert.
- The client considers the last TxCert requiring sync. It extracts the Tx within the TxCert and derives all its input objects (using Input(Tx)).
- For each input object it checks whether the Tx that generated or mutated last (using the Sync_v index on Ct_v) has a certificate within B, otherwise its certificate is read from Ct_v and added to the list of certificates to sync.
- If no more certificates can be added to the list (because no more inputs are missing from B) the certificate list is sorted in a causal order and submitted to B.

The algorithm above also applies to updating an object to a specific version to enable a new transaction. In this case the certificate for the Tx that generated the object version, found in $\text{Sync}_v[\text{ObjRef}]$, is submitted to the lagging authority. Once it is executed on B the object at the correct version will become available to use.

A client performing this operation is called a *relayer*. There can be multiple relayers operating independently and concurrently. They are untrusted in terms of integrity, and their operation is keyless. Besides clients, authorities can run the relayer logic to update each other, and replicas operating services can also act as relayers to update lagging authorities.

Bulk. Authorities provide facilities for a follower to receive updates when they process a certificate. This allows replicas to maintain an up-to-date view of an authorty's state. Furthermore, authorities may use a push-pull gossip network to update each other of the latest processed transaction in the short term and to reduce the need for relayers to perform this function. In the longer term lagging authorities may use periodic state commitments, at epoch boundaries or more frequently, to ensure they have processed a complete set of certificates up to certain check points.

5 SCALING AND LATENCY

The Sui system allows scaling though authorities devoting more resources, namely CPUs, memory, network and storage within a machine or over multiple machines, to the processing of transactions. More resources lead to an increased ability to process transactions, leading to increased fees income to fund these resources. More resources also results in lower latency, as operations are performed without waiting for necessary resources to become available.

Throughput. To ensure that more resources result in increased capacity quasi-linearly, the Sui design aggressively reduces bottlenecks and points of synchronization requiring global locks within authorities. Processing transactions is cleanly separated into two phases, namely (1) ensuring the transaction has exclusive access to the owned or shared objects at a specific version, and (2) then subsequently executing the transaction and committing its effects.

Phase (1) requires a transaction acquiring distributed locks at the granularity of objects. For owned objects this is performed trough a reliable broadcast primitive, that requires no global synchronization within the authority, and therefore can be scaled through sharding the management of locks across multiple machines by ObjID. For transactions involving shared objects sequencing is required using a consensus protocol, which does impose a global order on these transactions and has the potential to be a bottleneck. However, recent advances on engineering high-throughput consensus protocols [9] demonstrate that sequential execution is the bottleneck in state machine replication, not sequencing. In Sui, sequencing is only used to determine a version for the input shared object, namely incrementing an object version number and associating it with the transaction digest, rather than performing sequential execution.

Phase (2) takes place when the version of all input objects is known to an authority (and safely agreed across authorities) and involves execution of the Move transaction and commitment of its effects. Once the version of input objects is known, execution can

take place completely in parallel. Move virtual machines on multiple cores or physical machines read the versioned input objects, execute, and write the resulting objects from and to stores. The consistency requirements on stores for objects and transactions (besides the order lock map) are very loose, allowing scalable distributed key-value stores to be used internally by each authority. Execution is idempotent, making even crashes or hardware failures on components handling execution easy to recover from.

As a result, execution for transactions that are not causally related to each other can proceed in parallel. Smart contract designers may therefore design the data model of objects and operations within their contracts to take advantage of this parallelism.

Check-pointing and state commitments are computed off the critical transaction processing path to not block the handling of fresh transactions. These involve read operations on committed data rather than requiring computation and agreement before a transaction reaches finality. Therefore they do not affect the latency or throughput of processing new transactions, and can themselves be distributed across available resources.

Reads can benefit from very aggressive, and scalable caching. Authorities sign and make available all data that light clients require for reads, which may be served by distributed stores as static data. Certificates act as roots of trust for their full causal history of transactions and objects. State commitments further allow for the whole system to have regular global roots of trust for all state and transactions processed, at least every epoch or more frequently.

Latency. Smart contract designers are given the flexibility to control the latency of operations they define, depending on whether they involve owned or shared objects. Owned objects rely on a reliable broadcast before execution and commit, which requires two round trips to a quorum of authorities to reach finality. Operations involving shared objects, on the other hand, require a a consistent broadcast to create a certificate, and then be processed within a consensus protocol, leading to increased latency (4 to 8 round trips to quorums as of [9]).

REFERENCES

[1] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. 2021. Fraud and Data Availability Proofs: Detecting Invalid Blocks in Light Clients. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II (Lecture Notes in Computer Science, Vol. 12675)*, Nikita Borisov and Claudia Diaz (Eds.). Springer, 279–298.

[2] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the Age of Blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*. ACM, 183–198.

[3] Mathieu Baudet, George Danezis, and Alberto Sonnino. 2020. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*. ACM, 163–177.

[4] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Ra in, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. Move: A Language With Programmable Resources. https://developers.libra.org/docs/move-paper.

[5] Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, and Yoni Zohar. 2020. Resources: A Safe Language Abstraction for Money. *CoRR* abs/2004.05106 (2020). arXiv:2004.05106 https://arxiv.org/abs/2004.05106

[6] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

[7] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. 2021. SoK: Blockchain Light Clients. *IACR Cryptol. ePrint Arch.* (2021), 1657.

[8] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. 2020. Online Payments by Merely Broadcasting Messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 26–38.

[9] George Danezis, Eleftherios Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *CoRR* abs/2105.11827 (2021).

[10] David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. 2021. Fast and Reliable Formal Verification of Smart Contracts with the Move Prover. *CoRR* abs/2110.08362 (2021). arXiv:2110.08362 https://arxiv.org/abs/2110.08362

[11] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2018. AT2: Asynchronous Trustworthy Transfers. *CoRR* abs/1812.10844 (2018).

[12] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, Peter Robinson and Faith Ellen (Eds.). ACM, 307–316.

[13] Patrick McCorry, Chris Buckland, Bennet Yee, and Dawn Song. 2021. SoK: Validating Bridges as a Scaling Solution for Blockchains. *IACR Cryptol. ePrint Arch.* (2021), 1589.

[14] Marco Patrignani and Sam Blackshear. 2021. Robust Safety for Move. *CoRR* abs/2110.05043 (2021). arXiv:2110.05043 https://arxiv.org/abs/2110.05043

[15] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[16] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 137–150. https://doi.org/10.1007/978-3-030-53288-8_7