I'd like to argue against a widespread perception that under the initial sharding protocol, the only way to do cross-shard transactions will be with asynchronous messages (i.e. contract locking/yanking, and so on). I do agree that if we expect the least amount of innovation from protocol engineers, then asynchronous transactions are what we'll get (asynchronous transactions are the solution if protocol engineers shirk it off and leave it as a problem to be solved by users, contract authors and dapp devs). But it would be a shame to accept a "worse is better" fate. Instead, I want to highlight some insight why solving the problem at the protocol layer, enabling synchronous cross-shard transactions that would work with no change to the current DX/UX (so that transactions which touch multiple contracts would continue to work for dapp devs on Ethereum 2.0 as they do now on Ethereum 1.0), should actually be easier than is commonly perceived.

This insight is not new, I have shared it before with a few people offline (you know who you are

) as early as the week before Devcon2. And even then it was arguably not novel / fairly obvious. I've since seen some trickles of it come through in posts here and there. First, in the Sharding FAQ:

How would synchronous cross-shard messages work? […] one fairly simple approach can be described as follows:

- A transaction may specify a set of shards that it can operate in
- In order for the transaction to be effective, it must be included at the same block height in all of these shards.
- Transactions within a block must be put in order of their hash (this ensures a canonical order of execution)

Then later, on the forum:

We know that with merge blocks and clever use of fork choice rules 47, we can have a sharded chain where a block on one shard is atomic with a block on another shard; that is, either both blocks are part of the final chain or neither are, and within that block there can be transactions that make synchronous calls between the two shards (which are useful for, among other things, solving train-and-hotel problems 25). […]

It is also worth noting that merged state execution technically does not require merge blocks. Instead, one can simply define a deterministic function that assigns shard IDs into pairs at each block number (eg. if we want to cycle evenly, and there are k shards and k is prime, then at block number N shards, shuffle shard IDs with i -> (i * N) % k and then merge-execute blocks 1 and 2, 3 and 4, etc using post-shuffling IDs). Merge-executing two blocks would simply mean that a transaction starting on one block is capable of reading and writing state on the other block.

And again, positioned favorably even:

Cross Shard Locking Scheme - (1) [

Sharding

](/c/sharding)

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. I think the best solutions to these problems are those solutions t…

I think the best solutions to these problems are those solutions that recognize that shards are virtual galaxies, and not physical sets of computers, and so the nodes executing two shards can, eg. with stateless client witness protocols, temporarily be merged for one block.

Despite the occasional hint that synchronous cross-shard transactions might be quite practical/maybe-not-too-difficult, they are still under-appreciated as a killer feature of sharding (perhaps because of a confused oversimplification that asynchronicity == parallelism and parallelism == good. more on that later).

Even among the ewasm team (where most of my contributions are directed lately), as we've begun shifting our focus from an ewasm 1.0 prototype testnet to an ewasm 2.0 execution engine for shards, any mention of "sharding" is quickly followed by questions about "asynchronous calls", much to my dismay. I promised a write-up pushing back against the insidious complexity of asynchronous calls, which if allowed to accumulate would form an evil asynchronous temple of doom (filled with footguns), and can postpone it no longer.

Transactions: myths, surprises and opportunities

The insight I want to highlight starts with a talk by Martin Kleppmann about transactions. In particular, the paragraph beginning around 29m34s:

If you want serializable transactions across multiple services, you would have to run some kind of atomic commit protocol. These are distributed systems protocols, which make sure that if you've got various different parties who are all participating in a transaction (various different services), either all of them commit or none of them commit. Protocols such as two-phase commit, three-phase commit, transaction managers, etc., they implement this kind of thing. However, as we saw earlier, the whole idea of serializability is that transactions appear as if they were in a total serial order. You can execute literally in serial order or not, but that order is definitely defined, so whenever two transactions conflict, somebody has to decide which of the two came first. From a distributed systems point of view, this makes it effectively an atomic broadcast problem. That is, the problem of getting a bunch of messages to different nodes in exactly the same order. But the issue with atomic broadcast is that, from a theoretical and also practical perspective, it's actually equivalent to consensus.

Here "services" are shards, and "protocols such as two-phase commit" are the contract locking/yanking ideas for achieving asynchronous cross-shard transactions (essentially the same as cross-chain atomic swaps, or "cross-chain relays/bridges", or two-way pegs with SPV proofs, etc.). Kleppmann was speaking about "microservices" and how you should try to avoid doing transactions across microservices if at all possible, because if you want to do them right then you'll end up implementing some protocol that's equivalent to a consensus protocol. Well, in our case we can't avoid doing global transactions (across services), because its the only way to prevent double-spends. But also in our case, we're already doing consensus. That means we already have a total order for transactions. So there's no need for two-phase commit protocols between contracts on different shards.

Consolidating concurrency control and consensus

This same insight was articulated in a 2016 paper, the Janus paper (Consolidating Concurrency Control and Consensus for Commits under Conflicts):

Conventional fault-tolerant distributed transactions layer a traditional concurrency control protocol on top of the Paxos consensus protocol. This approach provides scalability, availability, and strong consistency. When used for wide-area storage, however, this approach incurs cross-data-center coordination twice, in serial: once for concurrency control, and then once for consensus. In this paper, we make the key observation that the coordination required for concurrency control and consensus is highly similar. Specifically, each tries to ensure the serialization graph of transactions is acyclic. We exploit this insight in the design of Janus, a unified concurrency control and consensus protocol.

[…] the layering approach incurs cross-data-center coordination twice, in serial: once by the concurrency control protocol to ensure transaction consistency (strict serializability [34, 45]), and another time by the consensus protocol to ensure replica consistency (linearizability [14]). Such double coordination is not necessary and can be eliminated by consolidating concurrency control and consensus into a unified protocol.

The key insight of Janus is to realize that strict serializability for transaction consistency and linearizability for replication consistency can both be mapped to the same underlying abstraction. In particular, both require the execution history of transactions be equivalent to some linear total order.

Applied to sharding, the insight is that the master chain (consensus) provides a total order over "shard blocks". This total order of shard blocks means we get an implicit total order of cross-shard transactions (concurrency control), for free. By taking advantage of the total order determined by consensus to coordinate cross-shard transactions in the protocol layer, we don't need to do locking/yanking in the contract layer. In other words, coordinating cross-shard transactions through asynchronous calls is redundant "double coordination" that can be eliminated.

Chain of chains, or sharded chain?

Contract locking/yanking, where a contract on one shard communicates with a contract on another shard by verifying tx receipts from the other shard, is just another form of two-phase commit. The sharding protocol might simplify this receipt verification, making it easier to verify receipts from a shard versus verifying receipts from a completely different blockchain. In order to verify receipts from a completely different blockchain, the contract itself also needs to process and verify the headers of the other chain (this is how Peace Relay works, for example). In general, this is the direction taken by Interledger, Cosmos, Polkadot, and probably other "chain of chains" aspirants.

Because two different blockchains have their own consensus processes, each determining the order of their own chain's blocks and not the other's, two-phase commit protocols like cross-chain atomic swaps are necessary. But if "shard chains" are under one master consensus process that determines the order of all shard blocks, then two-phase commit between shards is not necessary. The order over shard blocks implicitly coordinates the concurrency of all cross-shard transactions, providing synchrony for free (again, coordination through locking/yanking would be needless "double-coordination").

Chain Fibers, with delayed execution

Note that efficiently coordinating cross-shard transactions requires knowing ahead of time which shards each transaction may access. In the Janus paper, this access list is called the "dependency list".

The basic idea was already present inChain Fibers. In Chain Fibers, a total order of cross-shard transactions is forced by forming shard blocks over "non-overlapping" sets of fibers (X-fiber blocks), ensuring that no cross-shard transactions conflict. You could allow X-fiber blocks with overlapping sets, and resolve conflicts by some total canonical order determined by the master chain, but then there's a worry that the overhead of data communication (as opposed to coordination for concurrency control) between different fiber sets would defeat the goal of processing fiber blocks in parallel.

A note about parallelization: if the goal of scaling is merely to process more token transfers, then we don't need synchronous cross-shard calls. Each shard can be isolated and each token contract can live on a different shard. But for other use cases, like lotteries, or decentralized exchange, or cryptokitties, and so on, all the locking/yanking between different shards would cause a parallel slowdown. For these other use cases, the goal of sharding is to partition the state but preserve the global transaction boundary (i.e. to enable synchronous cross-shard transactions, as the transaction load is inherently serial and cannot be parallelized).

Back to cross-shard data communication: iterate on Chain Fibers by de-coupling consensus from state execution. Do a consensus game only on data availability and the order of shard blocks (i.e. “data blobs”), and do state execution as a separate delayed process. Then, it seems feasible to allow shard blocks with overlapping access lists and do “merged execution”, as long as the data coordination that happens during merged execution can keep up with the rate at which shard blocks arrive. If the clients are stateless then I’d say it can, because the data needed is already being propagated in the form of merkle proof “witnesses” included in transactions.

It is this merged execution which allows for synchronous cross-shard transactions, i.e. cross-shard transactions using the same synchronous CALLs of Ethereum 1.0 that we know and love (yes, there are some counter-intuitive aspects of EVM 1.0 CALLs that we don’t love, such as contract re-entry. But these issues are orthogonal to synchrony versus asynchrony; there are other ways to fix them without throwing out synchrony and just wholesale drinking the async kool-aid, which I’m afraid will induce a trip that is not so great. Cue Kleppmann: “Every sufficiently large deployment of [asynchronous shards] contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of transactions.”).