Formal Verification of HotStuff

Leander Jehl

▶ **To cite this version:**

Leander Jehl. Formal Verification of HotStuff. 41th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Jun 2021, Valletta, Malta. pp.197-204, 10.1007/978-3-030-78089-0_13 . hal-03740260

HAL Id: hal-03740260
https://inria.hal.science/hal-03740260

Submitted on 29 Jul 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

Formal Verification of HotStuff

Leander Jehl

University of Stavanger, Norway leander.jehl@uis.no

Abstract. HotStuff is a recent algorithm for repeated distributed consensus used in permissioned blockchains. We present a simplified version of the HotStuff algorithm and verify its safety using both Ivy and the TLA Proof Systems tools.
We show that HotStuff deviates from the traditional view-instance model used in other consensus algorithms and instead follows a novel tree model to solve this fundamental problem. We argue that the tree model results in more complex verification tasks than the traditional view-instance model. Our verification efforts provide initial evidence towards this claim.

1 Introduction

The advent of blockchain technology has significantly increased interest in Byzantine Fault Tolerant (BFT) systems. BFT systems tolerate arbitrary misbehavior of a fraction of participating nodes. Therefore, these systems build a key component for recent permissioned [2, 23] and federated [17] blockchain systems. However, BFT algorithms are notoriously difficult to design or implement and thus have been the subject of numerous efforts in formal methods [1, 13, 14, 22]

In this paper we verify safety properties of the HotStuff [23] algorithm using both the TLA Proof System (TLAPS) [5] and the recent Ivy tool [18]. The HotStuff algorithm is used in the Diem, formerly libra blockchain, and was, to the best of our knowledge, not previously formally verified. Drawing inspiration from blockchain technology, the HotStuff algorithm presents a novel paradigm in consensus algorithms, replacing views and instances with a tree model. We show that this new paradigm complicates the verification of safety properties compared to the more traditional view-instance model. We find that the need to ensure the tree structure and a reachability or ancestor predicate for the tree model in inductive invariants complicates verification. The different nature of the formal frameworks used allows us to address this issue in different ways. Additionally, our efforts led us to discover a simplified version of the HotStuff protocol, which restricts the use of reachability to the safety properties and removes it from the algorithm.

In addition to contributing to the formal verification and better understanding of the HotStuff algorithm and its novel paradigm, this work also presents a case study and comparison of the two verification systems, TLAPS and Ivy. Both of these aim to make formal verification available to practitioners and engineers and at least the TLA+ model checker is actively used in industry [19]. Thus, our comparison of the recent Ivy tool with more mature TLAPS forms an

important evaluation of that system. We are not aware of a previous use of the Ivy tool that did not involve the original authors.

2 View-instance and tree model for repeated consensus

This section presents the different paradigms used to implement repeated consensus. Repeated consensus is a fundamental problem in distributed computing. The problem requires processes to maintain an append-only log in the presence of faults. Repeated consensus allows to implement arbitrary objects in a fault tolerant manner, by maintaining a log of deterministic operations, applied to the object. This state machine approach [21] is widely deployed in the cloud, e.g. in Zookeeper [10], but also builds the basis for recent blockchain systems.

A common model for algorithms solving repeated consensus (e.g. [3, 16]) is, what we call the *view-instance model*. As shown in Figure 1, this model uses a matrix of possibly infinitely many slots (or fields), each indexed by two integers, instance and view. In each slot a value can be proposed by a leader and committed by the processes. We say that the slot is committed. Since fault tolerant consensus is impossible to solve in an asynchronous system [6], some slots may remain uncommitted, or even without a proposed value, e.g. due to leader failure or network partitions. Indeed, in some algorithms, such as FlexiblePaxos [9] or PBFT [3], slots have a preassigned leader that may fail. In such cases, values may be committed in additional slots, using higher views. The instances represent the entries in the distributed log. If a value is committed in one slot, it is adopted for that instance. Algorithms thus need to ensure, that values committed in different slots belonging to one instance are the same. This property forms the main safety condition for algorithm using the view-instance model.

The view-instance model has advantages both for algorithm design and performance. For algorithm design, the slots belonging to one instance can be viewed as solving a single instance of the consensus problem. Most prominently the Paxos algorithm has been presented in this way [16]. Similarly, the safety condition stated above can be applied to a single instance. Thus, in the formal verification of these algorithms, a common approach is to first validate the algorithm in a single instance [14, 15] and subsequently extend the model to cover multiple instances [4, 20]. Additionally, the view-instance model allows optimizations that apply operations across instances. For example a new view may be started simultaneously in all instances [16].

Tree model: HotStuff [23] introduces a new way to implement repeated consensus, that organizes slots in a rooted tree, as shown in Figure 2. We call this the *tree model*. If a slot is committed, the slot and its ancestors are used as the current prefix of the log. Thus, the depths of slots in the tree-model corresponds to the instance in the view-instance model. If a slot cannot be committed, the processes can try to commit one of its descendants instead. Alternatively, a new slot can be proposed at the same depth. To distinguish slots at the same depth, the tree model uses rounds, similar to views in the view-instance model. E.g. if

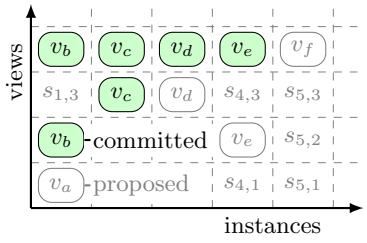

Fig. 1. View-instance matrix with proposed and committed slots.

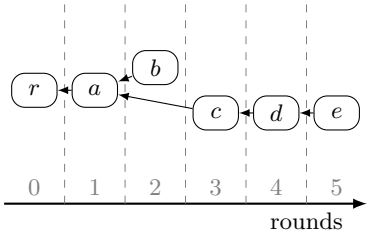

Fig. 2. Slots with parent relation in the tree model.

a slot was not proposed at a certain depth, a new slot can be proposed at the same depth, but with a larger round. Different from views, rounds do not restart for every depth, but instead the round of a slot is larger than the round of its parent. The core safety property in the tree model is that any two committed slots are on the same branch of the tree, or equivalent for two committed slots, one is the ancestor of the other.

The advantage of the tree model is that it is not necessary to ensure that different slots commit the same value. Thus, this model is suited to allow simple leader change procedures, since it is not necessary to ensure that the new leader proposes the same value as an old leader. Indeed, leader change with linear communication complexity is the main contribution of the HotStuff protocol. An additional advantage is that not every slot has to be actually committed, as long as a descendant of the slot is eventually committed. Similar to the view-instance model, the tree model alows specific optimizations. HotStuff spreads the process of committing a slot over the slot's descendants.

The verification of an algorithm in the tree model poses some novel challenges. First, the existing approach to first prove safety of a single instance and then extend this proof to multiple instance does not apply. Second, in the tree model, the safety property, and in some cases also the algorithm contain a reachability predicate. Reachability on finite graphs, i.e. the transitive closure of the neighbor relation, cannot be expressed in first order logic [12].

3 Simplified HotStuff algorithm

We now present our simplified version of the HotStuff algorithm and explain how it differs from HotStuff [23]. As common in BFT algorithms, HotStuff assumes $3f + 1$ processes, of which at most f may be faulty. Faulty processes may stop but also violate the protocol. However, faulty nodes cannot subvert cryptographic primitives. Especially, a digital signature scheme is used to authenticate messages, preventing impersonation of correct (non-faulty) processes.

The algorithm contains the following two operations. The leader of a round may propose new slots. If the followers accept the proposed slot they sign it. The leader collects these signatures into a certificate, containing $2f + 1$ signatures.

We say that a slot is *certified*, if there exists a certificate for that slot. The root slot has a certificate at startup. Slots are propagated using reliable broadcast [8], ensuring that either all or none of the correct nodes receive it.

To correctly *propose a slot*, it needs to include the signature of the leader, and the round in which it is proposed. Additionally, the slot must specify its parent and contain a certificate for that parent slot. Thus, any slot that has a child is certified. Finally, the slot's round must be bigger than its parent's round.

Two rules govern whether a correct process *signs a slot*. Rule 1 allows processes to only sign in increasing rounds. E.g. after signing in round 3, a correct process would no longer sign a slot in round 2. For Rule 2, processes maintain a locked slot l and only sign a new slot s_{new}, if the round of the parent of s_{new} is greater or equal to the round of l.

$$\text{parent}(s_{new}).round \stackrel{?}{\geq} l.round \qquad \text{(Rule 2)}$$

The lock l of every process is initially set to the root of the tree. On signing a new slot s_{new} a process checks whether the grandparent of s_{new} has a higher round than the process's current lock l. If that is the case, l is updated to said grandparent.

A slot s is *committed*, if it has a grandchild s_{gc}, which is certified, and no rounds are omitted between s and s_{gc}: $s.round + 2 = s_{gc}.round$. In practice, on receiving a new slot, processes check whether its great-grand parent is committed. For example, in Figure 2, slot d does not commit slot a, because round 2 was omitted between them. However, if slot e is certified, slot c will be committed.

The two rules stated guarantee safety. To propose a slot that can be signed by all correct processes, a new leader collects the last certificate (with highest round) from $2f + 1$ processes, selects the one with the highest round among them, and uses the certified slot as parent in a new proposal.

3.1 Original HotStuff

Here we explain how the original HotStuff algorithm differs from our simplified version presented above. We refer the reader to [23] for an in depth specification of the original algorithm. Original HotStuff does not require the parent of a new slot to be certified. Instead of a certificate for its parent, the slot includes a certificate for one of its ancestors. Thus, HotStuff has a more complex tree structure with each slot specifying both a link to its parent and a link to a certified ancestor. It is these ancestor links that correspond to parent links in our simplified version.

To be able to commit a slot in original HotStuff, parent and certified links must point to the same slot, resulting in the same condition as in our simplified version. Thus, during normal operation and when original HotStuff can commit slots, it is identical to our simplified version.

The removal of uncertified parent links significantly simplifies Rule 2. The original rule is as follows, where $\text{parent}_c(s)$ is the certified ancestor of s.

$$\text{ancestor}(l, s_{new}) \vee \text{parent}_c(s_{new}).round > l.round \qquad \text{(ORule 2)}$$

4 Verification

In the following we report on our effort to verify safety of simplified HotStuff in both TLAPS and Ivy. Models and proofs are available online [11]. We mainly focus on our experience with these tools, their ease of use, as perceived by us and how we modelled the ancestor relationship.

Ivy is a recent tool for the verification of distributed algorithms [18]. The default tactic in Ivy, used to verify safety properties, is a proof of an inductive invariant using an SMT solver. To avoid the SMT solver diverging, Ivy requires the specification to be written in uninterpreted first order logic. This prohibits the use of interpreted theories, e.g. integers. The required rewriting is quite straightforward. For example, instead of using integers, we require that rounds are totally ordered and use an intersection property on sets of processes, instead of process counts [20].

Additionally, Ivy requires that the verification condition must lie in a decidable logical fragment called FAU [7]. This requires the elimination of certain functions and quantifier combinations in the model. On submitting a model, Ivy checks if the given model lies within FAU and if not, specifies which functions or formulas violate conditions. Given a verification condition in FAU, Ivy either proves the invariant, or produces a counterexample to inductivity. Counterexamples can be ruled out through additional invariants.

Violations through functions can be removed by replacing functions with relations. For example, we had to rewrite the function relating a proposed slot to its parent. Violations through quantifier combinations are more difficult to remove. Following Padon et. al. [20], resolving this issue requires to introduce new relations into the model. Understanding which relations we could add required a good understanding of our model and the quantifier restrictions it implies. For example consider the following condition expressing that for a given slot z and process n, there exists a slot s in a higher round, which n has signed and which is not a descendant of z:

$$\exists s \in \text{Slot} : s.round > z.round \wedge \neg\text{ancestor}(z, s) \wedge \text{signed}(n, s) \tag{1}$$

If z or n are free variables or under universal quantification, this expression does violate FAU in our model. Realizing that we could existentially quantify over rounds, we introduced two predicates, $\text{signedIn}(N, R)$ and $\text{signedAncIn}(N, R, S)$ and replaced Expression (1) with the following, that specifies that n has signed a slot in some round r but has not signed an ancestor of z in round r:

$$\exists r \in \text{Rounds} : r > z.round \wedge \text{signedIn}(n, r) \wedge \neg\text{signedAncIn}(n, r, z) \tag{2}$$

To model the ancestor relation we included this relation as predicate in our model and update it whenever a new slot is added to the tree. This is similar to the signedIn and signedAncIn relations added to express invariants in FAU. The drawback with this approach is that it requires many invariants to be added. We

were able to prove that our ancestor relation can be implemented by checking parent links of individual slots inside a while loop.

Our finished model contains 36 auxiliary invariants, 18 or which are only concerned with the tree structure and ancestor relation. While developing the proof, we struggled with long running times in Ivy. In some cases the tool timed out, without a proof or counterexample. After a decomposition and some final simplification, however, Ivy verifies our proof in a few seconds.

The complexity of our model is significantly larger than what we found in consensus algorithms using the instance-view model. For example, to verify both Paxos or Multipaxos, the authors of the Ivy tool required only 4 auxiliary invariants. While some complexity may be due to our inexperience, we believe this also shows an increased complexity of the tree model.

One of the main advantages with Ivy was that it was easy to modularize or refactor our model. After several such refactorings, the proof was still running, or at least easy to reestablish.

HotStuff in TLA+ Based on our experience in Ivy we specified simplified HotStuff in TLA+ and proved safety using TLAPS. We also verified the safety condition for small instances of our model using the TLC model checker. In using TLAPS, we benefited from an active and helpful user group.

In TLA+ we could use both integers and functions, where our Ivy model uses totally ordered sets and relations. While the availability of integers made little difference, the ability to use functions significantly simplified the formulation of the model and the inductive invariant.

TLA+ and TLAPS allowed us to define the ancestor relation inductively. However, we were unable to apply lemmas about the ancestor relation to primed variables. This however was necessary to use the ancestor relation in the inductive invariant. Instead, we discovered Rule 2, which allowed us to formulate both the model and the inductive invariant without the notion of ancestry. The ancestor relation thus only appears in the safety property, which we prove follows from the inductive invariant.

Again we found that the verification was quite complex, due to the complex safety condition in the tree model, but also due to the need to ensure a correct tree structure in the inductive invariant. In total our proof amounts to approximately 1000 lines, plus additional 200 lines to prove properties about the ancestor relation. For comparison, previous work proved safety of Paxos [15] and Multi-Paxos [4] in around 500 lines.

5 Conclusion

We have presented the tree model for repeated consensus and a simplified version of the HotStuff algorithm. The advantages of the model, especially similarity to techniques from permissionless blockchains encourages further investigation.

Our verification efforts using both Ivy and TLAPS highlight both advantages and disadvantages of these tool and suggest that the tree model may result in more complex verification tasks, than the traditional view-instance model.

References

1. Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562. Springer (2019)
2. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. thesis (2016)
3. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4) (2002)
4. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: International Symposium on Formal Methods. pp. 119–136. Springer (2016)
5. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the tla+ proof system. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning. Springer (2010)
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
7. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer Aided Verification. Springer (2009)
8. Hadzilacos, V., Toueg, S.: Fault-Tolerant Broadcasts and Related Problems, p. 97–145. ACM Press/Addison-Wesley Publishing Co. (1993)
9. Howard, H., Malkhi, D., Spiegelman, A.: Flexible paxos: Quorum intersection revisited. In: 20th International Conference on Principles of Distributed Systems (OPODIS 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
10. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX annual technical conference. vol. 8 (2010)
11. Jehl, L.: Verifying simplified hotstuff (2021). https://doi.org/10.5281/zenodo.4711071
12. Kolaitis, P.G.: On the expressive power of logics on finite models. In: Finite model theory and its applications, pp. 27–123. Springer (2007)
13. Konnov, I., Veith, H., Widder, J.: Smt and por beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. Springer (2015)
14. Lamport, L.: Byzantizing paxos by refinement. In: Distributed Computing - 25th International Symposium, DISC. Lecture Notes in Computer Science, vol. 6950. Springer (2011)
15. Lamport, L., Merz, S., Doligez, D.: Paxos.tla (2014), https://github.com/tlaplus/tlapm/blob/master/examples/paxos/Paxos.tla
16. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
17. Lokhava, M., Losa, G., Mazières, D., Hoare, G., Barry, N., Gafni, E., Jove, J., Malinowsky, R., McCaleb, J.: Fast and secure global payments with stellar. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. SOSP (2019)
18. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: International Conference on Computer Aided Verification. CAV, Springer (2020)
19. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Communications of the ACM **58**(4), 66–73 (2015)

20. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made epr: decidable reasoning about distributed protocols. Proceedings of the ACM on Programming Languages **1**(OOPSLA) (2017)
21. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. **22**(4) (1990)
22. Vukotic, I., Rahli, V., Esteves-Veríssimo, P.: Asphalion: Trustworthy shielding against byzantine faults. Proc. ACM Program. Lang. **3**(OOPSLA) (2019)
23. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. PODC '19, ACM (2019)