ParallelChain F
Draft 2 (WIP)

Digital Transaction Limited (DTL)

June 21, 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. For DTL, ParallelChain F comes after ParallelChain S, a permissioned blockchain with which it shares execution and data layers (§3.2). The two systems differs in two key regards: 1. In ParallelChain F, each node's state cache (§3.4) maintains a full picture of the world state, whilst in ParallelChain S, the state cache is sharded.

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); the reader is strongly encouraged to read the papers referenced in this section. We then describe the system design (§3), walking through the lifecycle of a transaction from proposal to commit for clarity (§4.1). DTL's XPLL network, our flagship ParallelChain F deployment, 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 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-

[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.

[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.

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 [7]. 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 simply 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 long or 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 [8]. 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 Pokémon 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.
- **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.

[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.

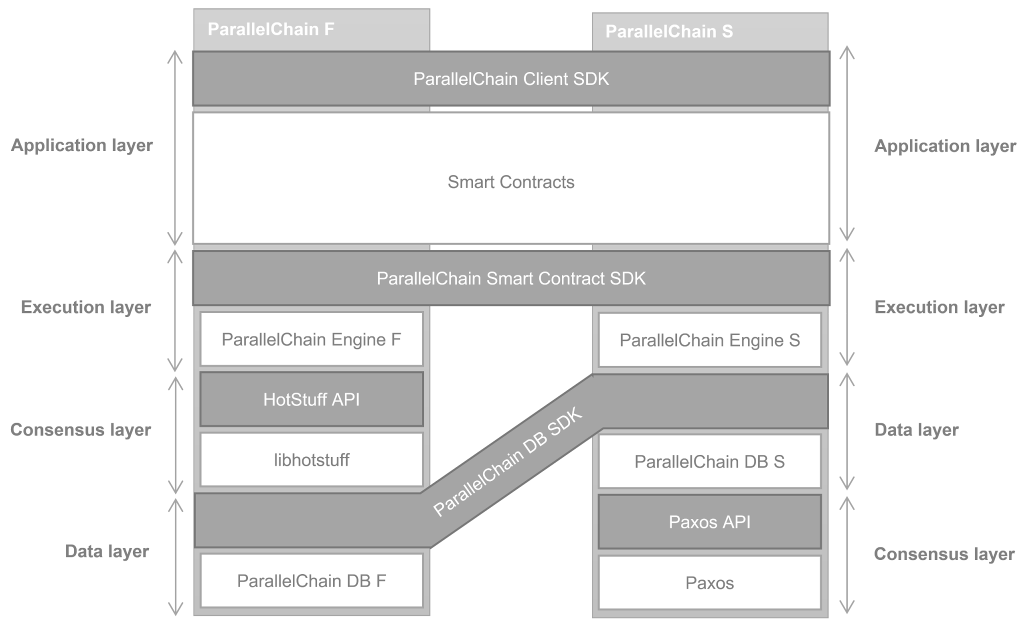


Figure 1: The Layered Architecture of ParallelChain

- 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 languages compilable to EVM bytecode like Solidity (to enable 'native' invocation fees). 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 world state (§3.4) 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 `ProposerAddr()` 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 limitation that remains in this initial release of ParallelChain F is the lack of 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 enterprise consortium deployments of the system, where permission to deploy smart contracts can be restricted, or the XPLL mainnet, since the XPLL smart contract implements invocation fees at the application layer. Adding Solidity (or other gas-determinable language) support to ParallelChain F is an important long term goal for the DTL team; check out fabric-chaincode-evm to see how the Hyperledger team went about adding optional Solidity support to their Hyperledger Fabric permissioned blockchain.

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 world 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 just an optimization to speed up reads.

In the interest of clarity, we make a distinction between two primitive concepts before specifying the Block data format:

- **Invocation**: An execution layer concept, i.e., *invoking* a smart contract. Generally speaking, smart contracts can be categorized into: 1. *Stateless* smart contracts, whose runtime behavior is

Field name	Data type	Remarks
`scName`	`string`	
`scVer`	`string`	
`input`	`string`	Passed as a parameter to the invoked smart contract's Handle function
`invokerAddr`	`[64]byte`	
`nonce`	`uint64`	The number of invocations the invoker has added to the blockchain; prevents invocation replay attacks

Table 1: Invocation data format

totally determined by the invoker's identity (most obviously their public address) and the `input` argument passed in with the call, and 2. *Stateful* smart contracts, whose runtime behavior is determined by the world state at a particular block height and invocation index in addition to the aforementioned factors.

- **Transaction**: A data layer concept. Stateful smart contracts interact with world state using the `Get()` and `Set()` methods exposed by the Smart Contract SDK. A transaction is a 2-tuple (readSet, writeSet): readSet an unordered set of of `Get`s, and writeSet an unordered set of `Set`s. A smart contract invocation produces exactly one transaction.

The blockchain that a ParallelChain F network collectively maintains is an ordered list of *invocations*, not *transactions*. The world state *at* a particular block height h and invocation index i is defined as the world state constructed by playing (i.e., executing) smart contract invocations one-by-one and in order from the genesis block until and inclusive of the i-th invocation in block h. Table 1 specifies the Invocation data format.

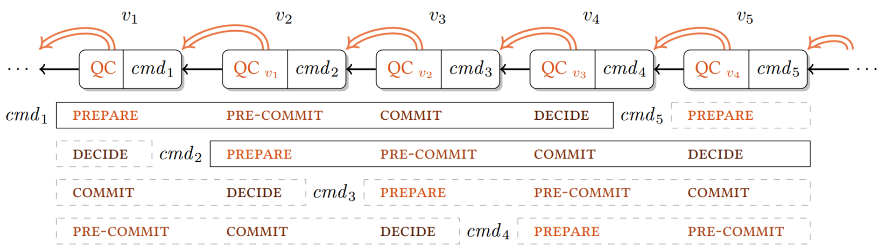
Readers aware of the ParallelChain S block structure might find this surprising, since the ParallelChain S blockchain is exactly an ordered list of transactions, without any information about the smart contract invocation that produced it. This deviation from the ParallelChain S design is made necessary by the need for ParallelChain F to provide users with validity (§3.1) guarantees in a network membership model much more challenging than that targeted by ParallelChain S. Barring emerging cryptographic techniques such as ZK-Proofs, the only way a node operator can be certain that a transaction is valid is to play for themselves the smart contract invocation that the network claims had produced the transaction.

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 prevent state cache divergence. This is unlike ParallelChain DB S, which can safely commit transactions in parallel since all full nodes in a ParallelChain S cluster are part of a single physical FoundationDB cluster.

The Block data format is mostly mundane, consisting mostly of an ordered list of transactions and the metadata fields typically associated with blockchains. Perhaps the most significant deviation from the norm is the inclusion of a `proposerAddr` field. We initially added for the purpose of implementing the `ProposerAddr()` function in the Smart Contract SDK, but this field will come in handy when we replace HotStuff with chained SBFT (§6.1) and implement sharding (§6.2), since gives networks a tool to identify lagging leaders and troubleshoot them to improve network performance.

Field name	Data type	Remarks
`blockFormatVer`	`uint16`	
`proposerAddr`	`string`	Public address of the consensus leader (§3.5) that broadcasted the PREPARE message containing this block
`prevHash`	`[32]byte`	KangarooTwelve blockHash of the previous block
`invocationsHash`	`[32]byte`	KangarooTwelve root hash of the Merkle tree convering `invocations`
`blockHash`	`[32]byte`	KangarooTwelve hash over the fields above
`invocations`	`[]Invocation`	Equivalently, `[]String`
`invokerSig`	`[64]byte`	ed25519 signature over above fields generated using the invoker's private key

Table 2: Block data format



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 [11]

A major limitation for ParallelCore DB currently is FoundationDB's 5-second transaction rule. Again, we do 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 a different key-value store.

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 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 three 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). HotStuff's 4-phase commit flow is illustrated in Figure 2.
- **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's 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 followers 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. Quickly rotating leaders also allows us to quickly and easily detect if a validator node has gone offline, which could be made a slashable offence to create an incentive structure that promotes higher network uptime.

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 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. Figures 3 and 4 compare the performance and scalability characteristics of HotStuff compared to some of the mentioned algorithms.

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.

[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.

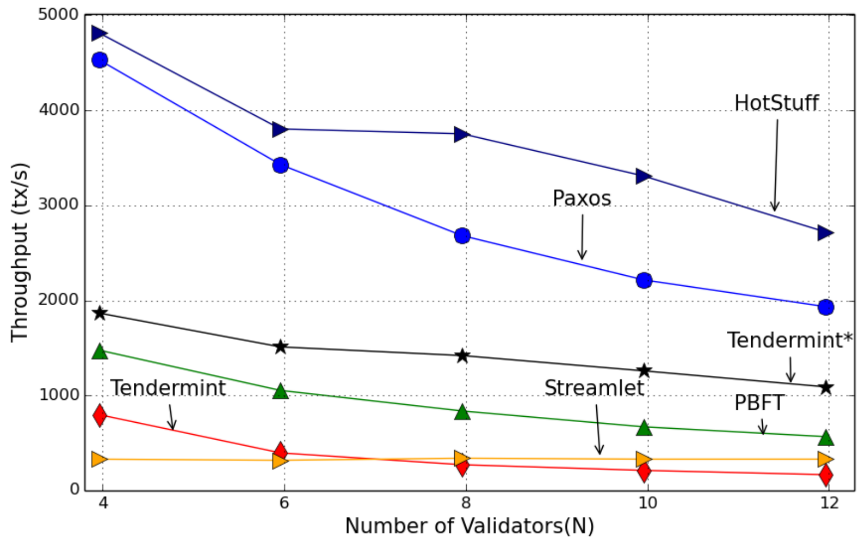


Figure 3: Consensus throughput comparison [13]

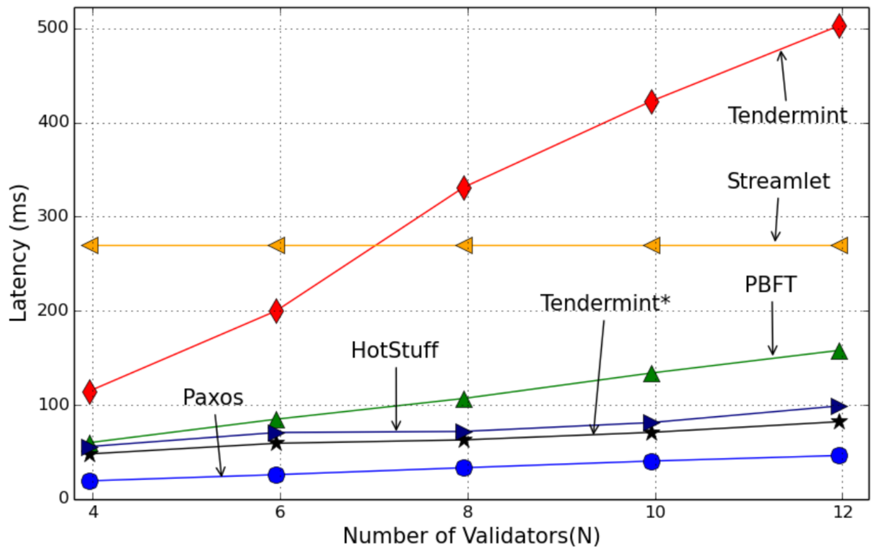


Figure 4: Consensus latency comparison [13]

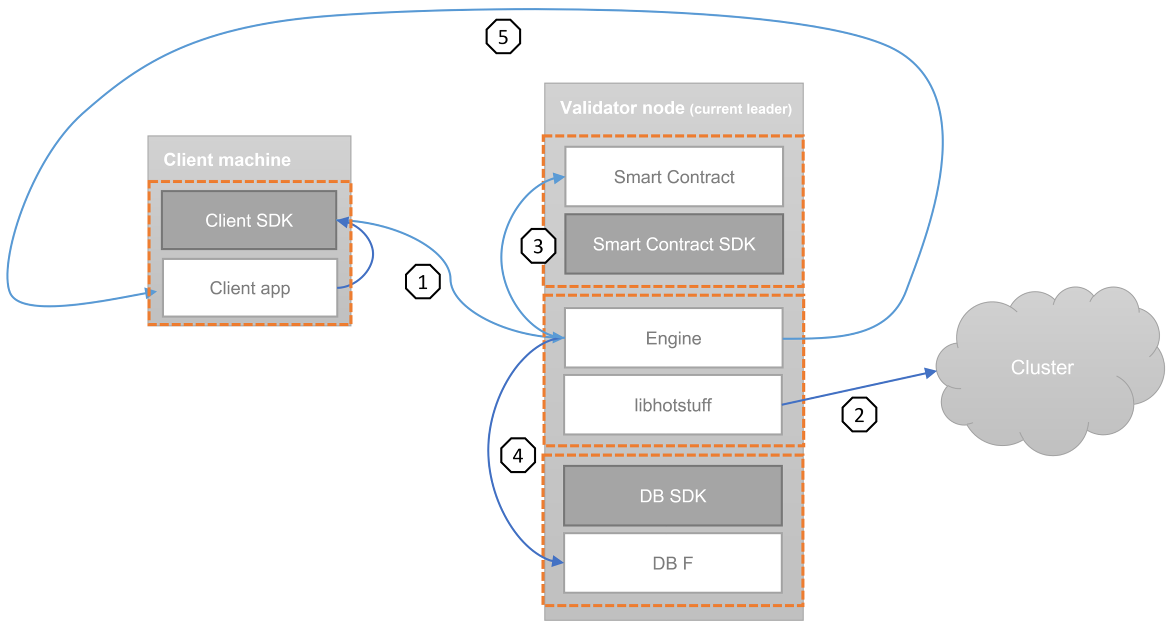


Figure 5: Invocation flow. Dashed orange boxes represent process boundaries.

4 ParallelChain in operation

4.1 Invocation flow in the normal case

Figure 5 describes at a high level how code execution jumps between machines and processes in the handling of a single smart contract invocation. A brief rundown of the stages:

1. Client app uses the Client SDK to make a gRPC call to Engine.
2. Engine passes invocations to libhotstuff, which decides when to pack a block and trigger a consensus round.
3. Once a block is committed, Engine executes the invocations in the block one-by-one by loading the smart contract binary into the *execution container* process and calling its `main()` function.
4. Engine passes the `Transaction` struct generated by the invocation execution into DB F to be committed into the state cache.
5. Engine responds to the Client SDK gRPC call.

The Client SDK call for invoking a smart contract accepts a `blocking` parameter. If `true` is passed, step number 5 becomes step number 2, and the client has to poll the network regularly to check if the invocation was committed into the blockchain. Moderately-sized ParallelChain F networks should be able to achieve small enough invocation latencies that we expect most applications to pass in `false`.

5 Applications

5.1 XPLL

TBD pending Tunde's XPLL smart contract design specification.

6 Future work

6.1 Chained SBFT

Ittai Abraham, a co-author of both HotStuff and SBFT has claimed that a 'chained SBFT' algorithm is possible [20]. We are inclined to believe him, but without more rigorous due dilligence, we cannot project with certainty that chained SBFT will have better performance than HotStuff. What we know for certain however is that SBFT as it exists today has the optimal fast path commit latency (1 message delay vs 4 in the case of HotStuff), and beats out every other non-chained BFT SMR algorithm we are aware of with increasingly large margins in larger cluster sizes. So far, nobody seems to have published a direct comparison between SBFT and HotStuff; we will update this paper if one becomes available or when we complete our own evaluations.

6.2 Hierarchical Deterministic Public Addresses

Currently, users have to include the singular public key associated with their private key in smart contract invocations. Adding HD Wallet support (BIP32) to ParallelChain F would will enhance privacy and anonymity by allowing users to use a different, deterministically generated public address to make smart contract invocations.

6.3 Consensus on hashes

Instead of passing around a complete block in all four stages of consensus (in HotStuff), we could instead pass around a hash of the block for in the latter 3 stages. Consensus on hashes is a common optimization for BFT SMR algorithms that has been known since at least [6].

6.4 Sharding

ParallelChain F as it has been described so far achieves good throughput and latency in clusters with dozens of nodes, and acceptable throughput and latency in clusters with hundreds of nodes. Like any distributed system tied together using classical consensus protocols, however, ParallelChain F exhibits 'reverse scalability', i.e., it *slows down* the more nodes are added into the cluster. Though we believe that the performance characteristics achieved by reasonably-sized ParallelChain F clusters (up to 40 nodes) running commodity server hardware should be enough to support the vast majority of real-world applications (case in point, the Visa network at peak loads processes *only* 4000 transactions per second), we agree with the blockchain community that there is obvious value in blockchain systems that perform better the more nodes are added into the network, instead of slowing down. A ParallelChain F network with true scalability would not have to pick between decentralization and throughput: the former would lead to the latter.

The blockchain research community seems to have coalesced around 'sharding' as the holy grail blockchain scalability solution, with the upcoming Ethereum 2.0 being the most prominent 'sharded blockchain' [17]. The sharding Ethereum 2.0 and other blockchains like NEAR are leveraging is qualitatively different from the conventional idea of sharding in Database engineering (which ParallelChain S uses), and even farther from the kind of sharding we envision ParallelChain F to move towards. Blockchain sharding designs generally go in one of two directions:

- **Shard data availability**: in this design, nodes assigned to a shard provide fault-tolerant data persistence for 'off-chain' transactions that are eventually included in the blockchain using a smart contract (hence, the term 'L2 scaling') that verifies a Zero Knowledge Proof (ZKP) that the transactions were exeucted correctly in a process termed ZK Rollup. The technical details of what Zero Knowledge Rollups are are beyond the scope of this document[6], but the tl;dr is that shard chains reduce the *storage* burden on nodes, potentially allowing for abundant consumer devices like laptops to play an active role in keeping the blockchain immutable and correct.
- **Shard execution**: this is the design we are considering for ParallelChain F. In this design, nodes assigned to a shard (Validator Group [VG] in our terminology) are sent blocks of *invocations* pre-ordered by an Ordering Group (OG) and are tasked with executing the invocations to produce blocks of *transactions*. These Transaction Blocks are then sent back to the OG for inclusion into the blockchain. Each VG and *the* OG form disjoint HotStuff/SBFT validator sets.

Execution shards improve the scalability of ParallelChain F in two ways:

- They reduce the size of consensus validator sets with an acceptable sacrifice in fault tolerance. PBFT-family BFT SMR algorithms generally tolerate up to 1/3 of the network suffering Byzantine faults. By frequently (on the order of a shuffling every 12 hours to few days) and randomly (e.g., using special nodes running VDFs like proposed for Ethereum 2.0 sharding), we can make it such that a malicious adversary needs to control a large share of all nodes to have a good chance of controlling a quorum of voting power in the OG or one of the VGs.
- They allow the burden of execution to be parallelized between many shards. The potential benefit from this is massive in networks with complex, long-running smart contracts.

Our disagreement with NEAR and Ethereum with regards to sharding design comes down to a matter of differing priorities. The former networks aim for a kind and degree of decentralization that we have already argued is unnecessary §3.3. In order to achieve this aim, they need ensure that members of their respective communities should be able to run validator clients in the background on their laptops if they so choose. What is making this infeasible today is the growth of the blockchain: high-end consumer laptops today ship with 512GB SSDs; the Ethereum blockchain is currently over 800GBs long, and even with pruning, node operators need to fully download the blockchain first to validate it. ParallelChain F is intended to be deployed on server hardware, and in servers, storage is abundant and dirt-cheap. The bottleneck with our current design is network latency and bandwidth, and execution time, which are the two problems mitigated by execution sharding.

Figures 6 and 7 introduces our proposed sharding architecture. The example network consists of two Validator Groups in addition to the Ordering Group. To explain our design, we have to introduce

[6]Readers interested in rollups and other L2 scaling solutions are nudged to read [18] and [19]

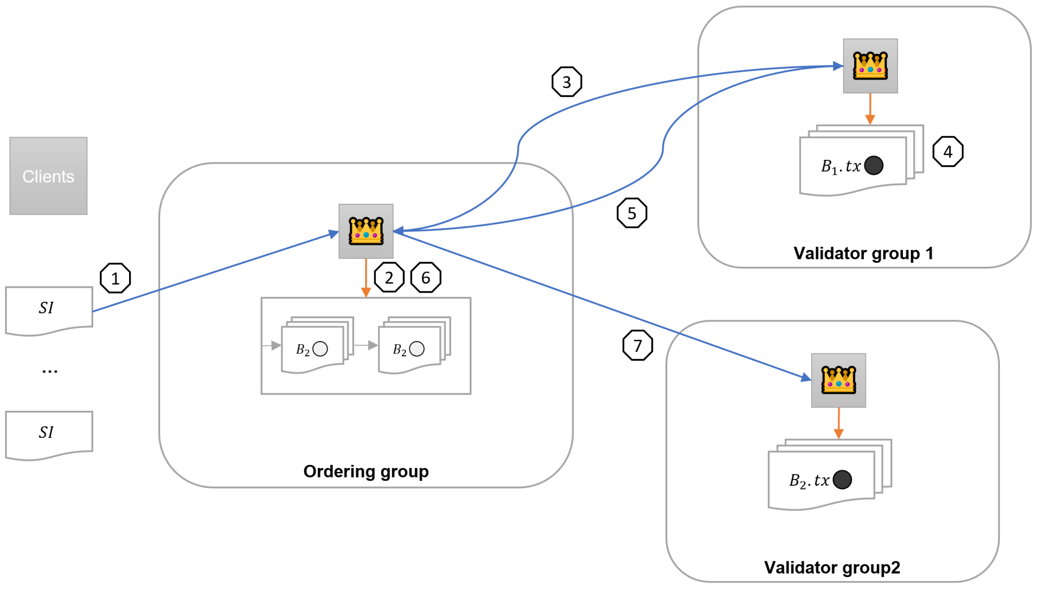


Figure 6: Sharding network architecture. The numbered events correspond to those in Figure 7

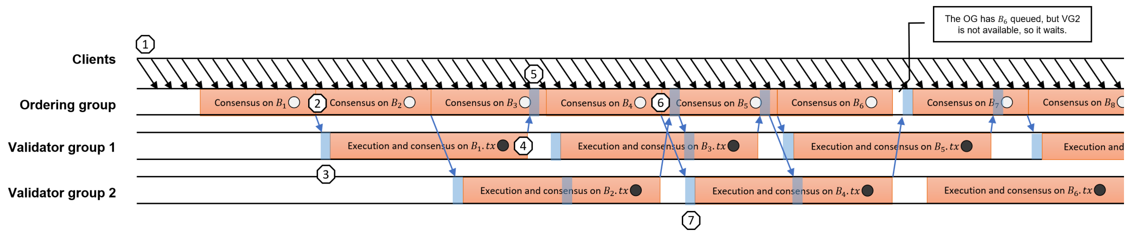

The OG has B_6 queued, but VG2 is not available, so it waits.
Clients
1
Ordering group
Consensus on B_1
2
Consensus on B_2
Consensus on B_3
5
Consensus on B_4
6
Consensus on B_5
Consensus on B_6
Consensus on B_7
Consensus on B_8
Validator group 1
Execution and consensus on $B_1.tx$
4
Execution and consensus on $B_3.tx$
Execution and consensus on $B_5.tx$
Execution and
3
Validator group 2
Execution and consensus on $B_2.tx$
Execution and consensus on $B_4.tx$
Execution and consensus on $B_6.tx$
7

Figure 7: The numbered events demonstrates how code execution jumps between ordering and validation groups in the handling of a single block.

Field name	Data type	Remarks
`didTerminate`	`bool`	`true` if the smart contract invocation that produced the transaction did not terminate gratefully.
`readSet`	`string`	e.g., "John.balance:2" where key:versionNum
`writeSet`	`string`	e.g., "Jane.balance:1"

Table 3: Transaction data format for a hypothetical sharded ParallelChain F

Field name	Data type	Remarks
`playInt`	`uint64`	

Table 4: playIndex, an additional field in the block data format for sharded ParallelChain

Field name	Data type	Remarks
`transactionsHash`	`uint32`	KangarooTwelve root hash of the Merkle tree convering `transactions`
`transactions`	`[]Transaction`	If the blockchain at height h contains transaction segments up to height $h-1$, the block at height $h+1$ either contains the transaction segment at height h, or nil. The latter case is true if the validator group tasked with generating transaction segment h is 'late'.

Table 5: Transaction segment, two additional fields in the block data format

a transaction data format (Table 3) which is included in the blockchain—recall that the current ParallelChain F blockchain only consists of invocations—and a extension to the block data format we term the 'transaction segment' (Table 5) to accomodate this inclusion. We first discuss the transaction segment data format, and then discuss how the version information stored in the transaction data format enable optimistic parallel execution of smart contracts.

Transactions are generated by VGs by executing smart contract invocations, and are included into blocks using the `transactions` and `transactionsHash` fields. What transactions are included in a block's Transaction segment is timing dependent: the OG cannot include transactions corresponding to the invocations in B_i in B_i itself, since the invocations will only have been run after it is transmitted to the VG; *earliest* these transactions can be included into the blockchain is in the transaction segment in B_{i+1}. The OG includes transactions into the blockchain in the order of their smart contract invocation, so even $B_{i+1}.txs$ arrives at the OG before $B_i.txs$, the OG will not include the former in a block until it has received the latter. This might cause liveness problems if validator groups work at different rates, or if computational complexity varies greatly between blocks, but this can be mitigated using additional mechanisms (that are currently floating around in the author's head, but will not be discussed in this paper for brevity). A single transaction segment might contain transactions corresponding to different blocks; computing which blocks exactly is simple: if $B_i.playIndex$ (Table 5) is $i-5$, and $B_{i+1}.playIndex$ is $i-3$, then the transaction segment in B_{i+1} contains transactions corresponding to *all* invocations in B_{i-4} and B_{i-3}. Validator groups play the transactions included in a block before executing invocations.

Consider the case of two blocks B_i and B_{i+1} where `playIndex` $= c$ in both blocks. The world state at `playIndex` $= c$ contains account balance mappings for a cryptocurrency application: $John.balance \leftarrow 100$, $Jane.balance \leftarrow 0$, and $Jack.balance \leftarrow 0$. Suppose now that John makes two smart contract invocations, one to transfer 100 tokens to Jane, and another to transfer 100 tokens to Jack. The first invocation is included in B_i, the other in B_{i+1}. Assuming that the smart contract John uses to make the transfers are properly programmed, the transfer to Jane should succeed, and the one to Jack should fail, but because both invocations are played on the world state at `playIndex` $= c$, both would succeed in generating transactions that will eventually be included in the blockchain. In a naive implementation, this would lead to a violation of ACID Isolation. Our sharding design protects Isolation by including version numbers in the `readSet` and `writeSet` fields of the transaction data format. When nodes play transactions on their local copy of the world state, they use these to detect Read-Write (RW) and Write-Write (WW) conflicts. If one is detected, the transaction is aborted and rolled back.

A rundown through the numbered events:

1. Client software sends invocations to the OG. If an invocation ends up reaching a node in the validator group (e.g., because a client's routing table is stale) that invocation is redirected to the OG.
2. The OG leader collects invocations and packs them into a block B_1. The block is put through a consensus round in OG for ordering and inclusion into the blockchain.
3. After the block is committed by consensus in the OG, the leader sends B_1 to the leader of Validator Group 1 (VG1). The VG1 leader executes the transactions in B_1 locally to compute $B_1.tx$.
4. B_1 and $B_1.tx$ are put through a consensus round to collect signatures; each node in VG1 executing the transactions itself to validate.
5. Once a quorum of signatures is collected from VG1, $B_1.tx$ is sent back to the OG leader (including the signatures).
6. The OG leader verifies the included signatures. Everything checks out, so it includes $B_1.tx$ in the transaction segment of B_4.
7. The VG2 leader receives $B_1.tx$, along with B_4 *and any other block it has missed* from the OG.

This sharding design has not been formally specified, verified, or proven. At this point, we are quite confident that the algorithm guarantees safety, and believe that incentive structures (e.g. staking) can be set up to incentivize high node uptime and therefore good liveness. We are actively soliciting comments, inquiries, and critique for our sharding design from the distributed systems community.

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

[17] Ethereum Wiki. (2021). On sharding blockchains FAQs https://eth.wiki/sharding/Sharding-FAQs

[18] Skidanov, A. (2019). Overview of Layer 2 approaches: Plasma, State Channels, Side Chains, Roll Ups. https://near.org/blog/layer-2/

[19] Interdax. (2020). Scaling Ethereum on L2: Optimistic Rollups and ZK-Rollups. https://medium.com/interdax/ethereum-l2-optimistic-and-zk-rollups-dffa58870c93

[20] Abraham, Ittai. (2019). What is the difference between PBFT, Tendermint, SBFT and HotStuff? https://decentralizedthoughts.github.io/2019-06-23-what-is-the-difference-between/