Not my area historically, likely missed some conversation, but was motivated to start compiling the chaotic wide spread disucssion, and to surface some things I liked about other languages but couldn't find in discussions. Credits to Dean for CT "motivational" tweet, getting more people to look at solving problems. Try it yourself some time, it's fun ;).

Atomic cross-shard transactions

Problem: shards are isolated, but transactions are meant to either execute fully, or not at all.

Also referred to as "Train and hotel problem".

Problem extension: If you take an approach where you "reserve" (lock) some resource, how do you:

- Speed up the unlocking.
- Instant case = synchronous cross-shard transactions
- Instant case = synchronous cross-shard transactions
- Avoid blocking other actions that could have proceeded.
- Design of locking and/or nonces
- Design of locking and/or nonces
- Allow out-of-order transactions, fundamental to a fee market (nothing to outbid otherwise)

Existing approaches

In no particular order:

- Receipts
- Basic non-atomic approach: first half on A, produce a receipt, continue on B.
- "Already consumed receipt IDs" need to be tracked for replay protection, even though it's otherwise a stateless system.
- Receipts can be processed out of order
- "Already consumed receipt IDs" need to be tracked for replay protection, even though it's otherwise a stateless system.
- Receipts can be processed out of order
- Sequential receipts:
- easy Replay protection
- small state representation of shard A -> shard B receipt status.
- every shard A maintains in its state, for every other shard B, two values:
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- DOS problem (everyone direct to a single shard) affected with fee adjustments.
- Non-standard state => witness problems => no direct effects. But work around is like "in order to include one of your own transactions, you must also provide witnesses for a cross-shard receipt that is in the queue"
- Comment from Dankrad: One disadvantage of rate-limiting is that then, there are no guarantees that receipts will be received in any finite amount of time, which makes potential locking mechanisms more difficult to design. The effect of that could be reduced if the sending shard had a way to check consumption of a receipt. I guess that could be done by keeping receipt counters on the beacon chain.
- easy Replay protection

- small state representation of shard A -> shard B receipt status.
- every shard A maintains in its state, for every other shard B, two values:
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- DOS problem (everyone direct to a single shard) affected with fee adjustments.
- Non-standard state => witness problems => no direct effects. But work around is like "in order to include one of your own transactions, you must also provide witnesses for a cross-shard receipt that is in the queue"
- Comment from Dankrad: One disadvantage of rate-limiting is that then, there are no guarantees that receipts will be received in any finite amount of time, which makes potential locking mechanisms more difficult to design. The effect of that could be reduced if the sending shard had a way to check consumption of a receipt. I guess that could be done by keeping receipt counters on the beacon chain.
- Sequential receipts with bitfields. (Will?)
- Basic non-atomic approach: first half on A, produce a receipt, continue on B.
- "Already consumed receipt IDs" need to be tracked for replay protection, even though it's otherwise a stateless system.
- Receipts can be processed out of order
- "Already consumed receipt IDs" need to be tracked for replay protection, even though it's otherwise a stateless system.
- Receipts can be processed out of order
- Sequential receipts:
- easy Replay protection
- small state representation of shard A -> shard B receipt status.
- every shard A maintains in its state, for every other shard B, two values:
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- DOS problem (everyone direct to a single shard) affected with fee adjustments.
- Non-standard state => witness problems => no direct effects. But work around is like "in order to include one of your own transactions, you must also provide witnesses for a cross-shard receipt that is in the queue"
- Comment from Dankrad: One disadvantage of rate-limiting is that then, there are no guarantees that receipts will be received in any finite amount of time, which makes potential locking mechanisms more difficult to design. The effect of that could be reduced if the sending shard had a way to check consumption of a receipt. I guess that could be done by keeping receipt counters on the beacon chain.
- easy Replay protection
- small state representation of shard A -> shard B receipt status.
- every shard A maintains in its state, for every other shard B, two values:
- (i) the nonce of the next receipt that will be sent from A to B
- (ii) the nonce of the next receipt that will be received from B to A.
- (i) the nonce of the next receipt that will be sent from A to B

- (ii) the nonce of the next receipt that will be received from B to A.
- DOS problem (everyone direct to a single shard) affected with fee adjustments.
- Non-standard state => witness problems => no direct effects. But work around is like “in order to include one of your own transactions, you must also provide witnesses for a cross-shard receipt that is in the queue”
- Comment from Dankrad: One disadvantage of rate-limiting is that then, there are no guarantees that receipts will be received in any finite amount of time, which makes potential locking mechanisms more difficult to design. The effect of that could be reduced if the sending shard had a way to check consumption of a receipt. I guess that could be done by keeping receipt counters on the beacon chain.
- Sequential receipts with bitfields. (Will?)
- Message passing
- Beacon chain overhead, scaling limit. Fits better with CBC.
- Affects fork-choice, needs to detect and disallow conflicting messages
- Beacon chain overhead, scaling limit. Fits better with CBC.
- Affects fork-choice, needs to detect and disallow conflicting messages
- Contract yanking
- In general, it’s a bad idea from a usability perspective for a contract that could be of interest to many users to be yankable, as yanking makes the contract unusable until it gets reinstantiated in the target_shard. For these two reasons, the most likely workflow will be for contracts to have behavior similar to the hotel room above, where the contract separates out the state related to individual interactions so that it can be moved between shards separately.
- Same problem as locking, just nicer semantics if not talking about many simulatenous same-resource users.
- Same problem as locking, just nicer semantics if not talking about many simulatenous same-resource users.
- In general, it’s a bad idea from a usability perspective for a contract that could be of interest to many users to be yankable, as yanking makes the contract unusable until it gets reinstantiated in the target_shard. For these two reasons, the most likely workflow will be for contracts to have behavior similar to the hotel room above, where the contract separates out the state related to individual interactions so that it can be moved between shards separately.
- Same problem as locking, just nicer semantics if not talking about many simulatenous same-resource users.
- Same problem as locking, just nicer semantics if not talking about many simulatenous same-resource users.
- Merge blocks
- “either both blocks are part of the final chain or neither are”
- Fuzzy shard responsibilities / requirements, and light client for single shard is affected.
- “either both blocks are part of the final chain or neither are”
- Fuzzy shard responsibilities / requirements, and light client for single shard is affected.
- Leader shards (Merge blocks discussion) cycle through which shard is the “primary” shard during each block height, and during that height allow transactions from that shard to read and write to all shards
- Yanking, Rust style. Implement ownership and borrowing.
- Optimistic style communication. Optimistic state roots (Conditional state objects):A layer 2 computing model using optimistic state roots Optimistic receipt roots (Store contract copies with different dependency signatures):Fast cross-shard transfers via optimistic receipt roots
- Sharded Byzantine Atomic Commit (S-BAC)
- R/W locking with timeouts
- IMHO the start of a good approach: 0 pressure on beacon-chain other than shard data root syncing.
- Problem:
- committing locks for each part of state takes time

- deadlocks to be avoided with lock timeouts
- set of locks focused on storage. Not general enough for EEs
- committing locks for each part of state takes time
- deadlocks to be avoided with lock timeouts
- set of locks focused on storage. Not general enough for EEs
- Comment from Vitalik: The problem I have with locking mechanisms is that they prevent any other activity from happening while the lock is active, which could be extremely inconvenient for users. In the train-and-hotel example, a single user would be able to stop all other users from booking trains or hotels for whatever the length of the lock is. Sure, you could lock individual train tickets, but that's already a fairly application specific solution.
- Vitalik iterating on it. But still close to storage locking, and targetting shards explicitly:
- Cross Shard Locking Scheme - (1)
- Cross Shard Locking Scheme - (1)
- Deadlocking problems
- Wound-wait, timestamp locks, but comparing transaction timestamps seems too complex.Resolving deadlock
- Wound-wait, timestamp locks, but comparing transaction timestamps seems too complex.Resolving deadlock
- Interesting logging based communication of locks
- March '18, Vitalik: It's worth noting that at this point I consider yanking to be a superior alternative to all of these locking techniques.
- Extended, abstracted and generalized away in proposal below
- IMHO the start of a good approach: 0 pressure on beacon-chain other than shard data root syncing.
- Problem:
- committing locks for each part of state takes time
- deadlocks to be avoided with lock timeouts
- set of locks focused on storage. Not general enough for EEs
- committing locks for each part of state takes time
- deadlocks to be avoided with lock timeouts
- set of locks focused on storage. Not general enough for EEs
- Comment from Vitalik: The problem I have with locking mechanisms is that they prevent any other activity from happening while the lock is active, which could be extremely inconvenient for users. In the train-and-hotel example, a single user would be able to stop all other users from booking trains or hotels for whatever the length of the lock is. Sure, you could lock individual train tickets, but that's already a fairly application specific solution.
- Vitalik iterating on it. But still close to storage locking, and targetting shards explicitly:
- Cross Shard Locking Scheme - (1)
- Cross Shard Locking Scheme - (1)
- Deadlocking problems
- Wound-wait, timestamp locks, but comparing transaction timestamps seems too complex.Resolving deadlock
- Wound-wait, timestamp locks, but comparing transaction timestamps seems too complex.Resolving deadlock
- Interesting logging based communication of locks
- March '18, Vitalik: It's worth noting that at this point I consider yanking to be a superior alternative to all of these locking techniques.
- Extended, abstracted and generalized away in proposal below

- Vitalik: Simple synchronous cross-shard transaction protocol
- We already know that it is relatively easy to support asynchronous cross-shard communication 100, and we can extend that to solve train-and-hotel problems via cross-shard yanking 16, but this is still highly imperfect, because of the high latency inherent in waiting for multiple rounds of cross-shard communication.
- Steps:
- Download block N on shard A.
- Collect all “foreign references” (that is, address references in transactions in shard A that come from other shards). For each foreign reference, ask the network for a Merkle branch for the state of the associated address from height N-1, and the block at the given position at height N.
- For all transactions in shard A, verify that the references are consistent; that is, for every reference (s_id, addr) in a transaction T (foreign or local), verify that the value of the block of shard s_id at position addr is also T, using the data acquired in stage 2 for foreign references. If it is not, throw the transaction out.
- For every transaction that passed step (3), execute it, using the state data acquired in stage 2.
- Use cryptoeconomic claims, ZK-SNARKs, or any other mechanism to gather an opinion about the state roots of other shards at height N.

- Download block N on shard A.
- Collect all “foreign references” (that is, address references in transactions in shard A that come from other shards). For each foreign reference, ask the network for a Merkle branch for the state of the associated address from height N-1, and the block at the given position at height N.
- For all transactions in shard A, verify that the references are consistent; that is, for every reference (s_id, addr) in a transaction T (foreign or local), verify that the value of the block of shard s_id at position addr is also T, using the data acquired in stage 2 for foreign references. If it is not, throw the transaction out.
- For every transaction that passed step (3), execute it, using the state data acquired in stage 2.
- Use cryptoeconomic claims, ZK-SNARKs, or any other mechanism to gather an opinion about the state roots of other shards at height N.

- Problems:
- Full transaction data to be included on all touched shards
- Assumes accounts use-case

- Full transaction data to be included on all touched shards
- Assumes accounts use-case

- Important: fees can be modeled to allow for high-probability synchronous tx inclusion. 90%

- We already know that it is relatively easy to support asynchronous cross-shard communication 100, and we can extend that to solve train-and-hotel problems via cross-shard yanking 16, but this is still highly imperfect, because of the high latency inherent in waiting for multiple rounds of cross-shard communication.
- Steps:
- Download block N on shard A.
- Collect all “foreign references” (that is, address references in transactions in shard A that come from other shards). For each foreign reference, ask the network for a Merkle branch for the state of the associated address from height N-1, and the block at the given position at height N.
- For all transactions in shard A, verify that the references are consistent; that is, for every reference (s_id, addr) in a transaction T (foreign or local), verify that the value of the block of shard s_id at position addr is also T, using the data acquired in stage 2 for foreign references. If it is not, throw the transaction out.
- For every transaction that passed step (3), execute it, using the state data acquired in stage 2.
- Use cryptoeconomic claims, ZK-SNARKs, or any other mechanism to gather an opinion about the state roots of other shards at height N.
- Download block N on shard A.

- Collect all “foreign references” (that is, address references in transactions in shard A that come from other shards). For each foreign reference, ask the network for a Merkle branch for the state of the associated address from height N-1, and the block at the given position at height N.
- For all transactions in shard A, verify that the references are consistent; that is, for every reference (s_id, addr) in a transaction T (foreign or local), verify that the value of the block of shard s_id at position addr is also T, using the data acquired in stage 2 for foreign references. If it is not, throw the transaction out.
- For every transaction that passed step (3), execute it, using the state data acquired in stage 2.
- Use cryptoeconomic claims, ZK-SNARKs, or any other mechanism to gather an opinion about the state roots of other shards at height N.
- Problems:
- Full transaction data to be included on all touched shards
- Assumes accounts use-case
- Full transaction data to be included on all touched shards
- Assumes accounts use-case
- Important: fees can be modeled to allow for high-probability synchronous tx inclusion. 90%

Step back, simple approach

This is a re-hash of all of the above ideas, simplified for the fast crosslinking we have:

We have fast cross-shard 1-slot state roots now (in healthy shards case).

The locking can be improved with this: timeouts are less scary, single slot cross-shard atomic transactions are easily achieved with locks. With no strain on the beacon chain other than regular shard linking.

Steps:

- Slot N-1

: Start tx. Lock resource on shard A, with conditions: * Commit that a section of tx data must be succesfully processed on shard B the next slot. Stage the change, resource can not be read/written while staged.

- Commit that a section of tx data must be succesfully processed on shard B the next slot. Stage the change, resource can not be read/written while staged.
- Slot N

: whoever continues the transaction on Shard B can simply make it check if shard A had the necessary resource locked on Slot N-1

, with the right modification to the resource in the state. (thanks to fast crosslinking). And no locked resources on shard B.

- Shard A staged change is:
- Persistable by anyone who can proof that the tx data was successfully processed within time.
- Undoable by anyone who can proof it was not
- Persistable by anyone who can proof that the tx data was successfully processed within time.
- Undoable by anyone who can proof it was not
- If it times out, does not continue on B, or continues on B without A being linked into view of B, it will atomically fail.
- The remaining staged change on A can be fixed by anyone at any slot after N-1

who needs the resource.

Now for multiple shards / more complex transactions, you abstract everything into staged changes:

- On every touched shard, stage the change and lock the resources for some special finalizing tx (on any shard)
- One slot later, the locks are all checked and the finalizing tx is completed successfully or not.

- Stages are either all committed or none at all, based on the finalizing tx.

I think this is a step in the right direction to:

- Not focus on storage. EEs are all very different, and resource locking can have so much more useful meaning.
- Some simple time element to easily avoid long locking. It's to the EE to decide on the right time default (or requirement) here for its users.

New proposal: capabilities

TLDR: Yes, lots of similarities with the receipts approach, but a minimal extract from existing non-receipt concurrency approaches, to have it translate better into locking/async programming, and drop the storage/balance thinking that is holding us back from making it work for general EEs.

Intro

First of all, "capabilities" are a fun but maybe bit obscure pattern of creating "unforgeable" objects. And in some contexts, they can be revoked by the creator. And then there are more variations.

Key here is that this is very tiny concurrency building block that is designed for "isolates": systems running completely separate from eachother, no shared memory, only learning about the others through user inputs / message passing, with messages built from object screenshots. Sounds familiar, eh?

And best of all, it's (albeit not that extensively) implemented in one of my favorite programming languages, Dart: [dart:isolate

](https://api.dartlang.org/stable/2.6.1/dart-isolate/dart-isolate-library.html). Isolates are similar to Elixir processes, or javascript webworkers. An isolate is single-threaded and has its own memory.

Dart is very minimal, in just providing a factory for capabilities, and not attaching any properties to them. Unlike reference-capabilites, such as in the Pony programming language. Quite an obscure language, but type-safe, memory-safe, exception-safe, data-race-free and deadlock-free (There is a paper with proofs).

Now although the properties by Pony are impressive, and conceptually also very interesting, it is not as easily ported as something as minimal as Dart object capabilities, and probably too opinionated. However, reference capabilities could be fun for a safe but super concurrent EE later down phase 2.

Also note that the minimal capabilities are not only globally unique and unforgeable, they are also unknowable except when passed to an isolate through a message. In a blockchain context it makes more sense to minimize message passing by just querying some protected state, but the unforgeable and unique properties can be preserved.

For more safe-concurrency conceptual gold, this article has a nice comparison

Capability definition

Now, let's define our own eth2 flavor capability:

- (shard, EE)

pairs are the actors in the system

- A capability is owned and maintained by an actor
- Either exists as part of some sparse merkle set (experimental SSZ definition) maintained by the actor, or not. Just need a root for each EE embedded in the shard data each slot to check against.
- Unforgeable

by other actors. A capbility is allocated with a special function. This hashes some desired capability seed v

with the creator to define the capability identifier: H(v, (shard, EE))

. Repeated allocation calls for v

just return the existing capability, unchanged.

- Revocable

by the owner actor. And revocation will only be effective after

the slot completes. The commitment made with a capability to other shards must uphold while those shards can't see what is happening.

- Has an implicit timestamp

: it is allocated in a shard block at slot t

, and it not existing at a prior slot x

(x < t

) can be proven by looking at the accumulated capabilities in x

.

- compositional

: on creation, the seed can be another capability. Deterministically identified by H(c, H(dest_shard, dest_EE))

, where c

is the capability that is created (H(v, (shard, EE))

). There is no moving of capabilities, only committing to those of other actors.

- publicly viewable

: Any actor can check if capability H(x, H(target_shard, target_EE))

exists at some (target_shard, target_EE)

at the previous slot (thanks to fast crosslinking). Note that historic capability tracking can be free: simply refer to the accumulating root at that given slot, instead of the latest root.

So not an object capability, not a reference capability, let's dub it the "commit capability"

.

Required EE host functions: allocate_capability(seed)

, revoke_capability(id)

, check_capability(id)

One slot locks

Now, upgrade the locking idea to use capabilities:

- Slot N-1

: Start tx. EE marks a resource as locked by some capability lock = H(H(res ID, nonce), H(shard A, EE))

. * Commit that a capability H(lock, H(shard B, EE))

will exist on slot N

. Stage the change, resource can not be read/written while staged.

- Commit that a capability H(lock, H(shard B, EE))

will exist on slot N

. Stage the change, resource can not be read/written while staged.

- Slot N

: whoever continues the transaction on Shard B can simply make it check if shard A has capbility lock

on Slot N-1

, with the right modification to the resource in the state (thanks to fast crosslinking), and that the current slot is N

. The EE in Shard B produces a capability unlock = H(lock, H(shard B, EE))

to declare success.

- Shard A staged change is:
- Persistable by anyone who can proof that capability unlock

at slot N

exists.

- Undoable by anyone who can proof it does not exist.

- Persistable by anyone who can proof that capability unlock

at slot N

exists.

- Undoable by anyone who can proof it does not exist.

- If it times out, does not continue on B, or continues on B without A being linked into view of B (i.e. B cannot see the lock

), it will atomically fail: A will not be able to persist the staged change, and B aborts.

- The remaining staged change on A can be fixed by anyone at any slot after N

who needs the resource, as it's public access to check the existence of the unlock

capability. (A smart EE does not require state changes to deal with expired capabilities)

Synchronous but deferred

Essentially the same as deferred receipts. However, abstracting away state / merkle proofs. The EE can design that. The EE is just provided a function to register and check capabilities.

Semantically synchronous, but deferred change:

- Slot N

: EE on shard A stages a change by unregistered

capability syn = H(H(res ID A, nonce), H(shard A, EE))

. To be persisted if synack

can be found, and when found also register ack

.

- Slot N

: EE on shardB stages a change by registering capability synack = H(H(res ID B, syn), H(shard B, EE))

. To be persisted if ack

can be found.

A simple syn-ack does the job here. (first syn

does not have to be registered, using it as part of synack

is good enough)

This can all be done in the same slot, as the execution is deferred to a later slot where the shards can learn about the capabilities published by eachother.

Chaining changes in the same slot

Simply make the EE add aditional persistence conditions when working on unconfirmed resources: last_cap

(ack

of previous transaction modifying the resource of interest) needs to be registerd too. Nothing is blocking, it essentially just optimistically runs the stacking transactions, defers evaluation for a slot, to be then lazily persisted. And best of all, the EE can program the chaining however it likes, and separate resources will not affect eachother unless used together in an atomic transaction.

Similarities/difference

A lot of the space is explored, the challenge really is to iterate well, don't opinionate it, and keep it bare minimum but powerful on protocol level.

The ability to do an existence check is similar to receipts; just provide a merkle-proof for cross shard data. However, important here is that we should only be looking to standardize the keys, not the values: a capability can be a hash or other compression of any data. It's not about the receipt contents, it's about the boolean property of existence of a given key. The remainder follows however the EE likes it.

So now we translated the receipts/locks/timeout/logging ideas into simple unforgeable objects, completely agnostic to EE style or state approach, that can build many other patterns too.

Building EE concurrency patterns

Simple Asynchronous Calls (async await, callback, etc.)

- Define a TX as a function invocation chain as {shardX, EE_Y}.then(() => ...)

(for success), {shardX, EE_Y}.onError(() => ...)

(for failure), {shardX, EE_Y}.finally(() => ...)

(after either success or failure completes) structure. Each chain call can be in a different shard/EE.

- The modification of every part P

in the chain is only persisted if a capability with the decision path input that describes the follow-up path is registered: * Path result capability is recursively defined as: * success: x = H(cap_then, 0)

when then

completes with success.

- error: x = H(0, cap_error)

when then

does not complete with success

- cap_then

and cap_error

are the result capability of the respective execution paths.

- success: x = H(cap_then, 0)

when then

completes with success.

- error: x = H(0, cap_error)

when then

does not complete with success

- cap_then

and cap_error

are the result capability of the respective execution paths.

- Persisting modifications:
- then

persisted when H(x, 0)

- onError

persisted if defined and when H(0, H(x, 0))

(the error handler must be successful)

- finally

persisted if defined and for any registered outcome, i.e. any of the H(a, b)

options.

- tx fails (no-op) if the onError

branch was not covered but hit H(0, H(0, x))

, and no finally was declared.

- then

persisted when H(x, 0)

- onError

persisted if defined and when H(0, H(x, 0))

(the error handler must be successful)

- finally

persisted if defined and for any registered outcome, i.e. any of the H(a, b)

options.

- tx fails (no-op) if the onError

branch was not covered but hit H(0, H(0, x))

, and no finally was declared.

- Path result capability is recursively defined as:
- success: x = H(cap_then, 0)

when then

completes with success.

- error: x = H(0, cap_error)

when then

does not complete with success

- cap_then

and cap_error

are the result capability of the respective execution paths.

- success: x = H(cap_then, 0)

when then

completes with success.

- error: x = H(0, cap_error)

when then

does not complete with success

- cap_then

and cap_error

are the result capability of the respective execution paths.

- Persisting modifications:
- then

persisted when H(x, 0)

- onError

persisted if defined and when H(0, H(x, 0))

(the error handler must be successful)

- finally

persisted if defined and for any registered outcome, i.e. any of the H(a, b)

options.

- tx fails (no-op) if the onError

branch was not covered but hit H(0, H(0, x))

, and no finally was declared.

- then

persisted when H(x, 0)

- onError

persisted if defined and when H(0, H(x, 0))

(the error handler must be successful)

- finally

persisted if defined and for any registered outcome, i.e. any of the H(a, b)

options.

- tx fails (no-op) if the onError

branch was not covered but hit H(0, H(0, x))

, and no finally was declared.

- async

/await

/async-try

are just syntax sugar for then

and onError

Message Driven Approach (Actor model)

Lots of options here, but capabilities are essentially the unforgeable objects to authorize messages between EEs at low cost. That could mean a capability based memory safety model like the Pony language has. Or simply focus on messages that claim existence of capabilities unique to the message, to authorize async changes.

Two Phase Commit (Wide Atomicity)

Commit request phase: one transaction to all EEs to stage a change, that will be locked in and persistable if a certain capability is registered in the future (and optionally with a timeout to free resources when no action is taken).

Commit phase: Notify all participants that everyone is set (staged the changes), and publish the capabability to force participants to persist the staged change everywhere eventually (or keep it staged until someone actually needs the resource, but never drop the change).

Locking — Read/Write

- Disable reads and/or writes for a certain resource until a capability is published
- Re-enable locks when the capability is revoked.
- Optionally the inverse: temporary enable things when a capability is revoked.

Contract Yanking

- Yank a contract by:
- Register it as yanked in its old shard, declare your foreign (it will be on the yank destination shard) promised but unpublished capability to unyank it. And possibly with a deadline to unyank (contract is free to specify other conditions too: e.g. incentive to be yanked to certain shards).
- Others can queue their planned changes by building on your promised unyank capability.
- Make changes to a copy on your preferred shard.
- Publish the Unyank. The EE may make choose if this is a passive option (anyone can unyank when you are done after a certain action), or if you must do it yourself. The original contract is required to load the state of the new contract when the unyank frees it.

- Register it as yanked in its old shard, declare your foreign (it will be on the yank destination shard) promised but unpublished capability to unyank it. And possibly with a deadline to unyank (contract is free to specify other conditions too: e.g. incentive to be yanked to certain shards).
- Others can queue their planned changes by building on your promised unyank capability.
- Make changes to a copy on your preferred shard.
- Publish the Unyank. The EE may make choose if this is a passive option (anyone can unyank when you are done after a certain action), or if you must do it yourself. The original contract is required to load the state of the new contract when the unyank frees it.

Building more patterns

Capabilities are nice for minimal concurrency legos, but also for other use-cases.

Permission systems

Instead of having to make every single EE compatible to read eachothers state (lots of duplicate EE code!), capabilities could be the “permission lego” we need. A simple “commit

” system to run with. Not conceptually new, but simple yet powerful.

- Classic owner permission, but cross-shard: H("address 0xabc... is owner of resource X", H(shard, EE))

where (shard, EE)

is the host of the address

.

Now visit another shard, and say “here’s this address you need to trust as owner, check it”, and then the EE checks the existence of the capability.

- Time-out permission: H("can do X until block N",

H(shard, EE))`

Permissions based on commitments are really that easy, why not?

To be explored

- Capability hash construction:
- non-EE specific capabilities: H(v, H(shard, 0))

. And fail on creation if it already exists.

- non-EE specific capabilities: H(v, H(shard, 0))

. And fail on creation if it already exists.

- Location of capabilities root in the state. Could go as root of a Merkle set into ShardState

.

- The approach of staging changes based on capability read expectations is up to the EE. UTXO EEs can use capabilities to make outputs spendable and inputs unspendable, no need for staging

. Cross-shard atomic instant UTXO transfers, why not?

- A very common pattern is make a commitment to do something when a capability from some other actor is seen, and include creating a new capability to trigger something else (e.g. the syn-ack handshake). Chaining capabilities may be a good protocol feature, to not have this as boilerplate in an EE.

- E.g. a commit capability may not have to be a Yes/No when checking, but could be a Yes, No, or Yes/No derived from the equal/inverse of some other capability. Basically indirect commitments.

- Could be very nice to improve the async then/onError

commitment dance with.

- No need to run code to fire a commitment based on some other commitment, making things run faster

- E.g. a commit capability may not have to be a Yes/No when checking, but could be a Yes, No, or Yes/No derived from the equal/inverse of some other capability. Basically indirect commitments.

- Could be very nice to improve the async then/onError

commitment dance with.

- No need to run code to fire a commitment based on some other commitment, making things run faster