Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol

Sean Braithwaite
Informal Systems, Lausanne, Switzerland

Ethan Buchman
Informal Systems, Toronto, Canada

Igor Konnov
Informal Systems, Wien, Austria

Zarko Milosevic
Informal Systems, Lausanne, Switzerland

Ilina Stoilkovska
Informal Systems, Wien, Austria

Josef Widder
Informal Systems, Wien, Austria

Anca Zamfir
Informal Systems, Lausanne, Switzerland

Abstract

Blockchain synchronization is one of the core protocols of Tendermint blockchains. In this short paper, we discuss our recent efforts in formal specification of the protocol and its implementation, as well as some initial model checking results. We demonstrate that the protocol quality and understanding can be improved by writing specifications and model checking them.

2012 ACM Subject Classification Software and its engineering → Software verification; Software and its engineering → Software fault tolerance; Theory of computation → Logic and verification

Keywords and phrases Blockchain, Fault Tolerance, Byzantine Faults, Model Checking

Digital Object Identifier 10.4230/OASIcs.FMBC.2020.10

Category Short Paper

Supplementary Material `https://github.com/tendermint/spec/tree/master/rust-spec/fastsync`

Funding Supported by Interchain Foundation (Switzerland).

1 Introduction

Tendermint is a state-of-the art Byzantine-fault-tolerant state machine replication (BFT SMR) engine equipped with a flexible interface supporting arbitrary state machines written in any programming language [2]. Tendermint is particularly popular for proof-of-stake blockchains, and constitutes a core component of the Cosmos Project [3]. At the heart of the Cosmos Project is the InterBlockchain Communication (IBC) protocol for reliable communication between independent BFT SMs; what TCP is for computers, IBC aims to be for blockchains.

Multiple Tendermint-based blockchains currently run in production on the public Internet and have for over a year, with new ones launching regularly, carrying billions of dollars of cumulative value in the market capitalizations of their respective cryptocurrencies. One of the primary deployments, the so-called Cosmos Hub blockchain, is operated by a diverse set of 125 consensus forming nodes connected to one another over an open-membership gossip network consisting of hundreds of other nodes.

Tendermint was the first to apply traditional BFT consensus protocols to blockchain systems [9]. The core Tendermint BFT consensus protocol constitutes a modern implementation of the consensus algorithm for Byzantine faults with Authentication from [6] built on top of an efficient gossiping layer. The latest description of the consensus protocol can be found in the technical report [4]. Tendermint consensus has been a source of inspiration for a wide variety of blockchain systems that have followed [15, 5], though few, if any, have achieved its level of maturity in production.

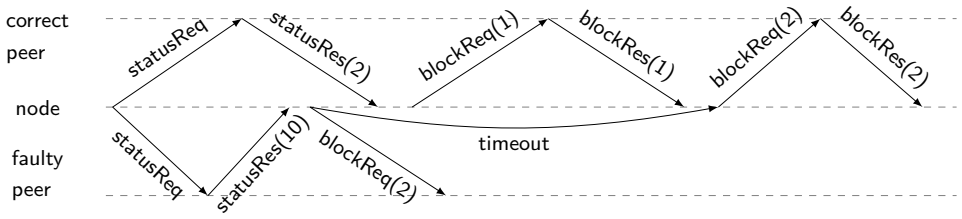

Figure 1 A Fastsync execution for a fully unsynchronized node of height 1.

The reference implementation of the Tendermint software is written in Go [14]. Under the hood, it consists of several fault-tolerant distributed protocols that interact to ensure efficient operation:

Consensus. Core BFT consensus protocol including the gossiping of proposals, blocks, and votes.

Evidence. To incentivize consensus participants to follow the consensus protocol (and not behave faulty), in the proof-of-stake systems, misbehavior is punished by destroying stake. This protocol gossips evidence of malicious behavior in the form of conflicting signatures.

Mempool. A protocol to gossip transactions, ensuring transactions eventually end up in a block are distributed to all participants.

Peer Exchange. Gossiping is based on communication only with a subset of the peers. Managing a list of available peers and selecting the peers based on performance metrics is done by this protocol.

Blockchain synchronization (Fastsync). If a peer gets disconnected by the network for some time, it might miss the most recent blocks in the blockchain. A node that recovers from such a disconnection uses the blockchain synchronization protocol called Fastsync to learn blocks without going through consensus.

We are conducting a project to formally specify and model check these protocols. The first protocol we considered was the blockchain synchronization protocol called *Fastsync*. Specifications can be found in English [10] and TLA$^+$ [11].

Fastsync. A full node that connects to a Tendermint blockchain needs to synchronize its state to the latest global state of the network. One way to achieve this is to update its local copy of the blockchain and replay all transactions, using Fastsync: initially, the node has a local copy of a blockchain prefix and the corresponding application state that may be out of date. The node queries its peers for the blocks that were decided on by the Tendermint blockchain since the time the full node was disconnected from the system. After receiving these blocks, the protocol executes the transactions that are stored in the blocks, in order to synchronize to the current height of the blockchain and the corresponding application state.

Figure 1 shows a typical execution of the Blockchain Synchronization protocol. In this execution, a new node connects to two full nodes: a correct peer and a faulty peer. The node requests the blockchain heights of the peers by issuing `statusReq`. Once a peer replies with its height, e.g., with `statusRes(10)`, the node can request for a block i by sending the message `blockReq(i)`. In our example, the correct peer receives the request `blockReq(1)` for block 1 and replies with the message `blockRes(1)` that contains the block. In a Tendermint blockchain, the commit for block (signed votes messages) `h` is contained in block `h+1`, and thus a node performing Fastsync must receive two sequential blocks before it can verify fully

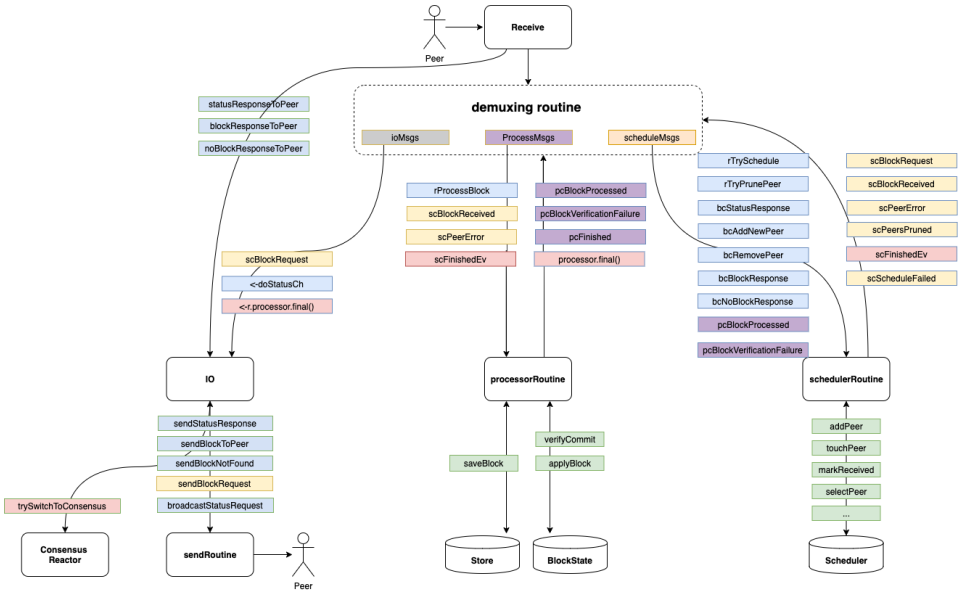

Figure 2 Concurrent threads of execution in the Fastsync implementation [13].

the first one. If verification succeeds, the first block is accepted; if it fails, both blocks are rejected, since it is not known which block was faulty. When the node rejects a block, it suspects the sending peer of being faulty and evicts this peer from the set of peers. The same happens when a peer does not reply within a predefined time interval. In our example, the faulty peer is evicted, and the node finishes synchronization with the correct peer.

The above example may produce an impression that it is easy to specify and verify correctness of Fastsync. (The authors of this paper thought so.) By writing several protocol specifications in English and TLA$^+$ and by running model checkers, we have found that the specifications, in particular in the presence of faulty peers, are intricate.

2 Architecture

The most recent implementation of the Fastsync protocol, called V2, is the result of significant refactoring to improve testability and determinism, as described in the Architectural Decision Record [13]. In the original design, a go-routine (thread of execution) was spawned for each block requested, and was responsible for both protocol logic and network IO. In the V2 design, protocol logic is decoupled from IO by using three concurrent threads of execution: a scheduler, a processor, and a demuxer, as per Figure 2. Rounded-corner rectangles represent concurrent threads that exchange the events that are depicted by rectangles on the edges between threads.

The demuxer acts as an internal bridge: it is responsible for translating between internal events and network IO messages, and for routing events between components. Both the scheduler and processor are structured as finite state machines with input and output events. Input events are received on an unbounded priority queue, with higher priority for error events. Output events are emitted on a blocking, bounded channel. Network IO is handled by the Tendermint p2p subsystem, where messages are sent in a non-blocking manner.

The IO component is responsible for exchanging (sending and receiving) Fastsync protocol messages with peers. There is one send and one receive routine per peer (denoted *Receive* and *sendRoutine* on Figure 2, respectively).

The scheduler contains the business logic for tracking peers and determining which block to request from whom. The scheduler receives relevant protocol messages from peers (for example *bcBlockResponse* and *bcStatusResponse*), but also internal events that are the result of the block processing in the processor (the events carry the information of whether a block was successfully processed or there was an error). The scheduler schedules block requests by emitting internal events (*scBlockRequest*) and also informs the processor about internal processing, for example, when block response is received (*scBlockReceived*) or if there is an error in peer behaviour (*scPeerError*).

The processor handles the computationally expensive block processing, including verification of consensus signatures and execution of all transactions. It manages the block store (denoted *Store* on Figure 2) and interacts with the Tendermint block execution component (denoted *BlockState*). Furthermore, it informs the scheduler whether a block processing was successful (*pcBlockProcessed*) or it has led to an error (*pcBlockVerificationFailure*).

Once the Fastsync protocol terminates, this is signaled to the Tendermint consensus component (denoted *ConsensusReactor*) with a *trySwitchToConsensus* event.

3 Specifications in English and TLA$^+$

Structured Specification in English. We start our formalization by a structured English specification [10], that consists of four parts:

1. *Blockchain.* Formalization of relevant properties of the blockchain and its blocks.
2. *Sequential problem statement.* Parts of the sequential safety specification read as follows:

   > "Let *bh* be the height of the blockchain at the time Fastsync starts. When the protocol terminates, it outputs a list of all blocks from its initial height to some height *terminationHeight* $\geq bh - 1$". (Fastsync cannot synchronize to the maximum height *bh* as in Tendermint, verification of block at height h requires the commit from the block at height $h + 1$.)

   This specification is sequential, as it ignores that the blockchain is implemented in a distributed system, in which validators may be faulty. Even if they are correct, they locally have prefixes of different lengths, so that *bh* is not clearly defined a priori.
3. *Distributed aspects.* Here we introduce the computational model and the refinement of the problem statement. For instance, the above translates to:

   > "Let *maxh* be the maximum height of a correct peer to which the node is connected at the time Fastsync starts. If the protocol terminates successfully, it is at some height *terminationHeight* $\geq$ *maxh* $- 1$."

4. *Distributed protocol.* Specification of the protocol, where we describe inputs, outputs, variables, and functions used by the protocol. We specify functions mainly by preconditions, postconditions, and error conditions. Further, we provide invariants over the protocol variables. These inform both the implementation and the verification effort.

Specifications in TLA$^+$. The structure of the English specification already highlighted interesting properties of the protocols and pointed to some issues. As it is written in natural language, the English specification is ambiguous. To eliminate the ambiguities, we have written three TLA$^+$ specifications, which focus on different aspects of the protocol and its architecture:

- *High-level specification (HLS).* This specification contains the minimal set of interactions in the synchronization protocol. Its primary purpose is to highlight safety and termination. HLS was mainly written by the researchers.
- *Low-level specification (LLS).* While HLS captures the distributed protocol, there was a significant gap between HLS and the implementation. For instance, the implementation uses additional messages and contains detailed error codes, which are missing in HLS. The low-level specification is much closer to the implementation, and it was mainly written by distributed system engineers.
- *Concurrency specification (CRS).* As discussed above, the V2 implementation utilizes several threads that communicate via queues. To formally capture this structure, we wrote a specification that models threads and message queues.

We discuss the modeling assumptions of HLS [11], which builds the basis for our model checking work discussed in Section 4. (1) The node starts with a finite set of peers, which can shrink, when the node suspects peers of being faulty. This set is partitioned in two subsets: correct and faulty. (2) The blockchain can grow up to a fixed height. By fixing the parameters of (1) and (2), we run finite-state model checking with TLC [8] and APALACHE [7, 1]. We model the distributed system as two components: the node and its peers. These two components communicate via two variables: `outMsg` that keeps an output message from the node to a peer, and `inMsg` that keeps an input message from a peer to the node; both variables may be set to `None`, indicating that there is no message. The components alternate their steps: The odd turns belong to the node, whereas the even turns belong to the peers.

This approach is simple but powerful. On one hand, it dramatically decreases the state space, as there are no queues, and alternation produces fewer states than disjunction. On the other hand, it does not decrease precision, as the peers consume and produce message at a high degree of non-determinism. Moreover, this approach allows us to easily formulate fairness in the system as weak fairness over the variable `turn`, which encodes the scheduled component.

Finally, V2 relies on several timeouts to guarantee termination. In TLA$^+$ specifications, we simply model timeouts with non-determinism and weak fairness.

4 Model Checking with TLC and Apalache

While developing TLA$^+$ specifications, we were using TLA$^+$ Toolbox and the TLC model checker [8]. We also checked the safety properties with the new symbolic model checker APALACHE [7, 1]. So far, we have checked the specifications for tiny parameters such as 1–3 peers and small Blockchain heights[1]. Table 1 summarizes the results and running times of TLC and APALACHE. A central property is the protocol's termination:

$$\text{WF}_{turn}(FlipTurn) \Rightarrow \Diamond(state = \text{“}finished\text{”}) \qquad \text{(Termination)}$$

[1] The original protocol specification in TLA$^+$ is available in the main branch: `https://github.com/informalsystems/tendermint-rs/tree/master/docs/spec/fastsync`. The refined protocol specification was located in a pull request at the moment of writing (July 23, 2020): `https://github.com/informalsystems/tendermint-rs/pull/466`. After peer-review of the updates, the refined specification will be merged into the main branch.

In the early experiments, we did not find violations of this property, as TLC did not finish its exhaustive search. However, later, after refining the specification, TLC reported a counterexample at depth 7, which is indicated with $[\text{✗}]_{=7}$ in Table 1. The refined termination property looks as follows:

$$\Big(\mathrm{WF}_{turn}(FlipTurn)$$
$$\wedge \diamond(inMsg.type = \text{“}syncTimeout\text{”} \wedge blockPool.height \leq blockPool.syncHeight)\Big)$$
$$\Rightarrow \diamond(state = \text{“}finished\text{”}) \qquad (\mathsf{TerminationByTO})$$

A global timeout guarantees that the protocol terminates, no matter what happens. TLC did not report liveness violations of TerminationByTO up to depth 14, which is indicated with $[\text{✓}]_{<14}$ in Table 1. However, the TLC search was not exhaustive, as we have terminated the model checker after 24 hours. We did not check liveness with Apalache, as it only supports safety at the moment. The more interesting property is “synchronization”, whose intuitive meaning is that when Fastsync terminates, it has reached the height of the blockchain. Let us formalize this as Sync1: To see that our modeling is precise, let us start with a property we know to be slightly wrong, namely, when the protocol finishes, it reaches the maximum height among the heights of the correct peers, i.e.,

$$state = \text{“}finished\text{”} \Rightarrow blockPool.height \geq MaxCorrectPeerHeight(blockPool) \qquad (\mathsf{Sync1})$$

The model checkers report counterexamples. One reason is that to verify a block h, one needs the commit signatures from block $h + 1$. We also observe that we do not require that the node that runs Fastsync needs to be connected to correct peers. Hence, we fix it in Sync2 by stating that height $MaxCorrectPeerHeight(blockPool) - 1$ should be reached when the node is connected to correct peers. This property also fails. This time we observe that a global timeout – that guarantees Termination – may terminate Fastsync before it has reached the maximal height. We add a precondition for “no timeout”, and call the property Sync3. Neither TLC, nor Apalache produce a counterexample up to computation depth 15 and 21, respectively.

The following two properties might appear to be intuitively correct, but the model checkers produce counterexamples. SyncFromCorrect states that the accepted blocks originate only from the correct processes. This property fails, as it does not consider that faulty peers may behave correct in an execution prefix (before they show faulty behavior). Thus, the initial intuition fails. CorrectNeverSuspected states that the correct peers are never removed from the peer set. This would be a desirable property, but the latest implementation V2 does not guarantee it.

5 Conclusions

We approach this work with a process-oriented goal in mind: By *Verification-Driven Development* [12] we understand a design process for distributed systems that makes it easier to test and verify the software. The re-design of the Fastsync protocol that resulted in a decomposition into state machines should be understood under this aspect. The design documents, namely the English and the TLA$^+$ specifications, are artifacts of this design process, and are means of communication between researchers, software engineers, and verification engineers. The structured English specification strikes a balance between mathematical rigor and readability. It serves as the base (i) for formal verification efforts in TLA$^+$ that

Table 1 Model checking results for TLC and Apalache against the high-level specification for the following parameters: 1 correct peer, 1 faulty peer, 4 blocks (Apple MacBook Pro 2019). The symbols in "result" are: found a bug ✗, and no bug up to given length [✓].

Property	TLC (6 CPUs, 13 GB)			APALACHE (1 CPU)	
	result	time	#states	result	time
Sync1	$[✗]_{=5}$	28s	1.1M	$[✗]_{=5}$	16s
Sync2	$[✗]_{=5}$	28s	1.1M	$[✗]_{=5}$	16s
Sync3	$[✓]_{<15}$	TO	875M	$[✓]_{<21}$	1h25m
Termination	$[✗]_{=7}$	7m25s	2.9M	*not supported*	
TerminationByTO	$[✓]_{<14}$	TO	461M	*not supported*	
SyncFromCorrect	$[✗]_{=9}$	8m34s	4.5M	$[✗]_{=9}$	1m23s
CorrectNeverSuspected	$[✗]_{=3}$	4s	126K	$[✗]_{=3}$	9s

will give precise semantics, and (ii) for implementations. The annotations with invariants, preconditions, and postconditions are very helpful for the software engineers to guide the implementation.

The formalization also led to a better understanding of the liveness properties that we expect and want from blockchain synchronization protocols. It also improved the understanding of the discrepancies between the current implementations (Fastsync V0, V1, and V2). We have found several liveness issues that come from unexpected behavior of faulty peers. For instance, rather than reporting bad blocks, faulty peers may be very slow in reporting good blocks. If they report them slower than the blockchain grows, but fast enough to not lead to a timeout at the node, V2 may never terminate. This highlights that a vital requirement had not been explicitly captured before, namely, a relationship between timeout duration, block generation rate, and message end-to-end delays. We made these requirements explicit as part of the English specification; they constitute timing assumptions upon which the protocol is based. As this is closely related to real-time, we are not able to capture and reproduce this with TLA$^+$. However, TLA$^+$ counterexamples and the English specifications helped us in isolating this scenario.

For safety verification, we can replace a timeout by a non-deterministic event that may occur at any time. For liveness we have to treat the relation of timeouts to message delays and processing times precisely. The extensive use of timeouts in the current implementation poses a challenge to liveness verification. Some of our current research challenges are therefore timeouts, and we are interested in answering the following questions: How to limit timeouts in the implementation? What is the most effective way to use timeouts in the implementation in order to stay precise in the verification? How can we capture the relation of the (local) timeouts to (global) message delays in model checking? We keep these challenges for future work.

Model checkers and the produced counterexamples were quite helpful in understanding and refining the protocol properties. After refining the protocol, which results in larger state space, we found that TLC could not reach error states within the reasonable time frame of one hour. However, Apalache was still finding errors within 10 minutes, which was still interactive enough for us. As future work, we also plan to find an inductive invariant and prove its correctness with Apalache (for fixed but larger parameters).

References

1 APALACHE: a symbolic model checker for TLA$^+$. https://github.com/konnov/apalache/, 2020. Last accessed: May 20, 2020.

2 Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of Blockchains. Master's thesis, University of Guelph, 2016. URL: http://hdl.handle.net/10214/9769.

3 Ethan Buchman and Jae Kwon. Cosmos whitepaper: a network of distributed ledgers, 2016. URL: https://cosmos.network/resources/whitepaper.

4 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938*, 2018. URL: https://arxiv.org/abs/1807.04938.

5 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

6 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J.ACM*, 35(2):288–323, 1988.

7 Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *PACMPL*, 3(OOPSLA):123:1–123:30, 2019.

8 Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ toolbox. In *F-IDE@FM 2019*, pages 50–62, 2019.

9 Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

10 Informal Systems. Fastsync – English specification, 2020. URL: https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/fastsync/fastsync.md.

11 Informal Systems. Fastsync – TLA$^+$ specification, 2020. URL: https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/fastsync/fastsync.tla.

12 Informal Systems. Verification-Driven Development: An Informal Guide, 2020. URL: https://github.com/informalsystems/VDD/blob/master/guide/guide.md.

13 ADR 043: Blockhchain reactor riri-org, 2020. URL: https://github.com/tendermint/tendermint/blob/master/docs/architecture/adr-043-blockchain-riri-org.md.

14 Tendermint core, reference implementation in Go, 2020. URL: https://github.com/tendermint/tendermint.

15 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.