Avoiding Equivocation

Equivocation is when validators send conflicting information about objects on the network. Sometimes this is because of bad actors trying to subvert the integrity of the Sui network. The network has measures in place to punish validators that might engage in such behavior.

The other source of equivocation does not originate from bad intentions. There are logic traps that smart contract code can inadvertently trigger, like when dealing with sponsored transactions . The effects of this type of equivocation can lock the objects your code interacts with until the end of the current epoch. This can lead to frustration for you, and more importantly, your users.

To avoid double spending, validators lock objects as they validate transactions. An equivocation occurs when an owned object pair (ObjectId , SequenceNumber) is concurrently used in multiple non-finalized transactions.

Perhaps the most common source of equivocation stems from the attempt to pay for gas using the same coin object. This can be a problem when performing multiple transactions that originate from the same sender.

It's not just objects that can cause equivocation. Any object used in multiple transactions can be a source of equivocation if not handled properly. Sui uses object versioning to track the status of objects across transactions and epochs. If a transaction modifies an object, then one of the results of that transaction is to update the version of that same object.

The versiong architecture is what supports paying the gas for a series of transactions using the same gas coin. Among other processing, the first transaction advances the version number of the coin and returns it to the sender. The next transaction can then use that same coin to pay for its gas. Equivocation is avoided because each transaction references a different version of the same coin.

For most smart contracts, equivocation is not something that's an intended behavior. Perhaps the most common source of unintentional equivocation comes from multiple transactions performed by the same address. If you don't take care to handle the gas fees properly, you could lock up your smart contract by trying to use the same coin (and version) for more than one transaction.

If using a single thread, serialize transactions that use the same owned object. PTBs allow your transactions to use multiple operations against the same owned object. A PTB is essentially a single, serialized transaction, which can prevent SequenceNumber errors.

You should always take advantage of the inherent batching that PTBs provide. For example, consider an airdrop scenario where you want to mint and transfer an object to many users. Because PTBs allow up to 1,024 operations in a single PTB, you can airdrop your object to 512 users in a single transaction. This approach is much more cost efficient than looping over 512 individual transactions that mint and transfer to a single user each time. Batching transactions might remove the need for parallel execution, but you must consider the atomic nature of PTBs; if one instruction fails, the whole PTB fails. Consequently, parallel transactions might be preferred for some use cases.

Parallel transactions from multiple threads can cause unintentional equivocation errors if not managed properly. One way to avoid owned object equivocation is to create a separate owned object for each transaction thread. This ensures that each thread uses the correct version of its object input.

In cases where creating multiple owned objects is not practical or desired, you can create a wrapper around an object used across threads. The wrapper is a shared object that authorizes access to its object through an allowlist. Any time a transaction needs to access the wrapped object, it gets permission from the wrapper in the same PTB. When authorization needs transferring, the allowlist for the wrapper gets updated accordingly. This approach can create a latency bottleneck as the object wrapper creates sequentialized transactions that rely on its object for input. The Sui TypeScript SDK provides an executor class, ParallelTransactionExecutor , to process parallel transactions efficiently.

The Sui SDK offers transaction executors to help process multiple transactions from the same address.

Use the SerialTransactionExecutor when processing transactions one after another. The executor grabs all the coins from the sender and combines them into a single coin that is used for all transactions.

Using the SerialTransactionExecutor prevents SequenceNumber errors by handling the versioning of object inputs across a PTB.

Use the ParallelTransactionExecutor when you want to process transactions with the same sender at the same time. This class creates a pool of gas coins that it manages to ensure parallel transactions don't equivocate those coins. The class tracks objects used across transactions and orders their processing so that the object inputs are also not equivocated.

If you find your smart contracts unintentionally locking objects, there are some tools you can use to help fix the issue.

You can install the sui-tool (https://github.com/MystenLabs/sui/tree/main/crates/sui-tool) utility and use the locked-object command to check the locked status of the passed asset on a specific RPC network (--fullnode-rpc-url value). If you provide an address, locked-object checks if all the gas objects owned by that address are locked. Pass an object ID to check if that specific object is locked.

Include the --rescue flag to try and unlock the object the command targets. Rescue is possible if the object isn't already locked by a majority of validators.

Common pitfalls to avoid

For most smart contracts, equivocation is not something that's an intended behavior. Perhaps the most common source of unintentional equivocation comes from multiple transactions performed by the same address. If you don't take care to handle the gas fees properly, you could lock up your smart contract by trying to use the same coin (and version) for more than one transaction.

If using a single thread, serialize transactions that use the same owned object. PTBs allow your transactions to use multiple operations against the same owned object. A PTB is essentially a single, serialized transaction, which can prevent SequenceNumber errors.

You should always take advantage of the inherent batching that PTBs provide. For example, consider an airdrop scenario where you want to mint and transfer an object to many users. Because PTBs allow up to 1,024 operations in a single PTB, you can airdrop your object to 512 users in a single transaction. This approach is much more cost efficient than looping over 512 individual transactions that mint and transfer to a single user each time. Batching transactions might remove the need for parallel execution, but you must consider the atomic nature of PTBs; if one instruction fails, the whole PTB fails. Consequently, parallel transactions might be preferred for some use cases.

Parallel transactions from multiple threads can cause unintentional equivocation errors if not managed properly. One way to avoid owned object equivocation is to create a separate owned object for each transaction thread. This ensures that each thread uses the correct version of its object input.

In cases where creating multiple owned objects is not practical or desired, you can create a wrapper around an object used across threads. The wrapper is a shared object that authorizes access to its object through an allowlist. Any time a transaction needs to access the wrapped object, it gets permission from the wrapper in the same PTB. When authorization needs transferring, the allowlist for the wrapper gets updated accordingly. This approach can create a latency bottleneck as the object wrapper creates sequentialized transactions that rely on its object for input. The Sui TypeScript SDK provides an executor class, ParallelTransactionExecutor , to process parallel transactions efficiently.

The Sui SDK offers transaction executors to help process multiple transactions from the same address.

Use the SerialTransactionExecutor when processing transactions one after another. The executor grabs all the coins from the sender and combines them into a single coin that is used for all transactions.

Using the SerialTransactionExecutor prevents SequenceNumber errors by handling the versioning of object inputs across a PTB.

Use the ParallelTransactionExecutor when you want to process transactions with the same sender at the same time. This class creates a pool of gas coins that it manages to ensure parallel transactions don't equivocate those coins. The class tracks objects used across transactions and orders their processing so that the object inputs are also not equivocated.

If you find your smart contracts unintentionally locking objects, there are some tools you can use to help fix the issue.

You can install the sui-tool (https://github.com/MystenLabs/sui/tree/main/crates/sui-tool) utility and use the locked-object command to check the locked status of the passed asset on a specific RPC network (--fullnode-rpc-url value). If you provide an address, locked-object checks if all the gas objects owned by that address are locked. Pass an object ID to check if that specific object is locked.

Include the --rescue flag to try and unlock the object the command targets. Rescue is possible if the object isn't already locked by a majority of validators.

Sui SDK

The Sui SDK offers transaction executors to help process multiple transactions from the same address.

Use the SerialTransactionExecutor when processing transactions one after another. The executor grabs all the coins from the sender and combines them into a single coin that is used for all transactions.

Using the SerialTransactionExecutor prevents SequenceNumber errors by handling the versioning of object inputs across a PTB.

Use the ParallelTransactionExecutor when you want to process transactions with the same sender at the same time. This class creates a pool of gas coins that it manages to ensure parallel transactions don't equivocate those coins. The class tracks objects used across transactions and orders their processing so that the object inputs are also not equivocated.

If you find your smart contracts unintentionally locking objects, there are some tools you can use to help fix the issue.

You can install the sui-tool (https://github.com/MystenLabs/sui/tree/main/crates/sui-tool) utility and use the locked-object command to check the locked status of the passed asset on a specific RPC network (--fullnode-rpc-url value). If you provide an address, locked-object checks if all the gas objects owned by that address are locked. Pass an object ID to check if that specific object is locked.

Include the --rescue flag to try and unlock the object the command targets. Rescue is possible if the object isn't already locked by a majority of validators.

Debug tools

If you find your smart contracts unintentionally locking objects, there are some tools you can use to help fix the issue.

You can install the sui-tool (https://github.com/MystenLabs/sui/tree/main/crates/sui-tool) utility and use the locked-object command to check the locked status of the passed asset on a specific RPC network (--fullnode-rpc-url value). If you provide an address, locked-object checks if all the gas objects owned by that address are locked. Pass an object ID to check if that specific object is locked.

Include the --rescue flag to try and unlock the object the command targets. Rescue is possible if the object isn't already locked by a majority of validators.

Related links