This post follows up from a post where I describe a mechanism to use locks to ensure atomicity of execution on the block-chain. In this post, I follow this up with a description of how to resolve dead-lock using the block-chain to time-stamp locks.

Wound-Wait mechanism to resolve deadlock

The wound-wait mechanism is a pre-emptive technique for deadlock prevention. When transaction T_i

requests a data item currently held by T_j

, T_i

is allowed to wait only if it has a time-stamp greater than that of T_j

, otherwise T_j

is rolled back (wounded).

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur:

1. If $\mathrm{timestamp}(T_i) < \text{timestamp}(T_j)$

, then T_i

forces T_j

to be rolled back – that is T_i

wounds T_j

. T_j

is restarted later with a random delay but with the same time-stamp.

1. If $\mathrm{timestamp}(T_i) > \text{timestamp}(T_j)$

, then T_i

is forced to wait until the resource is available.

Implementing this on the block-chain

Suppose that before any lock for a transaction gets added to the bock-chain, a transaction initialiser must be added, specifying all the storage and locks that could be referenced by a transaction.

The time-stamping of transaction T is the block-height of its initialiser.

Now, how can we ensure that condition (1) and (2) hold?

case (1)

If validators on a shard find that some T_i

's time-stamp is less than a transaction T_j

's on their shard, and T_j

is blocking T_i

from a resource, then the shard will refer to the locks T_j

has acquired, and contact other shards proving T_j

's other locks must be rolled back. Assuming we have data-availability proofs, all this information is available to clients in T_j

's shard, and so they can also forward the information to other shards as a fail-safe.

case (2)

I do have concerns with case 2. If a transaction T_i

must wait for another transaction T_j

with an earlier timestamp to finish execution, and T_j

never releases its locks then what happens? Is it ok for a transaction T_j

to “own” data? Perhaps this issue can be mitigated through time limits on locks, which may be specified by shards or even blobs of data themselves.

Pro

of the locking scheme: transactions execute atomically, and can be rolled back more or less atomically if need be.

con

- comes with some overhead.