@vbuterin @JustinDrake

The basic idea is to use read and write locks to ensure a transaction that references data on many shards can be executed atomically.

Suppose we had a set S

for the state/storage required by a transaction T

– which specifies:

1. The address of a blob of data;
2. Whether a read or write lock is required
3. The ID of each shard where the blob of data is held

Prepare Phase

Before we can execute T

, we require that a set of locks L

for storage S

be added to the block-chain. A lock l \in L

for a storage s \in S

is like a transaction except that it needs to be finalised by both the shard where s

is kept and T's parent shard.

Commit Phase

Before T

is executed, we need to:

(1) present merkle proofs that L

has been added to the state of all shards referenced in S

(2) merkle branches for the storage of S

.

Both (1) and (2) would need to be committed in T

's shard before T

can be executed and committed - otherwise no one in T

's shard would be able to check T

has been executed correctly.

How can we prevent deadlock in this system?

1. A simple solution is to have a time-out for lock-holding, which could be a certain block height, and to prevent a block from acquiring a lock for some time. This may be too slow.
2. A better solution would be using block-chains to time-stamp transactions T using the block-height of T's first reservation. With such a time-stamp, we could employ deadlock prevention using a wound-wait mechanism.

Edit

: See this post about resolving deadlock using the wound-wait scheme.