Formal Specification and verification of the BFT protocol Prime

Huzaifah Nadeem
University of Pittsburgh
h.nadeem@pitt.edu

***Abstract*—Prime is a Byzantine Fault Tolerant (BFT) consensus protocol that does not yet have a formal specification. The goal of this project is to learn TLA$^+$ through a real-world example which touches on the areas of BFT protocols, formal specification, and verification of distributed systems. Moreover, another aim of the project is to write part of the specification of Prime, formally, in TLA$^+$.**

I. Introduction

A. Background

1) Prime:

BFT protocols are consensus protocols that can tolerate certain byzantine attack models. One of the earliest BFT protocols, often referred to as PBFT, was described in the work [2]. However, an exploit was later discovered in [1] which can be used to slow a system down. At a high-level, if you control a malicious server, you can really slow down the system by delaying messages at just the right time.

Prime [1] is a BFT consensus protocol that aims to address the exploit for leader-based BFT protocols. The authors specifically discussed PBFT protocol [2] to describe this exploit. An attacker can use it to slow the system down. It is not an attack in the traditional sense since the protocol still meets the correctness properties while the attack in progress but rather its performance is slowed down significantly. Further, the authors point out that, in general, existing algorithms do not evaluate their performance under attack. The authors then evaluate their BFT protocol with and without being under attack and compare it PBFT with and without being under attack. Prime gives comparable results when not under attack and significantly better results when there is an attack in progress.

2) TLA and TLA$^+$:

Temporal Logic of Actions (TLA) [4] is a logic system to formally specifiy and describe system behavior. TLA$^+$ [3] is a formal specification language that models TLA. TLA$^+$ can be used to model systems in general and it has been used to specify distributed systems where it particularly excels at. Moreover, the toolkit for TLA$^+$ includes tools such as TLC and TLAPS which can be used to model-check and mechanically verify proofs, respectively, related to the system that is being specified.

B. Project Goals

The overall goal for the project, in the context of the course, was to gain a deeper and better understanding of distributed systems' verification through hands-on experimentation. For this, the plan was to write a formal specification for, and verify, Prime BFT protocol. The protocol has not been previously formally specified and its description only exists in English Language and in the form of a C program. My goal was to use TLA$^+$ to write this specification.

This project is interesting in the course as BFT protocols are used in all the distributed systems that need to be able to tolerate intrusions. In the course, we have looked at some of the tools that aim to help evaluate such systems or papers that aim to make specification, implementation, or verification easier. This project fitd in well in the context of the course as it is touching on the areas of formal specification, verification, and BFT protocols.

The expected outcome of the project had been to have a formal specification of Prime and model-checking the specification using TLC, and verify the correctness and liveness properties for each of the subprotocols. I specifically hoped to learn how TLA is used, and in general gain experience on formal specification. The reason for using Prime for this was to get the experience of doing this for an actual protocol that can be used and not just a toy example. I also hoped that as I evaluate Prime, any issues that I find, if any, can then be further studied to better understand how prime was designed and implemented.

C. Approach Overview

Prime has 9 subprotocols out of which 7 were described in the original paper. The last 2 were added later on to the prime's codebase. From the 7 in the paper, my plan was to specify the 4 that in the main body of the paper, namely: Preordering, Global Ordering, Reconciliation, Suspect-Leader.

For each protocol, my approach was to write the behavior first, followed by confirming that the behavior by running a model-checker on it and confirm it terminated (or does not terminate) as was expected. Afterwards, adjustments were made to fix any issues with the behavior and lastly the specific properties, as invariants, were checked to confirm that those are met.

D. High-level Outcomes

I was successfully able to write the behavior of preordering subprotocol whose model seems to run as expected. I was unable to get to Reconciliation and Suspect-Leader subprotocols. Moreover, I was able to partially write the behavior of

Global Ordering. I also partially wrote the behavior for the client subprotocol as a byproduct of writing the preordering subprotocol.

Regarding proving correctness, I was only able to make partial progress on that for the preordering subprotocol. The invariants that show certain properties are held. However, even when I introduce malicious servers, some invariants are not violated in my experimentation. This can be caused by several reasons that I will describe later.

II. Approach

A. Learning TLA+

Due to the fact that I was not familiar with writing TLA+ code prior to this project, the first task was to become familiar with it. I ended up using two main resources for this. The first one was a video course [5] by the creator of TLA+ , Leslie Lamport. The course describes in detail various concepts that go into writing TLA+ code. However, the course is quite short and therefore, certain ideas are glossed over. Therefore, for reference, I used a secondary source which was the TLA+ course and reference material by Hillel Wayne [6]. I did not, however, go over each lesson but rather used it as a reference guide.

B. Prime Subprotocols

Prime has the following 9 subprotocols:

- Client
- Preordering
- Global Ordering
- Reconciliation
- Suspect-Leader
- Leader Election
- View Change
- Proactive Recovery
- Reconfiguration

The first 7 were described in the original paper while the last 2 were added later on to the prime's code-base. Therefore for proactive recovery and reconfiguration, no english description exists. From the 7 in the paper, my plan had been to specify the 4 that in the main body of the paper, namely: Preordering, Global Ordering, Reconciliation, Suspect-Leader.

C. Specification Writing Process

1) Writing behaviors:
For each protocol, my approach was to write the behavior first. Behavior describes the sequence of possible states that the system can go to. Once the behavior has been written down, the next step was to confirm that the behavior was as expected. This was done by running the TLC model checker and confirming that it finishes or does not finish (and keeps on running) for the behavior. During this process, any issues with that behavior that may come up with would be resolved after consulting the English language description from the paper and/or by looking at the codebase, depending on the ambiguity of the English description.

2) Proving Properties:
Once a behavior has been written down which the model checker has suggested to be working as expected, the next step was to write the specific properties as invariants to check against for an execution of the behavior in the model checker. Further, the invariants were made to violate by going against the protocols assumptions, such as by having too many malicious servers, to confirm the expected behavior properties.

III. Results

A. Learning TLA+

I do feel confident that I was able to learn and understand most of the concepts that go into writing TLA+ code. However, I do have a decent level of confidence about writing the behavior as compared to writing models, proofs, assertions, refinement, and the verification parts. From the whole experience of this project, it appears to me that these two categories require different skill-sets to master. Writing the behavior is relatively similar to programming at a high-level and at the system, as a whole, scale. Programming skills offer a significant base to build this skill on. With that being said, there are significant differences such as the ideas about stuttering, and enabling of actions that come from math, set theory, and logic which requires some time to learn.

On the other hand, the other pieces of writing specification including, but not limited to, writing models, proofs, assertions, refinement, and verification, are very mathematical in nature particularly related to discrete math, proofs, and logic. It seems that, from my experience, that I lack practice for this and therefore, I need to work on this skill-set in order to become better at, and becoming more confident at, writing complete specifications.

B. Writing specifications for the subprotocols

I was successfully able to write the behavior of the preordering subprotocol whose model seems to run as expected. I was unable to get to Reconciliation and Suspect-Leader subprotocols. Moreover, I was able to partially write the behavior of Global Ordering. I also partially wrote the behavior for the client subprotocol as a byproduct of writing the preordering subprotocol.

1) Preordering:
The preordering subprotocol's behavior follows standard TLA+ format with an initial predicate `Init` and a next-state relation formula `Next`. The protocol has several variables and constants. It has 3 constants, one for the set of servers, one for the possible operations, and one for the number of malicious servers that the system can tolerate. The various variables include the set of all the messages in the system and the various data structures, at a system level, for the servers such as the next sequence number, preorder summaries for other servers, etc.

The `Next` formula is shown in fig. 1. It essentially says that each server can go to any of the possible states that the system describes. These actions correspond to various functions in the Prime code for preordering. For instance,

$$
\begin{aligned}
&Next \triangleq \\
&\quad \lor \exists\, s \in R : \\
&\qquad \lor RecvAndProcessClientMsg(s) \lor mal\_RecvAndProcessClientMsg(s) \\
&\qquad \lor RecvAndProcessPOReq(s) \\
&\qquad \lor RecvPOACK(s) \\
&\qquad \lor preorderOp(s) \\
&\qquad \lor GenSummaryAndSend(s) \lor mal\_GenSummaryAndSend(s) \\
&\qquad \lor RecvAndProcessPOSUMMARY(s)
\end{aligned}
$$

Fig. 1. Next state relation for preordering

`RecvAndProcessClientMsg` action corresponds to the routine that is executed upon receiving a client message. Each of these actions corresponds to such a distinct routine. However, some actions have enabling conditions to them that can enfore a certain sequence. For instance, the `RecvAndProcessPOReq` is enabled once there is at least one `PORequest` that has been sent by a server. There are two actions with the prefix 'mal_'. These are used to mimic how a malicious server may behave. These are covered in section III-C.

2) Global Ordering:
For global ordering, I was only able to write a high-level skeleton code before running out of time. Towards the end of the term, I shifted my focus on to model-checking the preordering subprotocol as that was not progress as smoothly as I had hoped. For the continuation of the project, this subprotocol will likely require a significant amount of time and effort based on its current state.

3) Client:
The client subprotocol was not part of the original plan but rather it came about due to it being a by-product of the first draft of the preordering subprotocol. Originally, in the preordering subprotocol, I had both servers and clients. A client initiated a sequence of states, which corresponded to it sending a new operation request to one of its servers. Once this request was initiated, the `PORequest` action of the servers was enabled which enabled the whole system to make progress on the preordering. Afterwards, the client was able to receive a response back from the servers. However, while I was writing this down as described, it occurred to me that this seems to be a combination of both client and preordering subprotocols. As described in the paper, the preordering subprotocol does not need to know the specific operation in the request or any of the clients' actions as a prerequisite to make progress apart from initiating the first action. Therefore, in the preordering subprotocol a server can be thought of as arbitrarily receiving a request from a client at any point in time to allow for the system to make progress. This second approach seemed to be more appropriate for better writing the specification concisely and therefore I went with it. I ended reusing part of the client code that I had written alongside preordering to write part of the client subprotocol. At its current state, it can be thought of as a high-level combined client and preordering subprotocol. It occurred to me that this could be useful in the future as once I have separate subprotocols for client and preordering, I can use them to define an implication statement such as `Client + Preordering => High-level Client + Preordering`. This would serve as yet another property to say that the overall specification is as expected.

4) Reconciliation and Suspect-Leader:
Even though these two subprotocols were part of my original plan, I could not not start working on these in time due to being stuck on proving properties for the preordering subprotocol.

C. Proving correctness for the subprotocols

Regarding proving correctness, I was only able to make partial progress on that for the preordering subprotocol. To prove correctness, my plan was to run the model checker on the behavior to see (1) it keeps on running to show that the servers are making progress (this is because there is always another operation to process), and (2) target the key properties of each subprotocol. Regarding the former, as the behavior was described earlier, the model-checker keeps on running which should indicate that it is making progress and there are no 'syntax' errors. Another thing to note is that the model checker tells the user if an action was never enabled. This, therefore, serves to indicate that the behavior works in the normal case. For the latter, for preordering there were 3 such properties:

- With $\leq f$ malicious servers in the system, there are no holes in the sequence numbers
- Correct servers never send inconsistent summaries
- Faultly or malicious servers cannot delay an operation introduced by a correct server

The last property is hard to model in TLA$^+$ due to it using time delays. There are ways around it by using auxiliary variables to denote messages' delays. However, due to this complexity, I was leaving this property for future work. For the other two properties, my approach was to write invariants that that show that these are properties are held. These invariants are shown in fig. 2. The `NoHolesInSeqNums` confirms that there were no holes in the sequence number, whereas `cardinality_blacklist_under_f` confirms the property about inconsistency in summary messages. A server is blacklisted once it sends an inconsistent summary and this invariant in a way acts as a sanity check for this. This is where the actions with the prefix 'mal_' are used. These actions are used alongside a newly introduced constant `f_ids`, which is a set. This constant is an auxiliary constant, which means that it is not an actual constant that is used by the system but rather is used to prove some properties. This constant is set to certain server ids when the model checker is initialized by the user. The actions with the prefix 'mal_' have an enabling condition to check if the server is part of this `f_ids` set. The action then proceeds only for the malicious servers. These actions make the server skip sequence numbers and send inconsistent summaries. The hope was that this will violate the invariants if we have more than f malicious servers and hence if the model checker catches this, this will serve as part of the proof towards saying that the behavior is correct. However, even when I introduce malicious servers, the inconsistency invariant is not violated in my experimentation.

This can be caused by several reasons. One reason can be that there is a huge number of possible states that the system can go to and therefore, I just have not given it enough time to search through all. Alternatively, instead of giving it more time, I may need to somehow force the model to go to these malicious states a certain number of times. Lastly, it is also certainly possible that I have written the invariants incorrectly, or I am not reasoning about these properties correctly.

$$
\begin{aligned}
NoHolesInSeqNums \;\triangleq\; & \forall\, s \in R : \\
& \forall\, j \in R : \\
& \quad \forall\, n \in 1 \,..\, preorderSummary[s][j] : \\
& \quad\quad \vee\, \exists\, m \in preorderCertificates[s] : \\
& \quad\quad\quad m.seq\_i \leq n
\end{aligned}
$$

$$
\begin{aligned}
cardinality\_blacklist\_under\_f \;\triangleq\; & \forall\, s \in R : \\
& Cardinality(blacklist[s]) \leq f
\end{aligned}
$$

Fig. 2. The invariants to prove the properties about preordering

IV. Limitations and Future Work

A. Lessons learned

There are a few general lessons that I have learned from this project. These mostly seem quite obvious in hindsight but these were not as clear to me when I was starting the project.

One lesson is the fact that specification writing takes a significant amount of time which is more than writing a program of a similar size. Part of the reason for this is because of my lack of skills related to this. However, another reason is the fact that 'debugging' a specification and a program is quite different. Another reason is because one has to refer multiple sources to confirm whether or not the behavior that has been written is the right one. This can arise due to ambiguities and vagueness in english description and similarly logical bugs in the code-base can also make this confusing. As an example, I had to go back on forth on the specific detail of when exactly do the servers update the sequence number for other servers. The paper makes it clear that this is updated when a summary message is received, however, that would mean that a server might reply multiple times to the the request because it may take a while to receive the summary message and meanwhile it will keep on thinking that the request has not yet been replied to. The code makes it clear that there is a separate 'slot' structure, which is just a hash table, to keep track of all the messages the server has received. This detail seems to be in the small category where it is important for the specification, but it seems not important enough to include in the english description.

Another lesson is that fact that following written specifications and writing them requires a different order of time. This is very similar to how it is in programming as well where when one comes across a new project, it takes quite a while to understand what is happening and how various components are connected while writing the same code will take longer.

Lastly, the project made me realize and appreciate the fact that specification writing and programming are distinct skills. Although having a good programming experience definitely helps, but it is not a requirement. This is because specs are more mathematical in nature and having that background will likely serve equally well if not better. That might in fact be better since the parts about behaviors, model-checking, and verification requires a significant mathematical expertise in logic and proofs.

B. Limitations

My current approach of having an alternative action for a malicious server can be a practical approach to confirm that our expected properties are met, however, that is not an exhaustive approach. This way I would only be able to explore the behaviors that I can think of. In the verification works in the literature, more broad claims are made which I would be unable to do so with this approach alone.

C. Future Work

I need to review more of the literature on the verification part of formal specification writing. This would be alongside other example specifications that I would be recreating. This will allow me to really make broader statements such as claiming that there are no protocol-level bugs meeting a certain criteria, for example.

A relatively small change could be to change my behavior for preordering such that the set of operations is not a constant but rather a finite set of arbitrary operations. At each instance of the `PORequest` action a server takes, it removes an element from the set of operations and proceed from there. This might make it more clear what is happening. For instance, if the model checker keeps on running, that would no longer be a good sign as it should stop once the set of operations is empty. However, a major drawback to this approach would be limiting my number of explored states depending on the size of this operations set. Considering that the model checker explores hundreds of thousands of states, it seems impractical to scale this set of operations to such a large number when defining the model.

Another next step is to spend more time on the model-checking part of preordering. As described in section III-C, the model checker does not behave as I expect it to. One of the immediate next steps would be to figure out the reason for that and work around or fix the problem.

Lastly, a major next step for me is to spend more time on learning the verification part. On that front, this whole experience is pointed me towards recreating examples that I have found online. Since in that case the example would serve as a reference for me to check against rather than just having the english language description and code implementation to manually confirm that I am writing the correct behavior.

References

[1] Yair Amir et al. "Prime: Byzantine Replication under Attack". In: *IEEE Transactions on Dependable and Secure Computing* 8.4 (2011), pp. 564–577. DOI: 10.1109/TDSC.2010.70.

[2] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. New Orleans, LA: USENIX Association, Feb. 1999. URL: https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance.

[3] Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2003.

[4] Leslie Lamport. “The temporal logic of actions”. In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726. URL: https://doi.org/10.1145/177492.177726.

[5] Leslie Lamport. *The TLA+ video course*. URL: https://lamport.azurewebsites.net/video/videos.html.

[6] Hillel Wayne. *Learn TLA+*. URL: https://old.learntla.com/introduction/.