ParallelChain F

Draft 1 (WIP)

Digital Transaction Limited

June 16, 2021

Contents

1 Outline

This paper introduces ParallelChain F, a distributed, permissioned smart contract platform that uses modern Byzantine Fault Tolerant (BFT) State Machine Replication (SMR) techniques and careful optimizations to achieve high throughput and low latency in clusters with tens of nodes, and acceptable throughput and latency in clusters with hundreds of nodes.

We begin by rationalizing the design choices we made in ParallelChain F by contextualizing Bitcoin, Ethereum, and other blockchains (including ours) in the history of distributed systems research (§2). We then describe the system design (§3), walking through the lifecycle of a transaction from proposal to commit for clarity (§4.1). Digital Transaction's XPLL, our flagship ParallelChain F network, is described in §5.1. Optimizations we are considering for inclusion in future versions of ParallelChain F, including (most promisingly) execution sharding, are discussed in §6. Finally, §7 concludes.

2 Background

2.1 The Byzantine Generals Problem and Nakamoto Consensus

Computers can be thought of as state machines: objects that deterministically transition between a set of possible states in response to input. More precisely, a computer can be modelled as a 3-tuple: (s, S, F), wherein $s \in S$ is the computer's current state, S is the set of states the computer can be in, and F is a set of state transition functions $f : (s1 \in S, m) \rightarrow s2 \in S$. Here, m (message) stands for anything 'perceivable' by a computer. m to an Apollo Guidance Computer might be an altitude reading from an Apollo LM's radar altimeter, m to your smartphone might be a push notification from a messaging application; m to a Bitcoin full node might be a BTC transfer from Jane to John.

Over its lifetime, a computer will receive and execute a long sequence of messages, and since computers are deterministic, multiple identical computers starting off with the same initial state will end up in the same final state after executing an identical sequence of messages. The computer science community has, since as early as the mid-1970s, sought to exploit this property to do *State Machine Replication* (SMR). One of SMR's original applications of interest was to have many geo-distributed computers implementing the same state machine act as if it was a single super-computer, one that was available to users living in far-apart places, and tolerant to fail-stops (total machine failure; computers suddenly going silent) caused by adverse events like natural disasters and power outages. Having computers that are deterministic is helpful for SMR, but not sufficient. In addition, one also needs a way to replicate the sequence of m across these computers. In a world where network and processing latencies are not predictable, this problem turned out to be non-trivial: if the message log was a set, and message order didn't matter, any kind of reliable broadcast would suffice. But message order does matter, and two machines separated by a vast ocean could easily receive a pair of messages in the opposite order.

The problem of replicating an ordered message log is an instance of the general problem of *consensus*: getting multiple replicas to agree on a value. In 1978, now-Turing Laureate Leslie Lamport offered a simple algorithm for SMR that used logical timestamps and clock synchronization to solve the message order consensus problem, but assumed a synchronous network (one where message delays are bounded); an unreasonable assumption in a global best-effort network like the internet. Just as significant, Lamport's algorithm could not tolerate fail-stops. An indication of the difficulty of the problem is that no truly practical, fault-tolerant algorithm would emerge for SMR that guarantees safety (logs do not diverge) and liveness (the algorithm eventually makes progress) in partially synchronous networks until Lamport's Paxos [5] in 1998[1].

Concomitant to work on fail-stop-tolerant consensus algorithms, the distributed systems community also worked on consensus algorithms that work in the more difficult Byzantine failure model. Byzantine faults (again, Leslie Lamport's definition) [4] include any arbitrary node failure, including the case of nodes saying different things to different nodes (duplicity), appearing to be alive to some members of the network but not others, and so on. We can here note two things: 1. The Byzantine failure model is strictly harder than the fail-stop fault model, i.e., a fail-stop failure is also a Byzantine failure, and 2. The Byzantine failure model captures the case of nodes acting in the control of malicious adversaries, who seek to (for whatever reason) damage the safety and liveness properties of the network. The first 'practical' algorithm for SMR in the Byzantine failure model [6] (aptly called Practical Byzantine Fault Tolerance) was offered by Barbara Liskov[2] and her then-PhD

[1] Lamport actually tried to get Paxos published in 1990, but his paper, which used an esoteric allegory to parliamentary proceedings in a fictional Greek island as a didactic tool, was misunderstood as a joke by the editors of TOCS.

[2] Also a Turing Awardee, for her work on abstraction and Programming Languages.

student Miguel Castro in 1999.

If you have followed so far, it might seem that the problem of consensus is a settled one. What is then, you might ask, so novel about Bitcoin's Nakamoto Consensus (Proof-of-Work) [1] algorithm that justifies it being hailed as a revolutionary technology on the cusp of disrupting the whole notion of society as we know it? A few more technically-inclined observers in the business community have suggested that Nakamoto Consensus is a solution to the Byzantine Generals problem [2]. You should know by now that even if it is[3], that fact alone isn't something noteworthy. The full truth is this: since all 'classical' solutions to the Byzantine Generals problem (e.g., PBFT) rely on some kind of voting mechanism, they are vulnerable to Sybil attacks (malicious actors creating many false identities to sway votes in their favor) *unless* they restrict network membership. Precisely speaking, what Nakamoto Consensus is is the first solution to the Byzantine Generals problem that works in the fully public, permissionless membership model.

The fact that 'mining' Bitcoin is highly lucrative and BTC is seen by lay-people as an investment product is a mere side effect of its central role in Nakamoto Consensus. Mining works with hash-chaining to create an incentive structure for mutually-distrusting, anonymous node owners to maintain a globally consistent transaction log, which encodes *exactly* how much money an identity controls at any particular block height. Bitcoin is not mere e-cash. In the case of PayPal, all users of the service needs to trust PayPal Holdings, Inc. to maintain its transaction logs honestly, and resist both the urge to to dishonestly benefit itself and its partners, and the demand by governments and other powerful institutions (or individuals) to censor transactions, seize funds, and infringe on privacy rights. With Nakamoto Consensus, Bitcoiners need not trust anybody.

2.2 Ethereum - the World Computer

For being the first algorithm to solve an extremely general problem in computer science, Nakamoto Consensus' initial flagship application, Bitcoin, seemed disappointingly limited, especially to Waterloo dropout Vitalik Buterin and his collaborators on Ethereum. Limitations of the Bitcoin Script programming language makes it impossible for the state machine replicated by the Bitcoin network to support anything other than basic financial applications like escrow and insurance. *Prima facie*, there is nothing necessary about these limitations in Bitcoin Script. We have been creating and using Turing-complete programming languages since the 1930s. The key problem with having a public, Turing-complete, replicated state machine is a subtle liveness problem: since the Halting Problem is undecidable, there is no easy way to prevent attackers (or bad programmers) from deadlocking the Bitcoin network by publishing a Bitcoin Script program that never terminates. Nakamoto either wasn't interested in a global state machine for general purpose computing, or gave up trying to solve the liveness problem and just constrained Bitcoin Script to be Turing-incomplete.

The major innovation of Ethereum are gas fees, which like mining (which Ethereum also has) can be understood as a technical solution with financial side-effects. Gas limits force

[3] Strictly speaking, since blocks in Bitcoin are never *truly* final, Nakamoto Consensus is not BFT. Practically, however, finality is almost certainly guaranteed after 5-6 blocks.

long or provably non-halting programs to terminate when they run out of gas, solving the liveness problem. It also incentivizes application programmers to write compact, efficient smart contracts, giving the network a sizeable performance boost at the execution layer. It is hard to understate how much having a Turing-complete programming language expands the design space for smart contracts: it is a mathematical result that every computation expressible in a 'conventional' programming language like C, Java, or Python is also expressible in Solidity. In theory, applications as disparate as escrow, calculating π to an ludicrous level of precision, and Pokemon Emerald can be developed and run by the Ethereum state machine.

The key words are *in theory*. In reality, the most popular applications on the Ethereum network are, like the applications on the Bitcoin network, financial applications. People can speculate about the reasons, but two likely explanations are: 1. The transaction fees of the Ethereum network are just too high to justify applications that are not (potentially) financially lucrative, and 2. The Ethereum state machine is too slow. ParallelChain F attempts to solve both problems.

2.3 'Nobody cares about decentralization' - or, the comeback of classical BFT

Vitalik Buterin identified a set of tradeoffs in blockchain design that he calls the Blockchain Trilemma. The three properties in the Trilemma, taken directly from [9], are:

- Scalability: the chain can process more transactions than a single regular node (think: a consumer laptop) can verify.
- Decentralization: the chain can run without any trust dependencies on a small group of large centralized actors. This is typically interpreted to mean that there should not be any trust (or even honest-majority assumption) of a set of nodes that you cannot join with just a consumer laptop.
- Security: the chain can resist a large percentage of participating nodes trying to attack it (ideally 50%; anything above 25% is fine, 5% is definitely *not* fine).

As with most other industries competing in free markets, it is unlikely that one blockchain network will be conceived that pareto dominates all other blockchain networks. The most pessimistic interpretation of the Trilemma is that any blockchain system needs to make unpleasant compromises with at least one of the properties. More optimistic interpretations claim either that: 1. Near/medium-term innovations will enable blockchain networks that are maximally scalable, decentralized, and secure, or 2. Some of the properties in the Trilemma are overrated, and do not bring significant value to a blockchain network.

We are optimistic on both counts. On 1., we believe that sharding, designed properly, will allow maximally decentralized (permissionless) blockchains to scale somewhat in the medium-term (§6.3). On 2., we believe that users do not see complete anonimity and permissionless membership (ala Bitcoin) as the primary desirable aspect of decentralization. This has a major design consequence: it allows us to use energy-efficient, fast, 'classical'

BFT consensus algorithms instead of Nakamoto Consensus or even epochal PoS as used in Tendermint-based systems like Cosmos. As a result, a modest ParallelChain F network can achieve much higher throughput and lower latency than the fastest 'traditional' blockchain networks, with the corollary that the network can remain profitable for its members even whilst exacting very low transaction fees from its users.

An empirically verifiable fact that supports our stance on 2. is that the vast majority of people do not subscribe to the strong crypto-anarchist vision of governmentless or nearly-governmentless society. Public dissatisfaction with currently ruling governments and successful corporations should be seen as an indictment of said governments and corporations, and not the idea of government, central banks, and 'trusted third-parties' wholesale. ParallelChain F is intended to deployed to form networks with a large, stable, and KYC-ed *validator* membership. In these networks, it provides a technological framework to connect together a large number of known parties with orthogonal financial interests, who are strongly incentivized to act honestly to maintain a global, Turing-complete smart contract platform. In case some validators do decide to collude to fool the network, our choice of fast BFT consensus protocol, HotStuff [11], prevents them from succeeding and records evidence of their malicious actions.

In a strong sense, then, ParallelChain F falls under the same class of systems as Cardano, Solana, and Diem, though we of course believe that it is a superior system.

3 Basic design

3.1 Design goals

Early on in the design process, we defined a set of properties that we deem any useful public[4] smart contract platform *must* have, even before optimizing for the properties in the Blockchain Trilemma:

- Local log immutability: the software running on validator node should be programmed such that it can only append to its blockchained transaction log. Optimally, the software should also have the ability to detect tampering of its local blockchain files.
- Consistency: different validator nodes' transaction logs must not diverge. If a smart contract platform uses a leader-follower model, these should communicate using a protocol that is resistant to Byzantine faults. Consistency failures are precursor to double spending attacks.
- Validity: validity is an application-layer concept. A transaction is valid *if and only if* its read and write sets are generated by a smart contract invocation on a 'correct' world state. As Ethereum Classic advocates say: code is law.

[4] We apologize if our loose and sometimes interchangeable use of the terms 'public' and 'permissionless' is confusing to the reader. To clarify: membership in a ParallelChain F network's validator set is permissioned, but members of 'the public' (where the scope of the public differs according to network) can deploy and call smart contracts on the network.

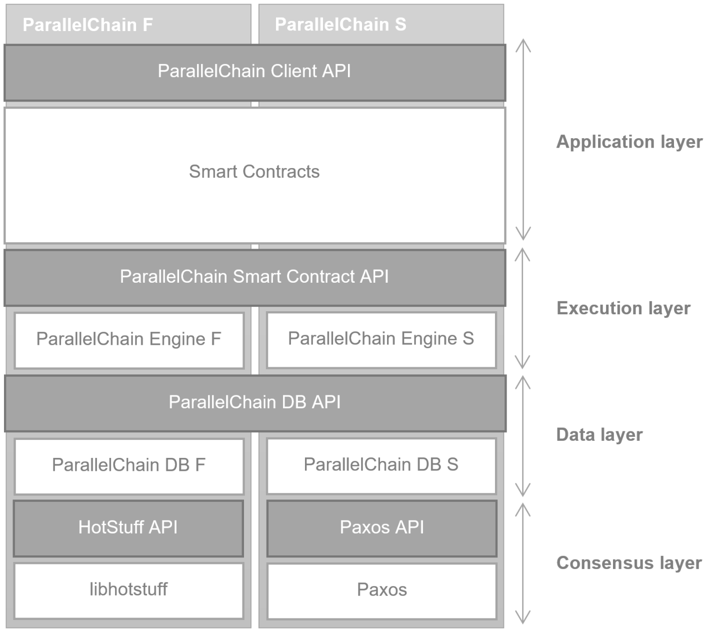


Figure 1: The Layered Architecture of ParallelChain

- Liveness: the network should be able to make progress with minimal downtime, even in the face of DoS attacks and frequent membership changes.

Additionally, since we designed ParallelChain F concurrently with XPLL, we also agreed on the principle of generality:

- Generality: XPLL should be a layer-2 system implemented on top of ParallelChain F. We must not force features into ParallelChain F exclusively to enable XPLL.

In the future, we hope to field ParallelChain S and ParallelChain F as complementary products for networks with different desired characteristics: the former offering distributed-database-like scalability and blockchain immutability in trusted enterprise networks, the latter offering good throughput and latency in minimal-trust permissioned networks.

3.2 Layered architecture

The ParallelChain architecture (Figure 1) can be decomposed into four layers:

- Application layer: smart contracts written and deployed on the network by users (including non-validator node operators), and the client (desktop, mobile, web, embedded, etc.) software that invokes those smart contracts.

- Execution layer: a program (ParallelChain Engine) that runs smart contracts in response to client requests in an isolated environment to control or outright prevent their access of sensitive OS functions like networking and file system, and abstracts their access to state.
- Data layer: a program (ParallelChain DB) that writes the blockchain into a validator node's filesystem and maintains a 'stable state cache': a disk-oriented key-value store (in ParallelChain's case, FoundationDB) that reflects the replicated state machine's state at a particular block height.
- Consensus layer: a library that implements a consensus algorithm (HotStuff in the case of ParallelChain F, and Paxos in the case of ParallelChain S) that is used by the data layer process to keep the blockchain consistent in honest replicas.

3.3 Smart contracts

Currently, ParallelChain supports smart contracts written in the Go programming language. We do not consider adding multiple language support an immediate priority, but if we do add support for other languages in the future, this is likely limited to high-performance compiled languages (e.g., C/C++ and Rust) and Solidity (if we fail to design a gas calculation scheme for Go smart contracts). We'd rather support one high-performance language correctly than over-extend ourselves by supporting many low-performance languages poorly.

Users write Go source code importing the ParallelChain Smart Contract API package, and then *flatten* their source code using the ParallelChain CLI. *flattening* is a process whereby source code is minified for storage-efficiency and packed into a single self-contained `go.main`, without any import statements (including from the Go standard library). This way, smart contract behavior should remain identical in the long term, even if referenced dependencies change.

The user then deploys the smart contract to the ParallelChain network using the ParallelChain CLI. *deploying* a smart contract is a transaction on the blockchain that includes a `Set()` on a key in a reserved key range. The user provides ParallelChain CLI with a network-unique name and version number for the smart contract. Version number should be a positive integer without 'gaps'. Users invoke smart contracts using their network-unique name and version number, and not the key their source code is stored in. The user is also required to provide the CLI with the minor version of the Go programming language they would like their smart contract to be compiled with, again, to ensure that smart contract behavior is deterministic across validators and remains the same in the long term.

Since the blockchain is fully replicated, every validator node receives a copy of the smart contract source code. When a new smart contract is deployed, a validator node's ParallelCore DB process send a TCP message (via the loopback interface) to its ParallelCore Engine process, passing the latter with the smart contract's raw source code as an argument. The Engine compiles the smart contract into a shared object library (`.so`) file and places it in the node's file system.

The ParallelCore Engine process maintains a separate *execution container* process that has minimal system permissions (no access to network, filesystem, shell, etc.) When a smart contract is invoked, the contract's shared object library is dynamically loaded into the execution container process and the main function is invoked. The smart contract safely communicates with the Engine process through the ParallelChain Smart Contract API. The Engine process restarts the execution container process in case it dies for any reason. In ParallelChain Engine F, smart contracts are invoked serially to prevent state cache divergence.

Sometimes, smart contract logic needs to 'hook' into the consensus layer. Two use-cases that we came across when we designed XPLL was: 1. Getting the public key of the current leader during smart contract invocation, who is entitled to the XPLL transaction fee, and 2. Adding and removing nodes from the network. The ParallelChain Smart Contract API provides functions for these use-cases whose implementations are wrappers for functions in the consensus layer. Typically, network administration actions like 2. should not be undertaken unilaterally. The API for these functions (and other functions with global side-effects) require that the caller passes a quorum (2/3) of signatures from validators that authorizes the specific action to be carried out. Smart contracts implements voting logic themselves.

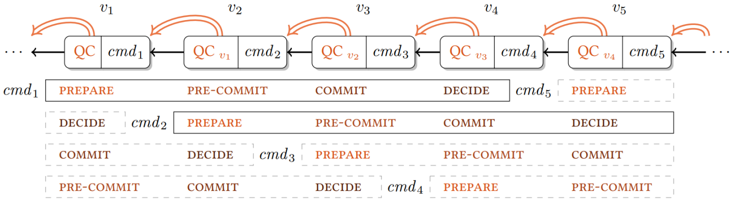
A major vulnerability that remains in this initial release of ParallelChain F is the lack of 'transaction fees' for smart contract invocations by default, meaning the network is vulnerable to deadlocks if smart contracts panic or do not halt. This is not a major problem with permissioned (consortium) deployments of the system, where permission to deploy smart contracts can be restricted, or for the XPLL mainnet, since the XPLL smart contract implements transaction fees at the application layer. An immediate goal for future releases of ParallelChain F is to design a gas fee calculation scheme like the one used in Ethereum. One exciting (but highly speculative) research direction is developing an interpreter to run Go assembly.

3.4 Blockchain and state cache

The state cache in a single ParallelChain F validator node is a single-node (non-distributed) ParallelChain DB instance. Even though ParallelChain DB consists mostly of a disk-oriented database (currently FoundationDB [10]), we refer to it as a 'cache' to make clear the fact that the ultimate authority about the state of the replicated state machine at a particular block height is the blockchain. Strictly speaking, maintaining an up-to-date snapshot of the world state in a database is simply an optimization to speed up reads.

The blockchain is a single, append only file on a validator's file system. TODO: Provide a diagram for block format. I do not like the idea of using the block format in ParallelChain S, since that is very sub-optimal and hacky.

When a smart contract running in an execution container exits, ParallelChain Engine calls the `Commit()` function in the ParallelChain DB API. ParallelChain DB F commits transactions serially to prevent state cache divergence. ParallelChain DB S can safely commit



v_1
v_2
v_3
v_4
v_5
QC cmd_1
QC $_{v_1}$ cmd_2
QC $_{v_2}$ cmd_3
QC $_{v_3}$ cmd_4
QC $_{v_4}$ cmd_5
cmd_1 PREPARE PRE-COMMIT COMMIT DECIDE cmd_5 PREPARE
DECIDE cmd_2 PREPARE PRE-COMMIT COMMIT DECIDE
COMMIT DECIDE cmd_3 PREPARE PRE-COMMIT COMMIT
PRE-COMMIT COMMIT DECIDE cmd_4 PREPARE PRE-COMMIT

Figure 2: Chained HotStuff commit flow

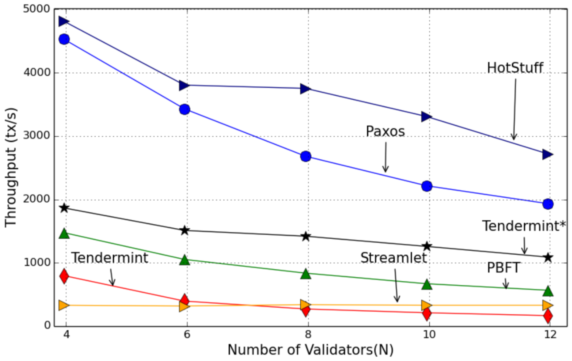


Figure 3: Consensus throughput comparison

transactions in parallel since all full nodes in a ParallelChain S cluster are part of a single physical FoundationDB cluster.

A major limitation for ParallelCore DB currently is FoundationDB's 5-second transaction rule. Again, we not expect this to be a major problem in networks where smart contracts are vetted, but this is an unacceptable problem in public networks and for applications that require long-running smart contracts. Transaction fees (§3.3) will incentivize smart contract developers to write short, focused programs, but the long-term solution is to rewrite both ParallelChain DB F and S to use another key-value store (§6.2).

3.5 Consensus protocol

ParallelChain DB F instances in validator nodes reach consensus on blocks to commit using an optimized version of the HotStuff protocol (§6).

HotStuff [11] is a BFT SMR protocol created by a group of researchers from Cornell, UNC-Chapel Hill, and VMware. It guarantees assuming partial synchrony that all honest nodes in ParallelChain cluster eventually get an identical sequence of blocks, and that a block, once committed at a block height, is never rolled back or replaced by another (unless node operators manually edit their blockchain file, for instance, in a concerted effort to recover

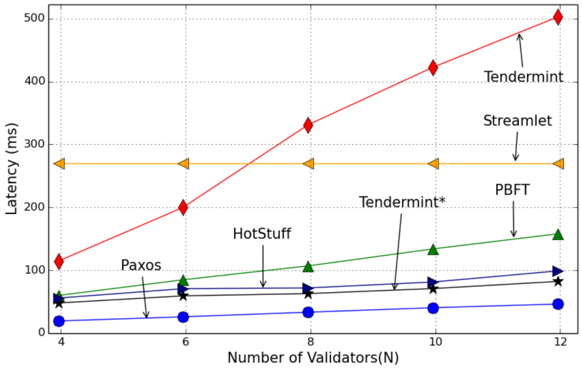


Figure 4: Consensus latency comparison

from an attack). HotStuff maintains liveness when the network in synchronous and at most 1/3 of the network is faulty, and maintains safety (consistency) when at most 1/2 of the network is faulty. We built our optimized HotStuff on top of the libhotstuff implementation made available by the protocol's authors.

HotStuff distinguishes itself from other algorithms in the PBFT family in two important ways:

- **It uses a 4-phase commit flow instead of the more common 3-phase flow.** The authors of HotStuff recognized that having one more phase in the commit flow allows every phase in the flow to be simplified to the point that every phase is comprised of messages between *the same pairs of nodes*. This does not only make the correctness of the algorithm very intuitive, but also allows the transaction flow to be heavily *chained* (i.e., pipelined in popular parlance): PREPARE messages for block i are piggybacked on PRE-COMMIT messages for block $i-1$, which are piggybacked on COMMIT messages for block $i-2$, which themselves are piggybacked on DECIDE messages for block $i-3$.
  Chaining has a dramatic impact on throughput: chained HotStuff has a theoretical throughput of 4x that of non-chained HotStuff, achieving an amortized commit latency of 1 commit per message delay. The trade-off is that *actual* commit latency from the point the primary starts transmitting a PREPARE message is 4 (instead of PBFT's 3, or SBFT's 1).
- **View changes are part of the commit flow.** Unlike other PBFT-family protocols with complex view-change messages outside of the commit flow that are quadratic in the number of messages, HotStuff homogeneous phases allows it to deterministically switch to a new leader after every commit with no additional messages. PBFT-family algorithms that use the 'stable leader' paradigm are vulnerable to long-running transaction censorship attacks, because even though a malicious leader cannot by itself challenge the safety properties of the algorithm, they can refuse to send out PREPARE messages for selected transactions (e.g., a government preventing a political enemy from transferring their funds to collaborators). It is difficult for honest follow-

> ers to detect censorship and trigger a VIEW-CHANGE, since the faulty leader might otherwise be sending out PREPARE messages for other transactions. With HotStuff, faulty leaders can only censor transactions for a single block before they are replaced by another (hopefully honest) leader.

Several other consensus protocols were considered and rejected during the design of ParallelChain F. Zyzzyva [14] has high throughput and low latency but was recently found to be vulnerable to sophisticated attacks [15]. Tendermint [12] is battle-tested in long-running high-traffic networks (e.g., Cosmos), but is pareto dominated by chained HotStuff [13] on every metric but latency[5]. CFT algorithms (like Paxos [5], used in ParallelChain S) are not suitable for the Byzantine failure model.

Looking beyond HotStuff, another algorithm that we are looking to evaluate for integration into future releases of ParallelChain F is SBFT [16] (§6.1). We think a chained version of SBFT has the potential to achieve lower latency and higher throughput than chained HotStuff, at the cost of acceptable vulnerability to censorship attacks.

4 ParallelChain in operation

4.1 Transaction flow in the normal case

4.2 Network administration

TODO: Talk about voting.

TODO: We'll be able to fill this in once we have a comfortable understanding of the features of ParallelCore DB and libhotstuff.

5 Applications

5.1 XPLL

TODO: We'll be able to fill this in when we get a more precise idea about how XPLL is going to look like from Tunde.

6 Future work

[5] A near-term development goal of the Tendermint Core implementation is to design a *chained Tendermint*. We are looking forward to benchmarking it when it becomes available.

6.1 Chained SBFT

6.2 New key-value store

6.3 Consensus on hashes

Block proposals can grow very large, even for seemingly simple operations. Remember that a transaction proposals in a block proposal not only includes everything an engine needs to invoke a smart contract, but also the read and write sets generated by *actually invoking* said smart contract on the primary. To counter this, instead of passing around and gathering votes for full transaction proposals in PRE-PREPARE, PREPARE, and COMMIT, we have HotStuff gather votes for hashes of block proposals instead. 'Consensus on hashes' is an optimization strategy for SMR that has been known since at least Liskov and Castro's 1999 paper on PBFT.

To implement this scheme, PRE-PREPARE messages take the form: <PRE-PREPARE, v, n, D(bp)>_signed, where v is the current view number, n is the message's sequence number, D(bp) is the SHA256 of block proposal bp.

When backups receive a PRE-PREPARE message, it decides its validity by computing the SHA256 of the contents of its SEA, checking if this equals D(bp), and validating each of the transaction proposals in sequence; if it does, then it broadcasts a positive PREPARE message. Since the primary only sends a PRE-PREPARE message for a block proposal to a backup b if it knows that b has stashed in its SEA all of the transaction proposals in the block proposal, the only typical* scenario in which an honest backup would broadcast a negative PREPARE message is if some transaction proposal in the bp is not valid.

6.4 Checkpointing and pruning

TODO:

6.5 Sharding

TODO: talk about the sharding design of ParallelChain S.

TODO: talk about how sharding data availability means bad latency and throughput for cross-shard transactions (and how the Ethereum community's priorities right now are to reduce blockchain size, not speed up execution, and how we partially solve the former using checkpointing and pruning).

TODO: I have an intriguing idea for execution sharding that seems correct.

7 Conclusions

The 'S' in 'ParallelChain S' stands for **s**harded and **s**calable.

The 'F' in 'ParallelChain F' stands for **f**ully replicated and Byzantine **f**ault tolerant. It also stands for **f**uture.

References

[1] Nakamoto, S. (2009). Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`

[2] Marc, A. (2014, Jan 21). *Why Bitcoin Matters.* Dealbook - The New York Times. `https://dealbook.nytimes.com/2014/01/21/why-bitcoin-matters/`

[3] Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7), July 1978. `https://dl.acm.org/doi/10.1145/359545.359563`

[4] Lamport, L., Shostak, R., Pease, M. (1982). The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982, p. 382-401. `https://dl.acm.org/doi/10.1145/357172.357176`

[5] Lamport, L. (1998). The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998, p. 133-169. `https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf`

[6] Castro, M., Liskov, B. (1999). Practical Byzantine Fault Tolerance. *Usenix Symposium on Operating Systems Design and Implementation*, 1999. `https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance`

[7] Buterin, V. (2013). Ethereum Whitepaper. `https://ethereum.org/en/whitepaper/`

[8] Copeland, J. (2017). The Church-Turing Thesis. *Stanford Encyclopedia of Philosophy.* `https://plato.stanford.edu/entries/church-turing/`

[9] Buterin, V. (2021). Why sharding is great: demystifying the technical properties. *Vitalik Buterin's Website.* `https://vitalik.ca/general/2021/04/07/sharding.html`

[10] Zhou, J., Miller, A., Sears, R., et al. (2021). FoundationDB: A Distributed Unbundled Transactional Key Value Store. *ACM Special Interest group on Management of Data (SIGMOD)*, 2021. `https://www.foundationdb.org/files/fdb-paper.pdf`

[11] Yin, M., Malkhi, D., Reiter, M., et al. (2019). HotStuff: BFT Consensus with Linearity and Responsiveness. *ACM Symposium on Principles of Distributed Computing (POCD)*, 2019. `https://dl.acm.org/doi/10.1145/3293611.3331591`

[12] Buchman, E., Kwon, J., Milosevic, Z. (2018). The latest gossip on BFT consensus. `https://arxiv.org/abs/1807.04938`

[13] Alqahtani, S., Demirbas, M. (2021). Bottlenecks in Blockchain Consensus Protocols. https://arxiv.org/abs/2103.04234

[14] Kotla, R., Lorenzo, A., Dahlin, et al. (2007). Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Symposium on Operating Systems Principles (SOSP)*, 2007. http://www.cs.cornell.edu/lorenzo/papers/kotla07Zyzzyva.pdf

[15] Abraham, I., Gueta, G., Makhli, D., et al. (2017). Revisiting Fast Practical Byzantine Fault Tolerance. https://arxiv.org/pdf/1712.01367.pdf

[16] Gueta, G., Abraham, I., Grossman, S., et al. (2018). SBFT: A Scalable and Decentralized Trust Infrastructure. https://arxiv.org/abs/1804.01626