Tendermint Blockchain Synchronization: Formal Specification and Model Checking

Sean Braithwaite[2], Ethan Buchman[1], Igor Konnov[3(✉)], Zarko Milosevic[2], Ilina Stoilkovska[3], Josef Widder[3], and Anca Zamfir[2]

[1] Informal Systems, Toronto, Canada
ethan@informal.systems
[2] Informal Systems, Lausanne, Switzerland
{sean,zarko,anca}@informal.systems
[3] Informal Systems, Vienna, Austria
{igor,ilina,josef}@informal.systems

Abstract. Blockchain synchronization is one of the core protocols of Tendermint blockchains. We describe our recent efforts on formal specification of the protocol and its implementation, and present model checking results for small parameters. We demonstrate that the protocol quality and understanding can be improved by writing specifications and applying model checking to verify their properties.

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 [6]. Tendermint is particularly popular for proof-of-stake blockchains, and constitutes a core component of the Cosmos Project [7]. At the heart of the Cosmos Project is the InterBlockchain Communication protocol (IBC) for reliable communication between independent BFT SMs; what TCP is for computers, IBC aims to be for blockchains.

Multiple Tendermint-based blockchains run in production on the public Internet for over a year, with new ones launching regularly. They carry billions of dollars of cumulative value in the market capitalizations of their respective cryptocurrencies. One of the primary deployments is the Cosmos Hub blockchain [24]. It is operated by a diverse set of 125 consensus forming nodes; they are connected over an open-membership gossip network consisting of hundreds of other nodes.

Tendermint was the first proof-of-stake blockchain system to apply traditional BFT consensus protocols at its core [18]. The Tendermint BFT consensus protocol constitutes a modern implementation of the consensus algorithm for Byzantine faults with Authentication from [11], built on top of an efficient gossiping layer. The latest description of the consensus protocol can be found in the technical report [8]. Tendermint has been a source of inspiration for a wide variety of blockchain systems that have followed [9,26], though few, if any, have achieved its level of maturity in production.

Supported by Interchain Foundation (Switzerland).

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 471–488, 2020.
https://doi.org/10.1007/978-3-030-61362-4_27

The reference implementation of the Tendermint software is written in Go [25]. 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 that should 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 the list of available peers and selecting 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 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 [13] and TLA$^+$ [14].

Fastsync. A full node that connects to a Tendermint blockchain needs to synchronize its state to the latest global state of the network. This network state is defined by the sequence of blocks that the system has decided upon. These blocks are numbered continuously, and a block's number is called its height, and the height of the most recently added block is called the current height of the blockchain. Thus, another way to put the blockchain synchronization problem is the need to catch-up to a recent height of the blockchain. One way to achieve this is 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. (Fastsync can be also used by a fresh node that connects to a blockchain; the node starts with the genesis file, i.e., the initial block.) 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(b1)` that contains the block. In a Tendermint blockchain, the commit (signed votes messages) for block `h` is contained

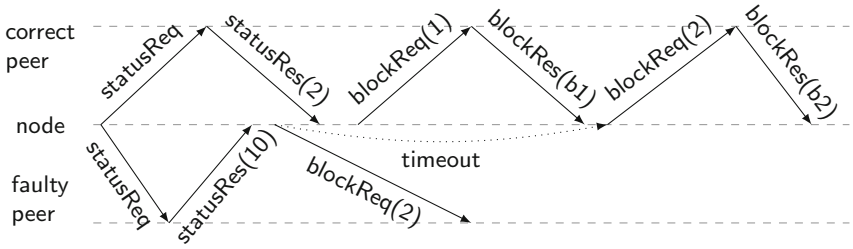

Fig. 1. A Fastsync execution for a fully unsynchronized node of height 1

in block `h+1`, and thus a node performing Fastsync must receive two sequential blocks before it can verify fully the first of them. If verification succeeds, the first block is accepted; if it fails, both blocks are rejected, as it is unknown 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 are intricate, in particular due to the presence of faulty peers. Moreover, the intuitive safety and liveness properties often fail to hold, and one has to refine the temporal formulas used to encode these properties. This effort significantly improves understanding of the protocol boundaries and of its guarantees.

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 [1]. 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, the protocol logic is decoupled from IO by using three concurrent threads of execution: a `scheduler`, a `processor`, and a `demuxer`, as per Fig. 2.

Both the scheduler and processor are structured as finite state machines with input and output events. Inputs 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 demuxer routine is responsible for all IO, including translating between internal events and network IO messages, and routing events between components.

The task of the scheduler is to ensure that a number of blocks are always available for verification by the processor. To achieve this, the scheduler tracks

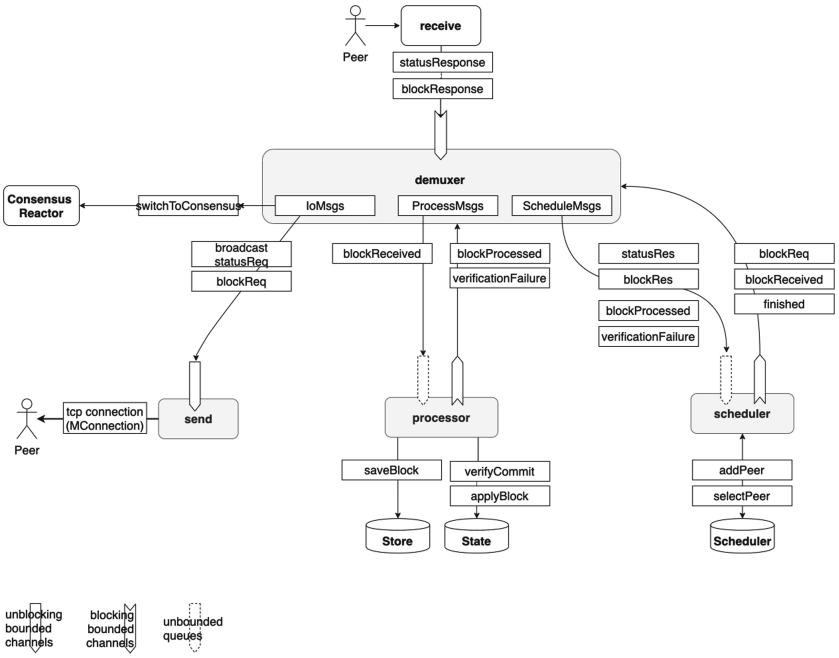

Fig. 2. Communication between components in the Fastsync implementation [1].

peers and their heights (via `statusReq` and `statusRes` messages and events) and makes block requests (`blockReq`) to peers. In order not to overload any one peer, the block requests are equally spread across the peers. The block responses (`blockRes`) are forwarded to the processor (`blockReceived`) for verification. The scheduler maintains lists of pending block requests and block responses for each peer. Peers that (i) are unresponsive, (ii) sent blocks which cause verification failure, or (iii) sent unsolicited blocks, are removed, together with any unprocessed blocks from these peers. In case there are pending block requests associated with the removed peers, these blocks are requested from other peers.

The processor performs block processing, including verification of consensus signatures and execution of all transactions, which is performed in increasing order of the block height. The blocks that are successfully verified are stored in `Store`, and the chain `State` is updated with the result of transaction execution (cf. Fig. 2). The result of the block processing (`blockProcessed` or `verificationFailure`) is sent to the scheduler via the `demuxer` routine. The scheduler keeps track of the block execution height and triggers termination—at the maximum peer height (`finished`). Once a node terminates executing Fastsync, it continues by executing the Tendermint consensus protocol to stay up to date with the latest blockchain changes.

3 Specifications in English and TLA$^+$

In addition to protocol verification, we are elaborating a verification-driven development process. Our goal is to have a design process that has researchers, verification engineers, and software engineers in the loop during the design and development of protocols. Fastsync was the first protocol where we adopted that process in part. By the time we started our verification efforts, the blockchain synchronization protocol had already been designed using a classic engineering process, whose artifacts include architecture decision records and English specifications that focus on data structures and APIs. There had already been two implementations of the protocol (V0 and V1), and our software engineering team was in the process of wrapping-up a third implementation (called V2) whose primary goal was increased testability.

In this project we started with joint sessions of software engineers, researchers and verification engineers, in which we wrote first TLA$^+$ specifications together. In order to better understand how TLA$^+$ can be of most help to us, we then wrote different specifications that focus, e.g., on the business logic, the local concurrency architecture of the implementation of V2, and the protocol level; we published the latter at [14]. While TLA$^+$ specifications provide precise semantics, we found that English specifications are a valuable tool for communication, both within the project, but also to others who are interested in the protocols. We thus developed a structure for English specifications [13]. While in the Fastsync project, this structured English specification was written after-the-fact, in the verification-driven development process and our current projects, it is the origin and deals as reference both for implementations written by the software engineers as well as for the TLA$^+$specification written by the verification engineers.

We chose TLA$^+$ for two reasons: (1) We have acquired a good understanding of the language by developing the Apalache model checker [16], and (2) TLA$^+$ has been successfully used by systems designers, e.g., at Amazon [5,23].

Structured Specification in English. We start our formalization by a structured English specification [13], that consists of four parts:

1. *Blockchain.* Formalization of relevant properties of the chain and its blocks.
2. *Sequential problem statement.* Here we consider the blockchain as a growing list of blocks, and define what we expect from the blockchain synchronization protocol with respect to this list. This specification is sequential. 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, which introduces uncertainty that has to be reflected in the distributed protocol as well as in the distributed problem statement.
3. *Distributed aspects.* Here we introduce the computational model and the refinement of the sequential problem statement. The computational model specifies assumptions about the system, such as assumptions on the message delays, process faults, etc. As a result, the problem statement is restricted to some fairness constraints, e.g., it is preconditioned by the process being connected to at least one correct peer.

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

Specifications in TLA$^+$. The structure of the English specification highlights interesting properties of the protocols and points to some issues. As it is written in natural language, the English specification is ambiguous. We have written three TLA$^+$ specifications that provide precise semantics, 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 properties. HLS was mainly written by the distributed system 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 closer to the implementation. It is mainly written by distributed system engineers.
- *Concurrency specification (CRS).* As discussed above, the V2 implementation uses several threads that communicate via queues. To formally capture this, we wrote a specification that models threads and message queues.

In the following, we focus on the high-level specification in English and TLA$^+$ [14]. In Sects. 4–7, we give the main abstractions and insights about the specifications. The TLA$^+$ specification has about 800 lines, hence we omit presenting it in full detail. In addition, it is parameterized by the blockchain length, and the set of peers. By fixing these parameters, we check its safety and liveness with TLC [17] and Apalache [16], detailed in Sect. 8.

4 The Blockchain Specification in English and TLA$^+$

A block is a data structure that contains application information (e.g., transactions) as well as metadata needed for the protocols. As we are interested in the blockchain synchronization, some of this metadata is relevant for our formal model. Figure 3 illustrates three blocks of a Tendermint blockchain. The blocks are consecutively numbered, and each block is assigned a number, called its *height*. As the blocks are a result of consensus by *validators*, the validity of a block is confirmed if a quorum of the validators signed (a hash of) a block. The validator membership in Tendermint changes over time, and is indeed a result of consensus itself. Moreover, the validators have an associated voting power, which is not necessarily uniform. For a block, a *validator set* is a set of pairs of IDs of validators and their associated voting power. The quorum we referred to

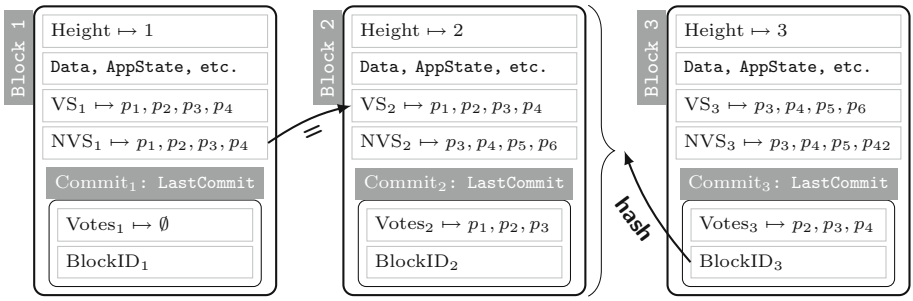

Fig. 3. The block structure in Tendermint blockchain. The fields VS and NVS denote the current and next *validator sets* of a block. Note that Votes_3 contains more than $^2/_3$ of voting power in NVS_1.

above thus corresponds to the set of validators that have more than $^2/_3$ of the total voting power in a given validator set.

To capture this, in a block of height i, the validator set of the current block is stored in the field VS_i, the validator set for the next block is stored in NVS_i, and the signed messages that confirm block i are stored in the field Commit_{i+1} of the block at height $i+1$. The nodes whose signatures are in Commit_{i+1} must be a subset of the set VS_i of validators at height i. A node running the blockchain synchronization protocol checks the quorums and signatures in order to locally confirm whether a downloaded block originates from the blockchain. Therefore, it is crucial to capture validator sets and commits in our formal specification.

TLA$^+$ *Specification.* The blockchain data structures in the implementation are quite rich. To render model checking possible, we abstract blocks so that the safety and liveness properties of the protocol are preserved, while the potential search space becomes finite and relatively small. We call this model Tinychain, and present it below. A compact version of TLA$^+$ code is shown in Listing 1.

First, note that in general, the number of blocks on a blockchain may grow unboundedly. As a result, the field height is also an unbounded integer. Hence, we parameterize the blockchain with the maximal height MAX_HEIGHT in line 3.

Second, the structure of the validator sets is not essential for the protocol. The few required properties of the blocks, hashes, and validator sets are axiomatized at an abstract level. Hence, we add two more parameters to the specification in lines 4–5: VALIDATOR_SETS and NIL_VS. The parameter VALIDATOR_SETS can be a set of any values, not necessarily sets. We usually define it as a set of uninterpreted constants, e.g., as {"$S1$", "$S2$", "$S3$"}. The parameter NIL_VS encodes an abstract set outside of VALIDATOR_SETS, e.g., "*Nil*".

Third, the field blockID in a commit and a block hash are needed only to test block equality, when trying to find out whether a block has been sent by a faulty or a correct peer. Hence, for every block, instead of its hash and the hash of the previous block in the commit, we introduce two predicates: hashEqRef and blockIdEqRef. These predicates restrict the behavior of faulty peers by comparing

Listing 1. Abstract blockchain for Fastsync in TLA$^+$

```
1  ---------------- MODULE Tinychain ----------------------
2  EXTENDS Integers
3  CONSTANTS MAX_HEIGHT, \* the maximal number of blocks
4            VALIDATOR_SETS, \* a set of abstract sets
5            NIL_VS \* a special abstract set outside of the above set
6
7  IsCorrectBlock(chain, h) ≜
8   ∧ chain[h].height = h \* the height is right
9   ∧ h > 1 ⇒
10    ∧ chain[h].VS = chain[h − 1].NVS \* the validators are from the prev. block
11    ∧ chain[h].lastCommit.voters = chain[h − 1].VS \* and they are the voters
12
13 IsCorrectChain(chain) ≜
14  LET OkCommits ≜ [blockIdEqRef: {TRUE}, voters: VALIDATOR_SETS]
15      OkBlocks ≜ [height: 1..MAX_HEIGHT, hashEqRef: {TRUE},
16                  wellFormed: {TRUE}, VS: VALIDATOR_SETS,
17                  NVS: VALIDATOR_SETS, lastCommit: OkCommits]
18  IN
19  ∧ chain ∈ [1..MAX_HEIGHT → OkBlocks]
20  ∧ ∀ h ∈ 1..MAX_HEIGHT: IsCorrectBlock(chain, h)
21 ==========================================================
```

a block against the reference chain. We explain this in Sect. 6. Finally, we abstract all simple structure tests with the predicate wellFormed, whose negation models that a block by a faulty peer does not pass simple consistency tests.

Having the abstract block structure, we define the predicate IsCorrectChain in lines 13–20 that constrains the block sequence chain. Line 19 restricts chain to be a function of a block height to a block from OkBlocks, that is, a set of records that is defined in lines 15–17. (The notation 'a: B' constrains the record field a to range over the set B). Using the predicate IsCorrectChain, we define the *reference chain*, to which the correct peers are synchronized.

5 The Blockchain Synchronization Problem in English

Sequential Problem Statement. The synchronization protocol must satisfy:

Sync. Let k 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 $terminationHeight \geq k - 1$. (Fastsync cannot synchronize to the maximum height k as in Tendermint, verification of block at height h requires the commit from the block at height $h + 1$.)

Liveness. Fastsync eventually terminates.

Safety. Upon termination, the application state is the one that corresponds to the blockchain at height $terminationHeight$.

Observe that **Sync** requires $terminationHeight \geq k - 1$. As in Tendermint the verification of a block at height h requires the commit from the block at height $h + 1$, Fastsync cannot synchronize to the height k. Also note that the blockchain may grow during the execution of Fastsync, that is, its height may increase before Fastsync terminates. In **Sync** we require to reach at least the height of the blockchain when the protocol starts (it is a minimal requirement), while we allow the protocol to go larger heights in case the blockchain grows.

Distributed Aspects and Faulty Peers. We consider a node FS that performs Fastsync by communicating with peers from a set *PeerIDs*, some of which may be faulty. We assume the authenticated Byzantine fault model [11] in which no peer (faulty or correct) may break digital signatures, but otherwise, no additional assumption is made about the internal behavior of faulty peers. That is, faulty peers are only limited in that they cannot forge messages. We do not make any assumptions about the number or ratio of correct/faulty peers.

Communication between the node FS and all correct peers is reliable and bounded in time: there is a message end-to-end delay Δ such that if a message is sent at time t by a correct process to a correct process, then it will be received and processed by time $t + \Delta$.

Without the assumption that *PeerIDs* contains a correct full node, no protocol can solve the sequential problem. To relax the problem in the unreliable distributed setting, we consider two kinds of termination (successful and failure). We specify below under what conditions Fastsync ensures successful termination and still solves the sequential problem.

Distributed Problem Statement. In the distributed setting, the synchronization protocol must satisfy:

Sync. 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 $terminationHeight \geq maxh - 1$.

Liveness. Fastsync eventually terminates: either successfully or with failure.

Non-abort. If there is one correct process in *PeerIDs*, Fastsync never terminates with failure.

Safety. The same as Safety in the sequential problem statement.

In the distributed setting, non-abort in conjunction with liveness ensures that if there is a correct process in *PeerIDs*, then Fastsync never terminates with failure, that is, it will terminate successfully.

6 Correct and Faulty Peers in TLA$^+$

Section 5 introduces a number of assumptions about correct and faulty peers. In this section, we give an idea about formalization of these assumptions in TLA$^+$. The node starts with a finite set of peers, which can shrink when the node suspects peers of being faulty. The initial set of peers is partitioned into two

Listing 2. Alternating composition of the node and peers in TLA$^+$

```
1  CONSTANTS CORRECT, FAULTY, ... \* the sets of correct and faulty peers
2  VARIABLES state, blockPool \* the node's state variables
3  VARIABLES peersState \* the peer's state variables
4  VARIABLES turn, inMsg, outMsg \* the composition variables
5  /* specification of the node and the peers */
6  ...
7  Init ≜
8    ∧ IsCorrectChain(chain) \* initialize the chain up to MAX_HEIGHT
9    ∧ InitNode ∧ InitPeers \* initialize the node and the peers
10   ∧ turn = "Peers" \* the first turn is by the peers
11   ∧ inMsg = NoMsg \* no incoming message from the peers to the node
12   ∧ outMsg = [type ↦ "statusRequest"] \* a request from the node to a peer
13
14 Next ≜
15   IF turn = "Peers"
16   THEN NextPeers ∧ turn' = "Node" ∧ UNCHANGED ⟨state, blockPool, chain⟩
17   ELSE NextNode ∧ turn' = "Peers" ∧ UNCHANGED ⟨peersState⟩
```

subsets: CORRECT and FAULTY. As expected, the node specification must not refer to either of these subsets, as the node is not able to distinguish the faulty nodes from the correct ones in the distributed setting.

Composition. Listing 2 shows the specification structure. The predicate Init constrains the initial states, whereas the predicate Next constrains the transition relation of the system. We model the distributed system as two components: the node and its peers. They 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 by flipping the variable turn: The odd turns belong to the node, and the even turns to the peers.

This approach is simple yet powerful. On one hand, it dramatically decreases the state space, as there are no queues, and alternation produces significantly fewer states than the disjunction, which would correspond to interleaving: NextNode ∨ NextPeers. On the other hand, it does not decrease precision, as the peers consume and produce messages independently of one another. Moreover, this approach allows us to easily formulate fairness in the system as weak fairness over the variable `turn`.

Correct Peers. The correct peers non-deterministically send their status (the chain height) to the node and respond to its requests. For example, if the node requests a block of height 5 from a peer "c3", the peer "c3" sends its block. The peers may also join and leave the network. We omit the technical details.

Faulty Peers. The faulty peers are authenticated Byzantine: In addition to the behavior of the correct peers, they may send unsolicited or corrupt messages, or

ignore the requests. As discussed in Sect. 4, the blockchain uses hashes, which limits the power of the faulty peers in sending blocks. This is where the hashing predicates come into play. The essential piece of TLA$^+$ code is given below:

```
1  SendBlockResponseMessage(...) ≜
2   ∨ ... \* a response by a correct peer to a node's request
3   ∨ ∃ peerId ∈ FAULTY: \* a faulty peer can always send a block
4     ∃ block ∈ Blocks:
5       ∧ block.height = height \* height mismatch is easy to detect
6       ∧ block.hashEqRef ⇒ block = chain[height] \* no hash forging
7       ∧ (height > 1 ∧ block.lastCommit.voters = chain[height − 1].VS)
8             ⇒ block.lastCommit.blockIdEqRef \* no equivocation by the validators
9       ∧ inMsg' = [type ↦ "blockResponse", peerId ↦ peerId, block ↦ block]
10      ∧ ...
```

Line 6 forces a faulty peer to produce the block as on the chain, when the predicate block.hashEqRef holds true, that is, the block hash matches the hash of the reference block on the chain. This is exactly the semantics of a perfect hash. Line 8 is perhaps less obvious. Intuitively, it says that if the block contains a commit for the previous block, and the voters in the commit coincide with the validators of the previous block on the chain, then the hash in the commit must be equal to the hash of the previous block on the chain. (The implementation tests whether voters constitute over $2/3$ of the VS voting power. However, we find our approximation sufficient for model checking.) Importantly, with Boolean hashEqRef and blockIdEqRef, we model the scenarios: (1) the hashes are equal to the reference hash; (2) the hashes are equal to a number different from the reference hash; and (3) the hashes are not equal.

7 The Node Protocol in English and TLA$^+$

Recall Fig. 1, that shows a typical execution of Fastsync. Using `statusReq`, the node FS asks a peer about its current height, that is, the length of the prefix of the blockchain the peer has stored. Each peer responds with `statusRes(h)`, where `h` is its current height. By collecting these responses, FS gets information about which peer has which blocks, and uses this information (1) to compute its target height (the maximum height its peers know of) and (2) to decide which blocks to request from which peer. It requests a block of height h from a peer by sending `blockReq(h)`, and a peer responds by sending `blockRes(bh)`, that contains a block of height h. FS stores all the received blocks locally, and checks all the signatures and hashes to make sure that there are no invalid blocks that could have been provided by faulty peers.

As the implementation uses external events (message reception) and timeouts to make progress, we have chosen to describe the model in terms of the following functions, that are triggered by events:

QueryStatus(): regularly (at least 2Δ, now 10 s) queries all peers from *PeerIDs* for their current heights by sending `statusReq` to all peers.

CreateRequest(): regularly checks whether certain blocks have no open request. If a block does not have an open request and its height is h, FS requests one from a peer. It does so by sending `blockReq(h)` to one peer.

In our specification, we leave the strategy of peer selection unspecified. Various implementations of Fastsync differ in this aspect. Version V2 (see Sect. 2) selects a peer p with the minimum number of pending requests that can serve the required height h, that is, whose height is greater than or equal to h.

When the messages `statusRes(h)` or `blockRes(b)` are returned from the peer at address `addr`, the following functions are called, respectively:

OnStatusResponse(addr Address, h int): The full node with address *addr* returns its current height. The function updates the height information about *addr*, and may also increase the target height.

OnBlockResponse(addr Address, b Block): The full node with address *addr* returns a block. It is added to blockstore. Then the auxiliary function `Execute` is called.

Execute(): Iterates over the received blocks. It checks soundness of the blocks (hashes, signatures, etc.), and executes the transactions of a sound block and updates the application state.

FS keeps track of several performance metrics: the last time a peer responded, the throughput to a peer, etc. If a peer p has not provided a block recently or it has not provided a sufficient amount of data, then p is removed from *PeerIDs*. Fastsync V2 schedules a timeout whenever a block is executed, that is, when the height is incremented. If the timeout expires before the next block is executed, Fastsync terminates. If this happens, then Fastsync terminates with failure. Otherwise it terminates successfully when it reaches the target height.

We omit the details about the other functions. Figure 4 shows an example of how we specify functions in the English specification. Rather than using pseudo code, we specify functions mainly using preconditions and postconditions. They have a clear meaning to verification engineers, but also give the software engineers a precise understanding of what the function should do without restricting them in how to satisfy these requirements in the source code.

TLA$^+$ *Specification.* We omit technical details of encoding the node communication. The implementation V2 relies on several timeouts to guarantee termination. Although precise modeling of time and timeouts is possible in TLA$^+$ [20], it obviously leads to state explosion. Hence, we simply model timeouts with non-determinism and weak fairness.

Listing 3 shows the block verification logic. Interestingly, VerifyCommit checks the predicates commit.blockIdEqRef and block.hashEqRef. There are two valid options with respect to the hash *hash* of the reference block: Either both the hashes are equal to *hash*, or they are both different from *hash*.

`func OnBlockResponse(addr Address, b Block)`

- Comment
  - if after adding block b, blocks of heights $height$ and $height{+}1$ are in $blockstore$, then `Execute` is called
- Expected precondition
  - $pendingblocks(b.Height) = addr$
  - b satisfies basic soundness
- Expected postcondition
  - if function `Execute` has been executed without error or was not executed:
    - $receivedBlocks(b.Height) = addr$
    - $blockstore(b.Height) = b$
    - $peerTimeStamp[addr]$ is set to a time between invocation and return of the function.
    - $peerRate[addr]$ is updated according to size of received block and time passed between current time and last block received from this peer (addr)
- Error condition: if precondition is violated: $addr$ not in $PeerIDs$; reset $pendingblocks(b.Height)$ to nil;

Fig. 4. Example of a function definition in the English specification

Listing 3. Block execution logic in TLA$^+$

```
1  VerifyCommit(block, commit) ≜
2    commit.voters = block.VS ∧ commit.blockIdEqRef = block.hashEqRef
3
4  ExecuteBlocks(pool) ≜
5    ... \* get stored blocks b0, b1, b2 for heights h−1, h, h+1
6    IF b1 = Nil ∨ b2 = Nil \* no two next consecutive blocks
7    THEN pool
8    ELSE IF b0.NVS ≠ b1.VS ∨ ¬VerifyCommit(b0, b1.lastCommit)
9         THEN RemovePeers({Sender(b1.height)}, pool)
10        ELSE IF ¬VerifyCommit(b1, b2.lastCommit)
11             THEN RemovePeers({Sender(b1.height), Sender(b2.height)}, pool)
12             ELSE [pool EXCEPT !.height = pool.height + 1]
```

8 Model Checking with TLC and Apalache

While developing TLA$^+$ specifications, we were using TLA$^+$ Toolbox and the model checker TLC [17]. We also checked the safety properties with the new symbolic model checker Apalache [2,16]. So far, we have checked the specifications for tiny parameters, such as 1 to 3 peers and Blockchain height from 3 to 5. Table 1 summarizes the results and running times of TLC and Apalache. A central temporal property is the protocol's Termination:

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

where $\text{WF}_x(A)$ in TLA$^+$ forces weak fairness of action A, if it changes x.

In 7 min, TLC finds a bug: Faulty peers may keep the node busy by sending blocks or joining and leaving the network. The more precise property TerminationByTO states that the protocol terminates, if there is a global timeout:

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

In this case, TLC finds no bug, though it does not finish state exploration. (We did not run Apalache, as it only supports safety.) We found that it is extremely hard to formulate the “normal” termination property in the presence of faults, i.e., without involving a timeout. We also formulated the property TerminationCorrect: The protocol terminates without a timeout, provided that all peers are correct. TLC exhaustively proves this property for one correct peer.

The more interesting property is “synchronization”, whose intuitive meaning is that by the time Fastsync terminates, it reaches the height of the blockchain. Let us formalize this as Sync1: To see that our modeling is precise, we start with a property that is slightly wrong, namely, when the protocol finishes, it reaches the maximum height among the heights of the correct peers, i.e.,

$$\Box\big(state = \text{“}finished\text{”} \Rightarrow blockPool.height \geq MaxCorrectPeerHeight(blockPool)\big)$$

Both 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 the node running Fastsync is not always 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 TerminationByTO — 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 (for executions up to 20 steps).

We formulated the invariant CorrectBlocks: The synchronized blocks have enough votes and contain correct signatures and hashes (the correct peers produce only the blocks that satisfy this property). By running Apalache, we found that this property was violated by the specification. After code inspection, we realized that the implementation executes an extra consistency test that was not captured in the specification (as it was not clear that it is part of the protocol). After fixing the specification, we have found no further counterexamples.

Both model checkers quickly find counterexamples for the following two properties that might appear to be correct. SyncFromCorrect states that the accepted blocks originate only from the correct processes. This property fails, as it does not account for the cases where faulty peers behave correct in an execution prefix (before showing faulty behavior). NoSuspectedCorrect states that the correct peers are never removed from the peer set. This would be a desirable property, but the current implementation V2 does not guarantee it.

Finally, TLC is quite fast when checking properties in the configuration with one correct peer. However, adding just one faulty peer blows up the state space,

Table 1. Model checking results for TLC and APALACHE against the high-level specification for 1 correct peer, 0/1 faulty peers, and 4 blocks. The experiments were run in an AWS instance equipped with 32GB RAM and a 4-core Intel® Xeon® CPU E5-2686 v4 @ 2.30GHz CPU. The notation: ✗ for "found a bug at depth k", $[\checkmark]_{<k}$ for "found no bug up to depth k", ✓ for "correct" (exhaustive search), TO for "timeout" (24 h).

Property	TLC (4 CPUs, 28 GB)							APALACHE (1 CPU)	
	1 correct/0 faulty/4 blocks				1 correct/1 faulty/4 blocks			1 correct/1 faulty/4 blocks	
	result	time	#states	diameter	result	time	#states	result	time
Sync1	$[\text{✗}]_{=5}$	13s	9K	5	$[\text{✗}]_{=5}$	28s	1.1M	$[\text{✗}]_{=5}$	16s
Sync2	$[\text{✗}]_{=5}$	7s	9K	5	$[\text{✗}]_{=5}$	28s	1.1M	$[\text{✗}]_{=5}$	16s
Sync3	✓	37s	68K	15	$[\checkmark]_{<15}$	TO	875M	$[\checkmark]_{<21}$	1h25m
Termination	$[\text{✗}]_{=8}$	11s	25K	8	$[\text{✗}]_{=7}$	7m25s	2.9M	*not supported*	
TerminationByTO	✓	33s	68K	15	$[\checkmark]_{<14}$	TO	461M	*not supported*	
TerminationCorrect	✓	42s	68K	15	*not applicable*			*not supported*	
CorrectBlocks	✓	31s	68K	15	$[\checkmark]_{<15}$	TO	873M	$[\checkmark]_{<16}$	1h53m
SyncFromCorrect	$[\text{✗}]_{=9}$	9s	33K	9	$[\text{✗}]_{=9}$	8m34s	4.5M	$[\text{✗}]_{=9}$	1m23s
NoSuspectedCorrect	$[\text{✗}]_{=3}$	6s	1K	9	$[\text{✗}]_{=3}$	4s	126K	$[\text{✗}]_{=3}$	9s

which prevents TLC from finishing state exploration. In this case APALACHE performs better. However, it runs bounded model checking, which gives us only bounded safety, that is, up to the predefined execution length.

9 Conclusions and Future Work

We approach this work with a process-oriented goal in mind: By *Verification-Driven Development* [15] 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 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 a base for (i) formal verification efforts in TLA$^+$, that provide precise semantics, and (ii) implementations. The annotations with invariants, pre- and postconditions are very helpful for the software engineers to guide the implementation.

The gap between informal English specifications, and formal TLA$^+$ specifications and the implementations is still a research challenge. As future work we will consider semi-formal methods that address this formalization gap. For instance, we have found that the distributed system engineers have a hard time specifying precise liveness properties, which truly requires one to think about temporal operators. Specifying fairness is the most challenging specification task in case of fault-tolerant distributed systems. Instead of asking the engineers to write the temporal properties directly, we could instantiate specification patterns [4] that collect the most-often occurring shapes of temporal formulas. This can be

done with the help of graphical tools such as ProPas [12]. In a more general setting, we could use the boilerplates approach offered by CESAR [3]. This is a specification method that uses restricted English grammar, where a designer selects the boilerplates that fit the specific requirement, and fills the details to arrive at a complete specification.

The formalization also led to a better understanding of the liveness properties that we expect and want from blockchain synchronization protocols, and to an improved awareness regarding the differences 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 captured before, namely, a relationship between timeout duration, block generation rate, and message end-to-end delays. As this issue is closely related to real-time, we are not able to directly capture it and reproduce it 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 future research challenges to liveness verification. Some of our current research questions are: 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?

The counterexamples produced by the model checkers were quite helpful in understanding and refining the protocol properties. After refining the protocol with small hashes, which resulted in a larger state space, TLC could not reach error states within the reasonable time frame of one hour. In contrast, Apalache was finding errors within 10 min, which was still interactive enough for us. Once we felt confident in the protocol after debugging it with Apalache, we shrinked the state space by introducing Boolean abstraction of hashes, allowing TLC to also report errors. As future work, we plan to find an inductive invariant and prove its correctness with Apalache (for fixed but larger parameters).

The language of TLA$^+$ built around refinement [19,21]. In the classical approach, one starts with an abstract specification A of a protocol and produces a more detailed specification C. To show refinement, the user substitutes the variables of A with expressions over the variables of C, which results in a specification $\gamma(A)$, and then proves that the behaviors of C can be replayed by $\gamma(A)$. It suffices to prove two statements: (1) the initial states of C are a subset of the initial states of $\gamma(A)$, formally, $C!Init \Rightarrow \gamma(A)!Init$; and (2) the transitions of C are a subset of the transitions of $\gamma(A)$, formally, $C!Next \Rightarrow \gamma(A)!Next$. To prove the steps (1) and (2) for all values of the parameters, the user has to use TLA$^+$ Proof System [10]. To debug the statements (1) and (2) for small parameters, one can use the model checkers TLC and Apalache.

In our case, the design flow was in the opposite direction. We started with the existing implementation and wrote several specifications of the protocol in TLA$^+$. Technically, we could construct a refinement mapping between the low-level specification and the high-level specification and check it with the model checkers for the small parameters. However, the potential feedback from this step seemed to be negligible, in comparison to checking safety and liveness of the protocol. A more pressing issue for us is how to establish conformance of the protocol implementation (in Google Go) to the TLA$^+$ specification. To this end, we are currently developing a model-based testing tool, which produces system tests out of TLA$^+$ traces, as generated by TLC and Apalache as output.

An alternative approach would be to use Ivy [22] instead of TLA$^+$ tools. The authors of Ivy demonstrated how one can do refinement and parameterized verification of consensus protocols with their tool. Their approach requires creative massaging of the specification with the goal of simplifying the SMT theory and transforming the constraints in the EPR form. We found that is much easier to explain TLA$^+$ to the engineers than uninterpreted first-order logic. It would be great to unite the clarity of TLA$^+$ and effectiveness of Ivy.

References

1. ADR 043: Blockhchain reactor riri-org (2020). https://github.com/tendermint/tendermint/blob/master/docs/architecture/adr-043-blockchain-riri-org.md
2. APALACHE: a symbolic model checker for TLA$^+$ (2020). https://github.com/informalsystems/apalache/. Accessed 10 Aug 2020
3. Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F., Gnaga, R.: Automatic checking of conformance to requirement boilerplates via text chunking: an industrial case study. In: ESEM (2013)
4. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP, pp. 7–15 (1998)
5. Brooker, M., Chen, T., Ping, F.: Millions of tiny databases. In: USENIX, pp. 463–478 (2020)
6. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Master's thesis, University of Guelph (2016). http://hdl.handle.net/10214/9769
7. Buchman, E., Kwon, J.: Cosmos whitepaper: a network of distributed ledgers (2016). https://cosmos.network/resources/whitepaper
8. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. arXiv preprint arXiv:1807.04938 (2018). https://arxiv.org/abs/1807.04938
9. Buterin, V., Griffith, V.: Casper the friendly finality gadget. arXiv preprint arXiv:1710.09437 (2017)
10. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA$^+$ proofs. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 147–154. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_14
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
12. Filipovikj, P., Seceleanu, C.: Specifying industrial system requirements using specification patterns: a case study of evaluation with practitioners. In: ENASE, pp. 92–103 (2019)

13. Informal Systems: Fastsync - English specification (2020). https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/fastsync/fastsync.md
14. Informal Systems: Fastsync - TLA$^+$ specification (2020).https://github.com/informalsystems/tendermint-rs/blob/master/docs/spec/fastsync/fastsync.tla
15. Informal Systems: Verification-Driven Development: An Informal Guide (2020). https://github.com/informalsystems/VDD/blob/master/guide/guide.md
16. Konnov, I., Kukovec, J., Tran, T.: TLA+ model checking made symbolic. PACMPL **3**(OOPSLA), 123:1–123:30 (2019)
17. Kuppe, M.A., Lamport, L., Ricketts, D.: The TLA+ toolbox. In: F-IDE@FM 2019, pp. 50–62 (2019)
18. Kwon, J.: Tendermint: consensus without mining. Draft v. 0.6, fall **1**(11) (2014)
19. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
20. Lamport, L.: Real-time model checking is really simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005). https://doi.org/10.1007/11560548_14
21. Lamport, L.: Byzantizing Paxos by refinement. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 211–224. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24100-0_22
22. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 190–202. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_12
23. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Comm. ACM **58**(4), 66–73 (2015)
24. Tendermint Inc.: Cosmos hub (2020). https://hub.cosmos.network
25. Tendermint core, reference implementation in Go (2020). https://github.com/tendermint/tendermint
26. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In: PODC, pp. 347–356 (2019)