Object and Package Versioning

You reference every object stored on chain by an ID and version. When a transaction modifies an object, it writes the new contents to an on-chain reference with the same ID but a later version. This means that a single object (with ID I) might appear in multiple entries in the distributed store:

Despite appearing multiple times in the store, only one version of the object is available to transactions - the latest version (v2 in the previous example). Moreover, only one transaction can modify the object at that version to create a new version, guaranteeing a linear history (v1 was created in a state where I was at v0 , and v2 was created in a state where I was at v1).

Versions are strictly increasing and (ID, version) pairs are never re-used. This structure allows node operators to prune their stores of old object versions that are now inaccessible, if they choose. This is not a requirement, though, as node operators might keep prior object versions around to serve requests for an object's history, either from other nodes that are catching up, or from RPC requests.

Sui uses Lamport timestamps in its versioning algorithm for objects. The use of Lamport timestamps guarantees that versions never get re-used as the new version for objects touched by a transaction is one greater than the latest version among all input objects to the transaction. For example, a transaction transferring an object O at version 5 using a gas object G at version 3 updates both O and G versions to 1 + max(5, 3) = 6 (version 6).

The following sections detail the relevance of Lamport versions for maintaining the "no (ID, version) re-use" invariant or for accessing an object as a transaction input changes depending on that object's ownership.

You must reference address-owned transaction inputs at a specific ID and version. When a validator signs a transaction with an owned object input at a specific version, that version of the object is locked to that transaction. Validators reject requests to sign other transactions that require the same input (same ID and version).

If F + 1 validators sign one transaction that takes an object as input, and a different F + 1 validators sign a different transaction that takes the same object as input, that object (and all the other inputs to both transactions) is equivocated, meaning they cannot be used for any further transactions in that epoch. This is because neither transaction can form a quorum without relying on a signature from a validator that has already committed the object to a different transaction, which it cannot get. All locks are reset at the end of the epoch, which frees the objects again.

Only an object's owner can equivocate it, but this is not a desirable thing to do. You can avoid equivocation by carefully managing the versions of address-owned input objects: never attempt to execute two different transactions that use the same object. If you don't get a definite success or failure response from the network for a transaction, assume that the transaction might have gone through, and do not re-use any of its objects for different transactions.

Like address-owned objects, you reference immutable objects at an ID and version, but they do not need to be locked as their contents and versions do not change. Their version is relevant because they could have started life as an address-owned object before being frozen. The given version identifies the point at which they became immutable.

Specifying a shared transaction input is slightly more complex. You reference it by its ID, the version it was shared at, and a flag indicating whether it is accessed mutably. You don't specify the precise version the transaction accesses because consensus decides that during transaction scheduling. When scheduling multiple transactions that touch the same shared object, validators agree the order of those transactions, and pick each transaction's input versions for the shared object accordingly (one transaction's output version becomes the next transaction's input version, and so on).

Shared transaction inputs that you reference immutably participate in scheduling, but don't modify the object or increment its version.

You can't access wrapped objects by their ID in the object store, you must access them by the object that wraps them. Consider the following example that creates a make_wrapped function with an Inner object, wrapped in an Outer object, which is returned to the transaction sender.

The owner of Outer in this example must specify it as the transaction input and then access its inner field to read the instance of Inner . Validators refuse to sign transactions that directly specify wrapped objects (like the inner of an Outer) as inputs. As a result, you don't need to specify a wrapped object's version in a transaction that reads that object.

Wrapped objects can eventually become "unwrapped", meaning that they are once again accessible at their ID:

The unwrap function in the previous code takes an instance of Outer , destroys it, and sends the Inner back to the sender. After calling this function, the previous owner of Outer can access Inner directly by its ID because it is now unwrapped. Wrapping and unwrapping of an object can happen multiple times across its lifespan, and the object retains its ID across all those events.

The Lamport timestamp-based versioning scheme ensures that the version that an object is unwrapped at is always greater than the version it was wrapped at, to prevent version re-use.

This leads to the following chain of inequalities for I's version before wrapping:

So the I version before wrapping is less than the I version after unwrapping.

From a versioning perspective, values held in dynamic fields behave like wrapped objects:

One distinction dynamic fields have to wrapped objects is that if a transaction modifies a dynamic object field, its version is incremented in that transaction, where a wrapped object's version would not be.

Adding a new dynamic field to a parent object also creates a Field object, responsible for associating the field name and value with that parent. Unlike other newly created objects, the ID for the resulting instance of Field is not created using sui::object::new . Instead, it is computed as a hash of the parent object ID and the type and value of the field name, so that you can use it to look-up the Field via its parent and name.

When you remove a field, Sui deletes its associated Field , and if you add a new field with the same name, Sui creates a new instance with the same ID. Versioning using Lamport timestamps, coupled with dynamic fields being only accessible through their parent object, ensures that (ID, version) pairs are not reused in the process:

So the version of the new Field instance is greater than the version of the deleted Field .

Move packages are also versioned and stored on chain, but follow a different versioning scheme to objects because they are immutable from their inception. This means that you refer to package transaction inputs (for example, the package that a function is from for a Move call transaction) by just their ID, and are always loaded at their latest version.

Every time you publish or upgrade a package, Sui generates a new ID. A newly published package has its version set to 1, whereas an upgraded package's version is one greater than the package it is upgrading. Unlike objects, older versions of a package remain accessible even after being upgraded. For example, imagine a package P that is published and upgraded twice. It might be represented in the store as:

In this example, all three versions of the same package are at different IDs. The packages have increasing versions but it is possible to call into v1 , even though v2 and v3 exist on chain.

Framework packages (such as the Move standard library at 0x1 ,the Sui Framework at 0x2 , Sui System at 0x3 and Deepbook at 0xdee9) are a special-case because their IDs must remain stable across upgrades. The network can upgrade framework packages while preserving their IDs via a system transaction, but can only perform this operation on epoch boundaries because they are considered immutable like other packages. New versions of framework packages retain the same ID as their predecessor, but increment their version by one:

The prior example shows the on-chain representation of the first three versions of the Move standard library.

Sui smart contracts are organized into upgradeable packages and, as a result, multiple versions of any given package can exist on chain. Before someone can use an on-chain package, you must publish its first, original version. When you upgrade a package, you create a new version of that package. Each upgrade of a package is based on the immediately preceding version of that package in the versions history. In other words, you can upgrade the nth version of a package from only the nth - 1 version. For example, you can upgrade a package from version 1 to 2, but afterwards you can upgrade that package only from version 2 to 3; you're not allowed to upgrade from version 1 to 3.

There is a notion of versioning in package manifest files, existing in both the package section and in the dependencies section. For example, consider the manifest code that follows:

At this point, the version references in the manifest are used only for user-level documentation as the publish and upgrade commands do not leverage this information. If you publish a package with a certain package version in the manifest file and then modify and re-publish the same package with a different version (using publish command rather than upgrade command), the two are considered different packages, rather than on-chain versions of the same package. You cannot use any of these packages as a dependency override to stand in for the other one. While you can specify this type of override when building a package, it results in an error when publishing or upgrading on chain.

Move objects

Sui uses Lamport timestamps in its versioning algorithm for objects. The use of Lamport timestamps guarantees that versions never

get re-used as the new version for objects touched by a transaction is one greater than the latest version among all input objects to the transaction. For example, a transaction transferring an object O at version 5 using a gas object G at version 3 updates both O and G versions to 1 + max(5, 3) = 6 (version 6).

The following sections detail the relevance of Lamport versions for maintaining the "no (ID, version) re-use" invariant or for accessing an object as a transaction input changes depending on that object's ownership.

You must reference address-owned transaction inputs at a specific ID and version. When a validator signs a transaction with an owned object input at a specific version, that version of the object is locked to that transaction. Validators reject requests to sign other transactions that require the same input (same ID and version).

If F + 1 validators sign one transaction that takes an object as input, and a different F + 1 validators sign a different transaction that takes the same object as input, that object (and all the other inputs to both transactions) is equivocated, meaning they cannot be used for any further transactions in that epoch. This is because neither transaction can form a quorum without relying on a signature from a validator that has already committed the object to a different transaction, which it cannot get. All locks are reset at the end of the epoch, which frees the objects again.

Only an object's owner can equivocate it, but this is not a desirable thing to do. You can avoid equivocation by carefully managing the versions of address-owned input objects: never attempt to execute two different transactions that use the same object. If you don't get a definite success or failure response from the network for a transaction, assume that the transaction might have gone through, and do not re-use any of its objects for different transactions.

Like address-owned objects, you reference immutable objects at an ID and version, but they do not need to be locked as their contents and versions do not change. Their version is relevant because they could have started life as an address-owned object before being frozen. The given version identifies the point at which they became immutable.

Specifying a shared transaction input is slightly more complex. You reference it by its ID, the version it was shared at, and a flag indicating whether it is accessed mutably. You don't specify the precise version the transaction accesses because consensus decides that during transaction scheduling. When scheduling multiple transactions that touch the same shared object, validators agree the order of those transactions, and pick each transaction's input versions for the shared object accordingly (one transaction's output version becomes the next transaction's input version, and so on).

Shared transaction inputs that you reference immutably participate in scheduling, but don't modify the object or increment its version.

You can't access wrapped objects by their ID in the object store, you must access them by the object that wraps them. Consider the following example that creates a make_wrapped function with an Inner object, wrapped in an Outer object, which is returned to the transaction sender.

The owner of Outer in this example must specify it as the transaction input and then access its inner field to read the instance of Inner . Validators refuse to sign transactions that directly specify wrapped objects (like the inner of an Outer) as inputs. As a result, you don't need to specify a wrapped object's version in a transaction that reads that object.

Wrapped objects can eventually become "unwrapped", meaning that they are once again accessible at their ID:

The unwrap function in the previous code takes an instance of Outer , destroys it, and sends the Inner back to the sender. After calling this function, the previous owner of Outer can access Inner directly by its ID because it is now unwrapped. Wrapping and unwrapping of an object can happen multiple times across its lifespan, and the object retains its ID across all those events.

The Lamport timestamp-based versioning scheme ensures that the version that an object is unwrapped at is always greater than the version it was wrapped at, to prevent version re-use.

This leads to the following chain of inequalities for I's version before wrapping:

So the I version before wrapping is less than the I version after unwrapping.

From a versioning perspective, values held in dynamic fields behave like wrapped objects:

One distinction dynamic fields have to wrapped objects is that if a transaction modifies a dynamic object field, its version is incremented in that transaction, where a wrapped object's version would not be.

Adding a new dynamic field to a parent object also creates a Field object, responsible for associating the field name and value with that parent. Unlike other newly created objects, the ID for the resulting instance of Field is not created using sui::object::new . Instead, it is computed as a hash of the parent object ID and the type and value of the field name, so that you can use it to look-up the Field via its parent and name.

When you remove a field, Sui deletes its associated Field , and if you add a new field with the same name, Sui creates a new instance with the same ID. Versioning using Lamport timestamps, coupled with dynamic fields being only accessible through their parent object, ensures that (ID, version) pairs are not reused in the process:

So the version of the new Field instance is greater than the version of the deleted Field .

Move packages are also versioned and stored on chain, but follow a different versioning scheme to objects because they are immutable from their inception. This means that you refer to package transaction inputs (for example, the package that a function is from for a Move call transaction) by just their ID, and are always loaded at their latest version.

Every time you publish or upgrade a package, Sui generates a new ID. A newly published package has its version set to 1, whereas an upgraded package's version is one greater than the package it is upgrading. Unlike objects, older versions of a package remain accessible even after being upgraded. For example, imagine a package P that is published and upgraded twice. It might be represented in the store as:

In this example, all three versions of the same package are at different IDs. The packages have increasing versions but it is possible to call into v1 , even though v2 and v3 exist on chain.

Framework packages (such as the Move standard library at 0x1 ,the Sui Framework at 0x2 , Sui System at 0x3 and Deepbook at 0xdee9) are a special-case because their IDs must remain stable across upgrades. The network can upgrade framework packages while preserving their IDs via a system transaction, but can only perform this operation on epoch boundaries because they are considered immutable like other packages. New versions of framework packages retain the same ID as their predecessor, but increment their version by one:

The prior example shows the on-chain representation of the first three versions of the Move standard library.

Sui smart contracts are organized into upgradeable packages and, as a result, multiple versions of any given package can exist on chain. Before someone can use an on-chain package, you must publish its first, original version. When you upgrade a package, you create a new version of that package. Each upgrade of a package is based on the immediately preceding version of that package in the versions history. In other words, you can upgrade the nth version of a package from only the nth - 1 version. For example, you can upgrade a package from version 1 to 2, but afterwards you can upgrade that package only from version 2 to 3; you're not allowed to upgrade from version 1 to 3.

There is a notion of versioning in package manifest files, existing in both the package section and in the dependencies section. For example, consider the manifest code that follows:

At this point, the version references in the manifest are used only for user-level documentation as the publish and upgrade commands do not leverage this information. If you publish a package with a certain package version in the manifest file and then modify and re-publish the same package with a different version (using publish command rather than upgrade command), the two are considered different packages, rather than on-chain versions of the same package. You cannot use any of these packages as a dependency override to stand in for the other one. While you can specify this type of override when building a package, it results in an error when publishing or upgrading on chain.

Packages

Move packages are also versioned and stored on chain, but follow a different versioning scheme to objects because they are immutable from their inception. This means that you refer to package transaction inputs (for example, the package that a function is from for a Move call transaction) by just their ID, and are always loaded at their latest version.

Every time you publish or upgrade a package, Sui generates a new ID. A newly published package has its version set to 1, whereas an upgraded package's version is one greater than the package it is upgrading. Unlike objects, older versions of a package remain accessible even after being upgraded. For example, imagine a package P that is published and upgraded twice. It might be represented in the store as:

In this example, all three versions of the same package are at different IDs. The packages have increasing versions but it is possible to call into v1 , even though v2 and v3 exist on chain.

Framework packages (such as the Move standard library at 0x1 ,the Sui Framework at 0x2 , Sui System at 0x3 and Deepbook at 0xdee9) are a special-case because their IDs must remain stable across upgrades. The network can upgrade framework packages while preserving their IDs via a system transaction, but can only perform this operation on epoch boundaries because they are considered immutable like other packages. New versions of framework packages retain the same ID as their predecessor, but increment their version by one:

The prior example shows the on-chain representation of the first three versions of the Move standard library.

Sui smart contracts are organized into upgradeable packages and, as a result, multiple versions of any given package can exist on chain. Before someone can use an on-chain package, you must publish its first, original version. When you upgrade a package, you create a new version of that package. Each upgrade of a package is based on the immediately preceding version of that package in the versions history. In other words, you can upgrade the nth version of a package from only the nth - 1 version. For example, you can upgrade a package from version 1 to 2, but afterwards you can upgrade that package only from version 2 to 3; you're not allowed to upgrade from version 1 to 3.

There is a notion of versioning in package manifest files, existing in both the package section and in the dependencies section. For example, consider the manifest code that follows:

At this point, the version references in the manifest are used only for user-level documentation as the publish and upgrade commands do not leverage this information. If you publish a package with a certain package version in the manifest file and then modify and re-publish the same package with a different version (using publish command rather than upgrade command), the two are considered different packages, rather than on-chain versions of the same package. You cannot use any of these packages as a dependency override to stand in for the other one. While you can specify this type of override when building a package, it results in an error when publishing or upgrading on chain.