Verification of HotStuff BFT Consensus Protocol With TLA+/TLC in an Industrial Setting

Vladimir Kukharenko[1], *Kirill* Ziborov[2], *Rafael* Sadykov[2,3], and *Ruslan* Rezin[3,*]

[1]Moscow Institute of Physics and Technology, 141701 Moscow, Russia
[2]Lomonosov Moscow State University, 119991 Moscow, Russia
[3]Innopolis University, 420500 Innopolis, Russia

Abstract. The extent of formal verification methods applied in industrial projects has always been limited. The proliferation of distributed ledger systems (DLS), also known as *blockchain*, is rapidly changing the situation. Since the main area of DLSs' application is the automation of financial transactions, the properties of predictability and reliability are critical for implementing such systems. The actual behavior of the DLS is largely determined by the chosen consensus protocol, which properties require strict specification and formal verification. Formal specification and verification of the consensus protocol is necessary but not sufficient. It is also required to ensure that the software implementation of the DLS nodes complies with this protocol. Finally, the verified software implementation of the protocol must run on a fairly reliable operating system. The financial focus of DLS application has also led to the emergence of the so-called *smart contracts*, which are an important part of the applied implementations of specific business processes based on DLSs. Therefore, the verifiability of smart contracts is also a critical requirement for industrial DLSs. In this paper, we describe an ongoing industrial project between a large Russian airline and three universities – Innopolis University (IU), Moscow Institute of Physics and Technology (MIPT) and Lomonosov Moscow State University (MSU). The main expected project result is a DLS for more flexible refueling of aircrafts, verified at least at the four technological levels described above. After brief project overview, we focus on our experience with the formal specification and verification of HotStuff, a leader-based fault-tolerant protocol that ensures reaching distributed consensus in the presence of *Byzantine* processes. The formal specification of the protocol is performed in the TLA+ language and then verified with a specialized TLC tool to verify models based on TLA+ specifications.

1 Introduction

The aim of this paper is to share our experience on applying formal specification and verification in the context of a large industrial project (hereinafter – *Project*). The

* Corresponding author: r.rezin@innopolis.ru

observations presented in this article can help other researchers in the formal methods to choose the right approach to the specification and verification of the developing systems. We begin our presentation of the results with a comparative analysis of consensus protocols for distributed ledger systems (DLS) to select one that would satisfy the Project requirements. The main task in this case is to formalize and automate the process of providing aircraft refueling services in order to control a timely payment and minimize the risks associated with the refusal of one of the parties to fulfill their obligations.

In a generalized form, the task of refueling an aircraft can be considered as the regulation of relations between two types of participants: consumers and service providers. Also, without loss of generality, it can be argued that each member of the relationship controls a DLS node and participates in changing the state of the DLS. The state of the DLS can be understood as a certain set of variables that characterize the processes of the real world. For example, it can be the amount of funds (balance) of the participants or the stage at which some aircraft is refueled. The DLS state is stored by each node and modified by special commands – *transactions* – that were initiated by the users of the network. A DLS' state cannot be changed in an arbitrary way, which is controlled by the consensus protocol. Without limiting the generality, we can say that the consensus protocol is a set of rules for the message exchange of a special kind between DLS nodes in order to agree on the list of transactions to execute and transit to a new state. However, the description of a DLS depends on the specifics of the controlled process. The concept of *smart contract* is introduced to avoid developing a separate system for each task. A *smart contract* is understood as a program written in a special programming language that describes how does the state of a single process change in the real world. The smart contract code is also stored as a DLS state. For example, it seems reasonable to have different smart contracts to manage aircraft refueling and fuel storage services. Therefore, the smart contracts allow simplifying the development of highly reliable systems for potentially unrelated tasks in the real world using, nevertheless, a common network infrastructure and a common implementation of the basic communication protocols between the network nodes. Based on this model, individual transactions contain calls to certain functions of smart contracts to move to a new DLS state from the current one. Almost in all existing DLS systems, the smart contract code cannot be changed after its deployment. This guarantees to each participant that none of the parties can withdraw from the obligations, even the developer of the smart contract.

It is essential that each participant of the relationship prioritize his own benefits – as much as the consensus protocol allows. The class of consensus protocols that govern this kind of relationship is commonly referred to as Byzantine Fault Tolerant (BFT). Another class of protocols used in practice are protocols resistant to the failure of a certain nodes set (Crash Fault Tolerant, CFT). Raft [1] is an example of a CFT protocol. CFT protocols are more efficient than BFT protocols, but they do not cope with malicious behavior of nodes, and therefore are not applicable within the framework of the problem under consideration.

Modern DLSs are also subdivided into open and closed ones. Open DLSs do not impose restrictions on the ability to connect an arbitrary node to the network, while closed DLSs have such restrictions. As a rule, the nodes in closed DLSs are either known to each other or fixed during network deployment, or special transactions are supported that allow existing nodes to add new ones. As you might guess, the open DLSs are much more complex than closed ones due to a more general task that they solve. For example, in the open DLS, there is a problem associated with the uncontrolled generation of "useless" transactions by nodes that are difficult to identify in the real world and cannot be disconnected from the DLS. This attack can ultimately lead to denial of service for the correct nodes. In contrast, in closed DLSs, the correct nodes can exclude a node from the network in the event of malicious behavior being exposed. Moreover, a participant in a

closed DLS can be contracted in the real world before another participant initiates a transaction to add a new node to the network. Thus, the main requirement for the consensus protocol in the closed DLSs is to detect a malicious behavior of a node while maintaining the correct state of the system. It is essential that the speed of operation in closed DLSs strongly depends on the number of nodes (at best, the dependence is linear), but the finality of the accepted state is guaranteed. In open DLSs, there is no such dependence on the number of nodes, but at each moment of time the finality of the state can be calculated only with a certain probability. Examples of open DLSs include Bitcoin [2], Ethereum [3] and EOS [4], closed ones include Hyperledger Iroha [5] and Fabric [6]. It is important that the modern consensus protocols for open DLSs contain a part that implements the consensus protocol for a fixed number of participants, which is also applicable in closed DLSs. Thus, on the basis of a closed DLS, it is possible to develop an open DLS.

Returning to the aircraft refueling problem, briefly described in the beginning of the section, it is worth noting that the participants of the refueling network are companies certified to provide services related to fuel, as well as aircraft owners. A limited set of nodes can be distributed between them to access the fueling network, since such companies may not be ready to spend resources on maintaining a dedicated node. On the other hand, the number of nodes is important for a network safety. For example, a part of the DPOS (Delegated Proof of Stake) protocol of the DLS EOS [4] consensus, which implements the BFT consensus protocol for a fixed number of participants, requires about 21 nodes, assuming that no more than 6 of them can be controlled by attackers. It is also important that it becomes necessary to select a certain privileged node set that will control the process of adding or removing new nodes, which automatically makes the DLS closed. It can be concluded that it is sufficient to choose a suitable consensus protocol for a closed DLS. However, this does not mean that a closed DLS cannot evolve into an open one: as noted earlier, the new state of an open DLS can be determined by a fixed number of participants, but this set should be periodically re-elected from the total number of participants with some additional logic based on the past states of the DLS.

In this article, we:

- Review the most popular BFT consensus protocols and the relationships between them (section 2);
- Provide the rationale for the choice of the HotStuff consensus protocol, as well as TLA+/TLC technologies for its verification (section 3).

Most importantly, we assess our experience with specification and verification of the HotStuff protocol in section 4.

2 Byzantine Fault Tolerance

By *consensus protocol* we mean a distributed algorithm for synchronizing the DLS state between nodes. In this paper, we will consider BFT consensus protocols, which assume the presence of f nodes from $n \geq 3f + 1$ with arbitrary malicious behavior. Nodes that follow the consensus protocol will be called *correct*. Each attempt to transition from one DLS state to another will be called *round*. As mentioned in the previous section, the state of the DLS changes as a result of the transaction execution sent by users. Thus, within a round of consensus protocol, nodes need to mutually agree on a single list of transactions to execute. The consensus protocol rounds are divided into phases for convenience, in each of them the nodes exchange messages for some purpose. Further, we say that the consensus protocol is *partially synchronous* if it is assumed that the message delivery time does not exceed the predetermined value of Δ after some network stabilization time (Global Stabilization Time, GST). Otherwise, we will call the consensus protocol *asynchronous*. In this paper, all considered protocols, with the exception of HoneyBadgerBFT, are partially synchronous.

The complexity of consensus protocols is tied to the number of operations to create or verify digital signatures used in the message exchange, since this operation often takes even longer than transferring data over the network.

Finally, the correctness of the consensus protocol is determined by two classes of properties: *safety* and *liveness*. The safety properties guarantee that for any moment in time t there is a moment in time $t' \geq t$ such that all correct nodes will have the same state, which will be the result of executing a list of transactions agreed within the framework of the consensus protocol. The liveness properties guarantee that for any moment of time t there is such a moment in time $t' \geq t$ that the state of the network will change in the presence of correct transactions from the user.

Further analysis of the most popular BFT consensus protocols is based on the evolution of one of the first protocols implemented in practice – PBFT (Practical Byzantine Fault Tolerant). Thus, the solutions to the problems that faced the scientific community after the application of the next improvements within the framework of the consensus protocol that were relevant at that time are demonstrated. We also look at the most current asynchronous consensus protocol – HoneyBadgerBFT – and compare it with partially synchronous protocols.

2.1 PBFT

Practical Byzantine Fault Tolerance (PBFT) [7] belongs to the class of partially synchronous BFT consensus protocols. This protocol defines the behavior of three entities: user, node and leader. Suppose there is a numbered node set $N = \{1, ..., n\}$, where n ($n \geq 3f + 1$) is the number of network nodes, and f is the maximum number malicious nodes. Then the leader for the r round is determined by the rule: $l = r \pmod n$. A user is an arbitrary device that can initiate transactions in the DLS using the node interface but may not participate in the consensus protocol.

The following user action algorithm is proposed:

- A signed transaction $\langle REQUEST, o, t, c\rangle_{\sigma c}$ is formed, where o is the operation data, t is the timestamp, c – user ID, σ_c – user signature.
- Based on the tracked round number, the node that is currently the leader is calculated. Further, it is to this node that the transaction is sent.
- $f + 1$ messages are expected from nodes of the form $\langle REPLY, r, t, c, i, rs\rangle_{\sigma i}$, where r is the current round number, r – transaction execution result.
- If it was not possible to get $f + 1$ results in a reasonable time, then send a first step request to all nodes.

The consensus protocol round consists of the following phases: *pre-prepare*, *prepare*, *commit*.

Without loss of generality, we assume that within a round, nodes form one user transaction for execution. A successful round of consensus protocol looks like this:

- The Leader forms and sends to the other nodes a message of the form:
- $\langle\langle PRE\_PREPARE, r, s, d\rangle_{\sigma l}, m\rangle$, where r is the number of the current round, s is the sequence number assigned to the user's transaction, m is the user's transaction, d is the hash value of the transaction.
- Other nodes, having received the *PRE_PREPARE* message, check its correctness. If the node did not receive a message with another d for the current values of r and n, then it goes into the *prepare* phase and sends to all nodes the message: $\langle PREPARE, r, s, d, i\rangle_{\sigma i}$

- On receiving the *PREPARE* message, the node checks if the values r, n and d match the *PRE_PREPARE* message. After receiving $2f + 1$ correct *PREPARE* messages from different nodes, the leader sends the message $\langle COMMIT, r, s, d, i\rangle_{\sigma i}$
- After receiving $2f + 1$ correct *COMMIT* messages from different nodes – provided that transactions with a lower sequence number are executed – the command contained in the m transaction is executed, and the DLS state on the i node changes.
- After executing the transaction, the i node sends the result to the user in a *REPLY* message, as noted earlier.

Along with the change of leader, which will be described below, the protocol guarantees the following important property: transaction m with sequence number s, executed by one node, will definitely be executed with the same sequence number by $f + 1$ correct nodes – possibly, after several leader changes.

In each phase, the node starts a timer at the Δ interval noted in the partial sync definition. This interval increases linearly if the timer is triggered and decreases if the phase ends earlier. If the timer is triggered, the node initiates the process of changing the leader.

Also, as an optimization, every k rounds each node starts the process of maintaining a stable state, which is used to reduce the amount of stored information and to synchronize the "lagging" nodes. The sequence number of the transaction corresponding to the last stable state is sent to the rest of the nodes in the message $\langle CHECKPOINT, s, d, i\rangle_{\sigma i}$ and is remembered locally after receiving $2f + 1$ of the correct message.

The process of changing the leader is initiated by the node by sending a message of the form $\langle VIEW\_CHANGE, r + 1, s, C, P, i\rangle_{\sigma i}$, where s is a sequence number of the last stable state (coincides with the serial number of the corresponding transaction), C is a set of $2f + 1$ signatures proving the correctness of the corresponding s state, P is a set consisting of sets P_m, corresponding to the transaction m, for which the node received $2f + 1$ *PREPARED* messages, and having a sequence number greater than s.

After the leader receives $2f + 1$ correct *VIEW_CHANGE* messages, the local stable state is updated to a new one with the maximum sequence number. Next, the leader forms and sends the message $\langle NEW\_VIEW, r + 1, V, O\rangle_{\sigma l}$, where: V is a set containing $2f + 1$ received by the leader messages *VIEW_CHANGE*; O is a set containing *PRE_PREPARE* messages for each transaction from P with a sequence number that exceeds the number corresponding to the current stable state. If the set of transactions that satisfy this condition is empty, then the special value d_{null} is used as a hash value, indicating that the leader has changed, but there are no commands to execute. Each node, in turn, checks the correctness of constructing the set O and other data when receiving *NEW_VIEW* and starts the described earlier two-phase process for each *PRE_PREPARE* message.

It is easy to see that with a favorable outcome of the round, the consensus reaching algorithm has a complexity of $O(n^2)$ due to the fact that at each of the n nodes each phase requires verification of $O(n)$ signatures. In case of a leader change, it is necessary to repeat the phases for about n transactions (since the number f of malicious nodes linearly depends on the total number of nodes), which leads to a complexity of about $O(n^3)$.

2.2 HoneyBadgerBFT

The BFT consensus protocol HoneyBadgerBFT [8] belongs to the *asynchronous* class, which means that there are no assumptions about DLS latency compared to the protocol discussed in the previous section. The main idea of the protocol is to use a distributed reliable delivery algorithm. Thus, the nodes do not need to use a timer to change state after a certain period of time, since a message sent to the network will sooner or later be delivered.

Let us assume there is a numbered node set $N = \{1, ..., n\}$, where n ($n \geq 3f + 1$) is the number of network nodes, f is the maximum number of malicious nodes. Before the network starts working, the keys necessary for the operation of threshold encryption [**Ошибка! Источник ссылки не найден.**] are generated: the public key PK and n private keys SK_i for each node. Threshold encryption, as shown below, is used to prevent censorship of transactions. It is assumed that each node has an unbounded buffer that accumulates user transactions.

An abstract description of the consensus protocol is as follows:

- When the r round begins, the i node randomly selects $\lfloor b / n \rfloor$ transactions from the first b currently available in the buffer. Let us call these transactions *proposed*.
- The selected set of transactions is encrypted with the public threshold signature key: $x = \mathrm{TPKE.Enc}(\mathrm{PK}, \mathrm{proposed})$.
- x is passed as input to the consensus protocol for round r ($ACS[r]$), the task of which is to agree on a common subset of the encrypted sets of transactions proposed by the nodes.
- After receiving the $\{v_j\}_{j \in S}$, $S \subset N$ sets, the $ACS[r]$ decryption process is initiated for the r round.
- For each $j \in S$ $e_j = \mathrm{TPKE.DecShare}(\mathrm{SK_i}, v_j)$ is calculated.
- A message of the form $\mathrm{DEC}(r, j, i, e_j)$ is sent.
- Upon receiving $f + 1$ messages $\mathrm{DEC}(r, j, k, e_{k,j})$, the set of transactions $y_j = \mathrm{TPKE.Dec}(\mathrm{PK}, (k, e_{k,j}))$.
- The total set of transactions for the r round is calculated as: $\mathrm{block}_r = \mathrm{sorted}(\mathrm{U}_{j \in S} y_{i)}$.
- Processed transactions are removed from the buffer.

The consensus protocol to obtain a common subset of transactions for node i uses the Reliable Data Transfer (RBC) [10] Binary Consensus (BA) [11] and has the following form:

- When a node receives the value v_i, it is sent via RBC.
- When the value v_j is received, it is passed to the BA binary negotiation protocol to determine the inclusion of this set of transactions in the final set.
- Once the $N - f$ matching results are obtained for different v_j, the rest of the results get 0 as their value.
- After the delivery of all v_j, the set $\{v_j\}$ is formed with elements for which BA returned 1.

The author of the original work established that for $b = \Omega(\lambda n^2 log(n))$ the consensus protocol in the worst case has complexity $O(b)$. It can also be noted that the protocol is most effective in conditions of an excess of transactions, since in this case there is a high probability that each node will select different sets of transactions, which increases the network bandwidth.

2.3 SBFT

The Scalable Byzantine Fault Tolerance (SBFT) [12] consensus protocol was developed to optimize PBFT. For this reason, in this section, we do not dwell on the algorithm in detail, but consider only the changes proposed by the authors of the protocol regarding PBFT.

First, it is worth noting that in each round of the PBFT protocol, each node verifies the signature of messages from other nodes, which has a complexity of $O(n^2)$, where n is the number of nodes in the network. For this reason, the SBFT authors proposed to introduce 2 additional node roles in each phase besides the leader: C-collector and E-collector. The task of the C-collector is to accumulate messages of a certain purpose from other nodes until

their number reaches the required threshold, and also to send one resulting message to the nodes, which contains a single signature consistent with the signatures from the received messages. Thus, the C-collector and the rest of the nodes together check only $O(n)$ messages if the round is successful. Combining several signatures into one is achieved by using the so-called threshold signatures [13]. The main disadvantage of this approach is that before starting the network, the nodes need to jointly generate a public key, which depends on the collection of private keys of each node. By analogy with the C-collector, the task of the E-collector is to collect messages from nodes with the result of the transaction execution, combine the signatures and send one message to the user.
Thus, the user only needs to receive a single correct message from the E-collector in order to make sure that the transaction was successfully executed in the DLS, instead of waiting for the $f + 1$ message from the nodes, as in PBFT.

The second important optimization is to add a "optimistic path" for the round. It consists in the fact that if the C-collector receives *PREPARED* messages from all n nodes before the timer expires, then it immediately generates a *FULL_COMMIT_PROOF* message and sends it to the nodes. If the message is correct, the node immediately adds the transaction to the queue for execution, bypassing the additional *precommit* phase of the PBFT protocol. If the fast path timer expires, the protocol proceeds in two phases (like PBFT, but taking into account the optimization of signatures), with the C-collector becoming the leader.

Further, the authors of SBFT propose to use not one collector, but c C-collectors, where c is much less than n. This increases the likelihood of the round going through the fast track even if some collectors are unavailable. As a result, it turns out that the number of nodes in the network is $n = 3f + c + 1$, where f is the number of malicious nodes, and c is the number of nodes that may be inaccessible, but cannot perform other malicious actions that are contrary to the consensus protocol.

Taking into account the optimization of signatures, we can conclude that SBFT has complexity $O(n)$ in case of a positive round outcome and $O(n^2)$ in case of a leader change.

2.4 Tendermint

The Tendermint [14] consensus protocol is built on the same phases as the PBFT, but in each round of the consensus protocol, the nodes agree not on a single transaction, but on a block consisting of a set of transactions. The authors of Tendermint propose optimization that avoids the complexity of repeating consensus protocol phases for each unsuccessful block after a leader change. This is achieved through a specially designed Tendermint Gossip protocol and a state machine that stores only the last block to repeat the consensus protocol in a new round in the event of a leader change. Unlike SBFT, Tendermint optimizes the second of two bottlenecks in the PBFT protocol: sending a message from each node to everyone else and repeating the consensus protocol phase for each unsuccessful transaction upon receiving the required number of *PREPARE* messages from the DLS nodes.

The Tendermint messaging protocol uses the assumption of partial synchrony of the network and ensures that if the correct node sends a message at time t, then the other correct nodes will receive it no later than $max\{t, \text{GST}\} + \Delta$, where *GST* is a certain moment in time when the network stabilizes, and Δ is the guaranteed message delivery interval in a stable network.

The authors of Tendermint suggest the names *proposal*, *prevote*, *precommit* instead of PBFT phases' names *pre-prepare*, *prepare*, *commit* to better understand the tasks of each phase; between the phases, however, a one-to-one correspondence can be made. Also, the Tendermint state machine means storing the following values to optimize the process of

changing the leader: *validValue*, *validRound*, *lockedValue*, *lockedRound*. The validValue and validRound values store the value of the last transactions' block for which $2f + 1$ *prevote* messages were received, as well as the corresponding round number. These values are reset after $2f + 1$ *PRE_COMMIT* messages are received, that is, after the block has been negotiated within the framework of the consensus protocol, but until then the correct leader sends these values in his *proposal* message. The lockedValue and lockedRound values are updated exclusively in the precommit phase, before sending the *PRE_COMMIT* message, and are required to track the fact that the *PREVOTE* message will not be sent to *PROPOSAL* which transaction block is different from lockedValue.

Thus, the Tendermint message distribution protocol, in conjunction with the state machine, guarantees that the lockedValue transaction block will sooner or later be accepted by the network, since after a finite number of rounds there will be a correct leader who will send *PROPOSAL* with validValue and validRound consistent with lockedValue and lockedRound. This message will be received by the remaining valid nodes due to the guarantees of the data transfer protocol.

As a result, the complexity of the Tendermint consensus protocol is estimated as $O(n^2)$ in the general case, where $n \geq 3f + 1$ is the number of network nodes, f is the maximum number of malicious nodes. It is worth noting, that this complexity can be improved to $O(n)$ using the SBFT threshold signature approach.

2.5 HotStuff

The Tendermint consensus protocol lacks the Optimistic responsiveness property due to the synchronous delay caused by the need to guarantee at least some kind of response: after reaching *GST*, the chosen leader only needs to receive $2f + 1$ of leader change messages, containing *validValue* to generate a *PROPOSAL* message with a block of transactions, which sooner or later will be accepted by the network. The following scenario demonstrates the absence of the described problem in the absence of a synchronous delay between rounds. Let us say one of the nodes receives $2f + 1$ *prevote* messages and updates both *validValue* and *lockedValue* and becomes locked while the other nodes change leaders. The new leader will not know the current *validValue* and will offer another block of transactions that will be rejected the blocked node. Moreover, in this new round, the situation may repeat itself, and another node will be blocked.

The described problem occurs due to the fact that the variables *validValue* and *lockedValue* are updated in the same phase of the round. The authors of the HotStuff [15] protocol suggested adding another phase to solve this problem and updating these variables in different phases. Thus, if a node is locked (the *lockedValue* variable is set), then this also means that at least $2f + 1$ nodes have set the validValue variable, and the correct leader will include these transactions in the proposed for the new round. Thus, in HotStuff, nodes vote in the *prepare*, *pre-commit*, *commit* phases and apply the result in the *decide* phase. Also, *prepareQC* and *lockedQC* are used instead of *validValue* and *lockedValue*. In addition, HotStuff assumes the use of the threshold cryptography mentioned more than once, which guarantees the linear complexity of the work. The leader of the round is responsible for collecting signatures and generating a single signature.

A successful round of HotStuff consensus protocol is described as follows:

- The Leader forms and sends to the rest of the nodes a message of the $\langle PREPARE, curProposal, highQC\rangle_{\sigma l}$, where *curProposal* is a block of transactions offered for agreement in the current round, *highQC* – proof that the transaction block matches the most current *prepareQC* on the network.

- Other nodes, having received the *PREPARE* message, check its correctness. If the host variable *lockedValue* is not set or matches *curProposal* and *highQC*, then the host sends the message $\langle VOTE\_PREPARE, curProposal\rangle_{\sigma i}$.
- Having received $2f + 1$ messages *VOTE_PREPARE*, the leader generates a threshold signature, updates *prepareQC* based on *curProposal* and node votes, sends a message $\langle PRE\_COMMIT, prepareQC\rangle_{\sigma i}$.
- The other nodes, having received the *PRE_COMMIT* message, check its correctness and signature. If successful, the node sends the message $\langle VOTE\_PRE\_COMMIT, prepareQC\rangle_{\sigma i}$ and update *prepareQC*.
- Having received $2f + 1$ messages *VOTE_PRE_COMMIT*, the leader generates a threshold signature, sets *lockedQC* equal to *prepareQC* and sends the message $\langle COMMIT, prepareQC\rangle_{\sigma i}$.
- The other nodes, having received the *COMMIT* message, check its correctness and signature. If successful, the node sends the message $\langle VOTE\_COMMIT, prepareQC\rangle_{\sigma i}$ and updates *lockedQC*.
- Having received $2f + 1$ of *COMMIT* messages, the leader forms a threshold signature and sends the message $\langle DECIDE, lockedQC\rangle_{\sigma l}$. In addition, the corresponding *lockedQC* block is queued for execution.
- Other nodes, having received the *DECIDE* message, check its correctness and signature. If successful, the node adds the corresponding block to the execution queue.

Further, the authors of HotStuff noticed that the nodes that received *PREPARE* messages can optimistically assume that the proposed block will be successfully accepted, and simultaneously initiate the next round. Therefore, the blocks waiting for acceptance become dependent on each other and can be represented as a list. Each subsequent block contains the node signatures for the previous block. Thus, if after this block three more blocks are added to the list, then this block can be considered accepted, since this event is equivalent to the passage of three phases of the HotStuff protocol. This algorithm is called Chained HotStuff.

Table 1. Comparison of partially synchronous BFT consensus protocols

Protocol	Correct Leader	Change of Leader	Optimistic responsiveness
PBFT [7]	$O(n^2)$	$O(n^3)$	Yes
SBFT [12]	$O(n)$	$O(n^2)$	Yes
Tendermint [14]	$O(n^2)$	$O(n^2)$	No
HotStuff [15]	$O(n)$	$O(n)$	Yes

Table 1 provides a final comparison of the most popular semi-synchronous BFT consensus protocols used in practice. Algorithmic complexity is measured in the number of digital signature checks per round depending on the number of nodes in the network $n \geq 3f + 1$, where f is the maximum possible number of malicious nodes. As shown in the previous sections, the PBFT trailblazer has a fairly high algorithmic complexity and complex leader change logic. The SBFT authors proposed threshold signature optimization and *optimistic path* optimization for stable networks, but the complexity of the leader change logic remains the same. Further, the authors of Tendermint proposed a significant simplification of the logic of changing the leader, but at the same time they sacrifice the *optimistic responsiveness* property and add a dependence on the data transfer protocol (Tendermint gossip) for correct operation. Finally, the HotStuff protocol offers a solution to the problems of the Tendermint protocol by introducing an additional phase but retaining linear complexity.

Separately, it should be noted that asynchronous protocols currently have a higher algorithmic complexity ($\Omega(\lambda n^2 log(n))$ – HoneyBadgerBFT) and are more difficult to implement.

3 Distributed Consensus Verification

The comparison of consensus protocols in Section 2 shows that HotStuff is the best choice as a consensus protocol for implementing a closed distributed ledger system in terms of complexity and optimistic responsiveness. It is worth noting, that due to the need to use the GOST 34.10 cryptographic standard and the absence of a solution for threshold signatures in it, the HotStuff implementation is planned without them. On the one hand, this will increase the algorithmic complexity up to $O(n^2)$, on the other hand, it will simplify the formal verification of the protocol implementation. Further, we note that all the considered BFT consensus protocols with linear complexity assume the use of threshold signatures, but either do not have the optimistic responsiveness property (like Tendermint), or have a higher complexity for the leader change phase (PBFT, SBFT) in comparison with Hotstuff.

In addition to the complexity of the consensus protocol and the time of its implementation, it is necessary to assess its formal verifiability: how difficult is to prove that the program code of the algorithm has the properties required by its formal description – the specification. As an example, we can consider the repeatedly mentioned safety property. Then one of the tasks of formal verification of the consensus protocol is a mathematical proof that a distributed system will never find itself in a situation where two different correct network nodes will have conflicting states (for example, one node will report that the aircraft refueling has finished successfully, and the other – which is not successful). Today, there are two approaches to formal verification of consensus protocols: verification by model checking (model checking [16] and verification by deductive methods [17] [18] [19]).

The model checking method is applied not to the source code of the protocol, which is used for compilation and execution, but to the abstract model of the protocol in the form of a Kripke structure. Each state of the machine corresponds to a set of protocol properties that are valid in this state. The proof that the constructed model has a given property is carried out by representing this property as a formula in one of the temporal logics – for example, LTL (linear time logic) or CTL (branching time logic). For each logic, there is a separate algorithm that checks the property for a given Kripke structure. The result of this algorithm is either a message that the property is being executed, or the return of a sequence of transitions (a counterexample) in the automaton that demonstrates a violation of the property. The advantage of the model validation approach is that the Kripke structure is more formal notation of the consensus protocol, rather than some pseudocode that is used in scientific articles and specifications to explain the idea of the protocol, which allows finding flaws either at the time of model formation or while studying a counterexample after completing the model validation process. One of the disadvantages of this approach is that often the models of consensus protocols in the initial record have a huge number of states, which does not allow the verification algorithm to be completed in an acceptable time, which results in the problem of simplifying the original model. Another disadvantage of the method is that it is not the source code of the protocol that is checked, but its abstract model, as a result of which the problem arises of proving the correspondence of the checked model to its software implementation.

On the contrary, the deductive verification methods are most often applied to software implementation but require the development of the operational semantics of the implementation language in the chosen verification system. Operational semantics is

required to obtain information on how this or that syntactic construction of a programming language changes the program state. The deductive approach to verification uses the idea of traditional proof formation by mathematical methods, but with the ability to automate template techniques – tactics (for example, the method of mathematical induction), which are an integral part of the *proofs* (proof assistant). Thus, formal verification by the deductive method implies writing a program for the consensus protocol in a certain programming language and transferring this code to a prover, which already describes the operational semantics for this language, followed by an attempt to automatically prove it using tactics built into the prover. The main disadvantage of this approach in comparison with model checking is that the process does not imply an automatic finding of a counterexample – that is, either it is possible to prove the desired property for the program, or it does not work; however, a failed attempt does not mean that such proof does not exist.

At the time of writing this paper, we managed to find the results of Tendermint and PBFT protocols verification. In the Tendermint verification work, the authors use the model verification method to prove that the constructed protocol model in the TLC system has a safety property. On the contrary, for PBFT verification, the deductive verification method in the Coq system was used. Moreover, a special framework has been developed, Velisarius [20], which can be used to verify an arbitrary BFT consensus protocol. Velisarius is a set of theories for the Coq [19] prover that incorporates basic abstractions of consensus protocols: for example, the *EventOrdering* theory for describing events occurring at different nodes in the network, or the *Process* theory, in which state machines of nodes are defined. With the help of Velisarius, the safety property of PBFT was proved, and from the obtained specifications, the executable code of the protocol in the OCaml language was generated. Despite the results of formal verification of Tendermint and PBFT, we decided to use Hotstuff as the main protocol of the distributed ledger system implemented within the Project, due to the presence of the optimistic responsiveness property and the simplified implementation of the leader change phase. It is worth noting that the similarity between Tendermint and Hotstuff allowed us to apply the ideas used to validate the Tendermint protocol model to verify the Hotstuff protocol model. That is why we decided to discuss verification of Tendermint in more detail.

3.1 Verification of Tendermint

The authors of [21] focus on the safety aspect and introduce the following assumptions for simplifying the model:

- The modelling only happens at level 1: several rounds of the protocol must result in exactly one decision.
- All time-outs are non-deterministic.
- The leader-defining function is non-deterministic and does not guarantee any fairness properties.
- All messages from the Byzantine processes are added to the set of the messages in the very beginning.
- Every process has the voting weight 1.
- Identity is used as the hash function.

Under all these assumptions, the model corresponds to the original description of the consensus algorithm [14]. The authors of [21], however, consider possible error scenarios besides only checking correctness of the algorithm. That is, they verify that violating the safety property can only result from one of the following conditions:

- Amnesia. There exists a set, we call it *Detectable*, of Byzantine processes s.t. $|Detectable| \geq (F + 1)$. Each process from *Detectable* had sent *PRE_VOTE* and

PRE_COMMIT messages for command v_1, and in subsequent rounds it sent a *PRE_COMMIT* message for v_2 when there were not enough *PRE_VOTE* messages.

- Equivocation. There exists a set, we call it *Detectable*, of Byzantine processes s.t. $|Detectable| \geq (F + 1)$. For each process from *Detectable* there exists a round in which the process sent two messages that are equal up to the command (i.e., all their fields are identical, except the *node* field).

Also, they have specified and model-checked invariant properties necessary for satisfying the safety property. The authors of [21] have used Apalache – an experimental tool. This model checker works faster than TLC, which is why it was possible to execute many runs with the number of processes (N) from 4 to 10. This result is very important: Tendermint has a lot in common with HotStuff, so we reused the Tendermint's specification structure and some ideas for specification and verification.

4 Specifying and Verifying HotStuff With TLA+/TLC

Since the HotStuff and Tendermint protocols are very similar, it was decided to implement the first version of the model based on the work of Igor Konnov [21]. Since he focused on the safety property, that is where the work began. All the following reasoning, as well as checking the model as a whole, is based on the fact that we trust the representation of the real model of behavior in TLA+. We describe in detail the assumptions and modifications we made and justify the existence of such assumptions.

4.1 Structural assumptions of the main model

We have prototyped a model based on the previous results. The list below captures its main features. We developed these features with the safety property in mind.

- The protocol only works with a fixed number of rounds.
- All timeouts are non-deterministic.
- The leader-defining function initializes right before the algorithm begins its execution.
- Messages from Byzantine processes get added to the set of messages as new quorum certificates (QC) emerge, from all the processes and to all the phases at once.
- Each process has voting power 1.
- Almost everywhere we only use the necessary part of the messages (the one directly used by the protocol).
- We removed the leader role from the *PRE_COMMIT*, *COMMIT* and *DECIDE* phases – the leader's role had been limited to collecting threshold signatures proving reception of messages).
- *STOP* is a dummy phase that we introduced to terminate the consensus protocol.
- *PREPARE* is a dummy phase that we introduced to set up the order of executing the *PREPARE* phase by the leader.

We borrowed the main structure from the work on verifying safety of the Tendermint protocol [21].

4.2 Leaf structure of the commands

The decisions described in section 4.1 formed the basis for elaborating the model. In particular, introduction of the leaf structure of the commands was an important addition to the model. Branching of the proposed commands is an important part of the algorithm itself

[15]. The structure is called “leaf” (not “tree”) because sequences of commands are represented as leafs for individual processes, though they are actually paths in the tree of commands.

To implement this structure, we initialize a binary tree during the model’s creation phase. This lets us avoid dynamic creation of new commands and thus of too many new states, and simplifies specification of the *ExtendsFrom* predicate. We also omit checking correctness of the proposed commands – this procedure is very specific to particular implementations of the protocol.

All commands, represented as nodes, are offered from the binary tree. The *Validity* property checks that all the nodes accepted by the process are continuations of each other. The *Agreement* property checks that all processes only accept nodes from the same branch.

4.3 Verification without the leaf structure

We ran the verification procedure several times on the serve, and the last run – checking the model with the leaf structure – was the most informative one. Verification runs of the basic model, however, deserves analysis as well. They terminated very quickly, which we see as their biggest value. The significant difference in execution times between the two kinds of runs is due to the larger number of states added by the leaf structure. This happens basically because another degree of freedom is added – a transition happens from 2 types of messages to $2^{MaxView+1}$ types. Also, we were able to debug the basic model exactly because of the early verification runs.

The largest execution time of the basic model was observed with 4 processes, 1 of which was Byzantine – around 7.5 hours with default TLC settings. Appendix A contains more details, together with the number of generated states. The remaining runs of the basic model were executed with the objective of getting an error. For example, we ran the model with 7 processes, 3 of which were Byzantine. The model checker worked for around 30 minutes and produced a counterexample leading to an error. We think this execution time to be a good result for a model of the HotStuff protocol. We expect to extend the basic model in an alternative way in the future to check other properties. Also, it is possible to use the existing model to get the algorithm of behavior of Byzantine processes in the event of violating the limitation on their amount ($3 * F < N$).

4.4 Verification with the leaf structure

We had only one verification run of the model with the leaf structure. Appendix A contains the details. First of all, we would like to note that the number of states increased around 10 times. The number of states directly affects our level of trust to the results produced by TLC. Because TLC hashes states, the probability of hashes’ collision grows with the number of states. Running the model several times, assigning another value to the *seed* parameter each time, may partially solve the problem. We think the execution time of 17 days to be acceptable because of the high complexity of the algorithm under verification and because of the presence of Byzantine processes. This execution time makes it possible to get results with small values of N. As with the basic model, it is still possible to run the model with the violated correctness conditions with the purpose of figuring out the Byzantine processes’ behavior.

5 Conclusions

In this article, we:

- Gave a brief description of an industrial project for building a distributed ledger, a mandatory part of which is formal verification of all the main components of the system being developed.
- Analyzed the popular distributed consensus protocols applicable to our task.
- Reviewed the methods used in practice for formal verification of such protocols.
- Explained choosing the HotStuff protocol and the model checking method for its verification.
- Reflected on our experience of verifying HotStuff with the TLA+/TLC tool chain.

We hope that our analysis will save time and effort for other researchers pursuing similar projects. In addition to applying model checking, it is also necessary to use deductive methods to verify the source code of the protocol – for example, using the Velisarius framework. The Project's plan includes such verification as its critical phase.

Appendix A

TLC Output

All model verification runs were performed with the default settings.

The basic model

The following output was produced by the verification tool for the basic model without the leaf structure [22]:

language=bash] 575075776 states generated, 79156964 distinct states found, 0 states left on queue. The depth of the complete state graph search is 48. The average outdegree of the complete state graph is 1 (minimum is 0, the maximum 12 and the 95th percentile is 4).

From which we conclude that:

- the model checker terminated with no errors;
- it took 7 hours and 23 minutes for the model checker to terminate;
- the model checker generated 575075776 states;
- out of the generated states, 79156964 were unique;
- the depth of the search in the state was 48;
- the average outgoing degree of the nodes in the graph was equal to 1.

The model with the leaf structure

The following output was produced by the verification tool for the basic model with the leaf structure [22]:

language=bash] Model checking completed. No error has been found. Estimates of the probability that TLC did not check all reachable states because two distinct states had the same fingerprint: calculated (optimistic): val = .30 based on the actual fingerprints val = .25 7338602054 states generated, 865144941 distinct states found, 0 states left on queue. The depth of the complete state graph search is 51. The average outdegree of the complete state graph is 1 (minimum is 0, the maximum 12 and the 95th percentile is 4). Finished in 17d 08h at (2020-09-05 02:30:38)

From which we conclude that:

- the model checker terminated with no errors;
- it took 17 days and 8 hours for the model checker to terminate;
- the model checker generated 7338602054 states;

- out of the generated states, only 865144941 were unique;
- the depth of the search in the state was equal to 51;
- the probability of collision according to the "optimistic" formula was equal to 0.30;
- the probability of collision according to the hashes being produced was equal to 0.25;
- the average outgoing degree of the nodes in the graph was equal to 1.

Acknowledgements

This research has been financially supported by the Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002).

References

1. M. Fazlali, S.M. Eftekhar, M.M. Dehshibi, H.T. Malazi, M. Nosrati, CoRR abs/1911.01231 (2019)
2. S. Nakamoto, Tech. rep., Manubot (2019)
3. G. Wood, Ethereum project yellow paper, **151**, 1 (2014)
4. E. Elrom, *EOS.IO Wallets and Smart Contracts*, 213 (2019)
5. F. Muratov, A. Lebedev, N. Iushkevich, B. Nasrulin, M. Takemiya, CoRR abs/1809.00554 (2018)
6. E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A.D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *Hyperledger fabric: a distributed operating system for permissioned blockchains*, in *EuroSys*, 30,1 (2018)
7. M. Castro, B. Liskov, *Practical Byzantine Fault Tolerance*, in *OSDI,* 173 (1999)
8. E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, S. Halevi, eds., Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria*,* (2016)
9. J. Baek, Y. Zheng, *Simple and efficient threshold cryptosystem from the Gap DiffieHellman group*, in *GLOBECOM* , 1491 (2003)
10. M. Ben-Or, B. Kelmer, T. Rabin, *Asynchronous Secure Computations with Optimal Resilience (Extended Abstract)*, in *PODC,* 183 (1994)
11. A. Mostéfaoui, M. Hamouma, M. Raynal, *Signature-free asynchronous byzantine consensus with t 2<n/3 and o(n^2) messages*, in *PODC*, 2 (2014)
12. G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M.K. Reiter, D. Seredinschi, O. Tamir, A. Tomescu, CoRR abs/1804.01626 (2018)
13. D. Boneh, B. Lynn, H. Shacham, J. Cryptol., **17**, 297 (2004)
14. E. Buchman, J. Kwon, Z. Milosevic, CoRR abs/1807.04938 (2018)
15. M. Yin, D. Malkhi, M.K. Reiter, G. Golan-Gueta, I. Abraham, *HotStuff: BFT Consensus with Linearity and Responsiveness*, in *PODC*, 347 (2019)
16. *Model Hecking,* (2010)
17. L.C. Paulson, *Isabelle - A Generic Theorem Prover* (with a contribution by T. Nipkow), 828 of Lecture Notes in Computer Science (1994)
18. T. Gauthier, C. Kaliszyk, J. Urban, *TacticToe: Learning to Reason with HOL4 Tactics*, in *LPAR*, 46 of EPiC Series in Computing, 125 (2017)

19. B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin., INRIA, 6 (1999)
20. V. Rahli, I. Vukotic, M. Völp, P.J.E. Veríssimo, *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*, in *ESOP*, 10801 of Lecture Notes in Computer Science, 619 (2018)
21. Igor Konnov, *Model Checking Tendermint* (2020) https://github.com/
22. Vladimir Kukharenko, *HotStuff TLA+ Specifications* (2020) https://github.com/