Consensus Protocols in TLA+

Justin Kim

May 2024

Abstract

It would be ideal to have a Byzantine consensus protocol that tolerates $n/2$ faults without requiring a very strong synchronous network assumption. To allow a small level of asynchrony, a relaxation of the model known as sluggish synchrony has been shown to achieve nearly the same bounds. There are many different synchronous Byzantine consensus protocols that each benefit specific applications, and in this paper, we examine if we can use formal methods to create sluggish synchronous protocols that closely resemble the original ones. We look at the difficulties that come with using TLA+ for this purpose, and why it does not seem to work.

1 Background

To achieve availability in distributed systems, redundancy is necessary because faulty machines are an inevitability. A key component of this redundancy is the notion of consensus, which enables the system to not only detect a fault, but continue to be available despite the presence of faulty machines. It seems impossible to completely eradicate faults, and even if that were possible, other bugs due to software bugs, transient silent data corruptions, or any form of human error is just as hard of a problem to solve. As such, a focal point in the design of these systems is having efficient, safe, and live consensus algorithms, in order to minimize the cost of redundancy necessary to reach the design goals.

Like traditional algorithms, consensus algorithms are specific to the model that they are designed for. Differences in what the network looks like, how the client interacts with the system, or the types of faults that we can encounter all influence what is possible or practical for the algorithm. On the other side, the objective of our algorithm is also dependent on the application, since in some settings a very strong guarantee of safety or liveness isn't always necessary.

1.1 Model Assumptions

In this project, I will focus on modeling Byzantine faults, which allows for a node in the system to exhibit any arbitrary behavior deviating from the protocol. Assuming there is some notion of network synchrony, we assume that messages from good nodes will always be received within a known amount of time, and it is well known that to tolerate up to f byzantine faults, we only need $2f + 1$ nodes to achieve safety and liveness. In practice, however, there is no such thing as a truly synchronous network, so often we must make some sort of relaxation to this network model.

A complete relaxation of this is the asynchronous setting, in which there are no guarantees of when or even if a message is received. It is well known that in this setting a consensus algorithm requires at least $3f + 1$ nodes to be safe from failures caused by network partitions. This bound doesn't improve even if we add a partial synchrony constraint to the network, which is a relaxation of the asynchronous model in which it can be assumed that messages will eventually be received [2].

In order to bridge this gap, there is an alternate weak synchrony assumption known as the mobile sluggish synchronous model [1], in which we assume that there can be f faulty nodes at any given point in time, but there are two types of faults. Like in the synchronous model, nodes can either be byzantine or honest. However, we also allow for a small subset of the honest nodes to be "sluggish" for some time, in which we assume that the synchrony assumption does not hold for the duration of this sluggish status. In this model, the set of sluggish nodes can change arbitrarily, however, the total number of sluggish or byzantine nodes

is guaranteed to be at most f. Note that the set of byzantine nodes does not change, as that would be impossible to design against.

We can see that this model can tolerate f combined byzantine and mobile sluggish faults with $2f + 1$ total nodes [1], which in some applications can significantly better than the partial synchronous setting. In most settings, we don't expect to see half of the nodes exhibit truly byzantine behavior, so this relaxation is a practical way to not lose too much over standard asynchrony assumptions.

1.2 TLA+

TLA+ is a specification language [5] that was intended to be a helpful tool in the software engineering process. Given a software implementation of some consensus protocol, it's possible to formally define and check whether or not a certain set of constraints are guaranteed to be satisfied by a model of that implementation. At a high level, a TLA+ specification can be thought of as a formal grammar of sorts that defines the set of all possible executions of a program. Because the nature of distributed protocols requires a lot of nondeterminism, especially when dealing with byzantine behavior, tools like these are necessary to achieve any formal guarantees, since it may be impractical to get good coverage simply through testing.

2 Consensus Protocols

In order to understand how TLA+ was used in this project, I will first introduce two consensus protocols. The first is Three-phase commit (3PC), a somewhat naive approach to achieving consensus in an asynchronous setting in which crash faults are enabled, using three rounds of communication between a leader and group of servers. The second protocol is a Byzantine fault tolerant (BFT) protocol called Sync HotStuff, which works in a synchronous setting and with simple modifications can work in the relaxed sluggish model as well. The code for these protocols can be found here at the link https://github.com/rlajustin/cs672-final/, although the implementation for Sync HotStuff is far from finished, which we will discuss later.

The primary goal of all consensus protocols is usually satisfy correctness and liveness. The correctness condition requires that no unsafe behavior occurs, like committing a transaction without the rest of the network knowing, but the time frame at which the network must know can be variable depending on what's needed. Liveness has to do with whether or not consistent progress is made. Since it's impossible to guarantee both of them in a traditional asynchronous setting, we usually add an additional assumption, which is that at some point in the protocol execution, the network will start to behave nicely, and that point it's required that liveness is satisfied.

2.1 Three Phase Commit

Upon receiving a request from a client, the following protocol is executed by the coordinator c.

1. **Ready:** The coordinator c broadcasts $\langle \text{ready} \rangle$ to all of the servers.
2. **Precommit:** After receiving a yes from every server, broadcast $\langle \text{precommit} \rangle$. If there is a timeout or the a server says no, broadcast $\langle \text{abort} \rangle$ and abort.
3. **Commit:** After receiving $\langle \text{ackP}, s \rangle$ from every server, broadcast $\langle \text{commit} \rangle$.

For every server s in the set of replicas, the following protocol is executed.

1. **Ready:** When server s receives $\langle \text{ready} \rangle$ from c, respond with $\langle \text{yes}, s \rangle$ if no problem found. Otherwise, respond with $\langle \text{no} \rangle$.
2. **Precommit:** When server s receives $\langle \text{precommit} \rangle$ from c, respond with $\langle \text{ackP}, s \rangle$ if not yet aborted.
3. **Commit:** When server s receives $\langle \text{commit} \rangle$ from c, commit and reply $\langle \text{ackC}, s \rangle$ if not yet aborted.

Figure 1: Three-phase commit

Three phase commit is a failure-resilient version of two phase commit, which trades an extra step of communication in order to eliminate the chance of an unsafe commit happening. A description of the protocol is in figure 1.

This protocol works for entirely asynchronous settings, since each step is essentially locked by requiring every server to acknowledge the previous step before the coordinator can move on. Thus, it's easy to understand why it's safe, since it's impossible for a server to receive a commit message unless the coordinator receives a precommit ack from every server. As such, even if a server were to crash, a commit can only happen if a transaction is actually safe to commit, and that server that aborts or crashes can simply learn of that commit once it comes back online.

2.2 Sync HotStuff

Introducing Byzantine faults increases the complexity. Now, you can't simply wait for every server to acknowledge a message because then liveness will never be satisfied. On the other hand, it's necessary to require that honest (non-Byzantine) replicas learn about dishonest behavior and hear messages before commits occur, since network partitions are highly problematic in this setting. A full description of Sync HotStuff and the sluggish synchronous network model can be found in the corresponding paper [1], the most important feature that we will need for the rest of this report is the idea of waiting for a majority quorum before moving on to the next step. At most $\lfloor n/2 \rfloor$ nodes are faulty (either Byzantine or sluggish), so requiring $> n/2$ acks (a quorum) will ensure that at least one non-faulty node receives a message. In a synchronous setting, if we then require that one node to forward the message to all other replicas, then we can ensure that within 2Δ time after receiving a quorum (where Δ is the maximum delay for a message), all honest and live nodes receive a message.

Another component behind these quorums is in Sync HotStuff, they come with signatures. Since they cannot be fabricated, it ensures that if a replica broadcasts a quorum certificate, honest nodes can trust that message to some degree because it means that at least one honest node agrees with the contents of that quorum, at least at the time at which it was created.

The full details and proof of Sync HotStuff initially did not seem extremely complicated, so I had thought that writing a TLA+ specification of it would not be so difficult. However, I was severely mistaken, mostly due to my lack of understanding of what the tool does.

3 Issues with using TLA+

While Lamport advertises a lot of benefits to using this tool as a key part of the engineering process, I wanted to explore how (if) it can be used in a more academic setting, looking at consensus protocols from a more abstract level. It turns out that consensus protocols under the traditional synchrony assumption often admit a somewhat simple modification that allows them to meet the same liveness and correctness guarantees in the sluggish synchronous model [4], and I initially aimed to try and formalize this somewhat using TLA+. However, at that point I had yet to gain any real experience using the tool, and for many reasons such a thing was not really feasible.

3.1 Byzantine Behavior

I didn't expect the sheer number of factors are necessary to consider when translating a high-level description of a protocol to a formal specification. One of the big reasons why I don't think it's possible for TLA+ to work at the level of abstraction I hoped it to is because of how specific every implementation is, and how these design decisions influence how Byzantine behavior is treated in the specification.

Even the concept of Byzantine behavior depends on the application. For example, in Sync HotStuff, every replica adds an unforgeable signature to every message that they send. If this cryptographic tool didn't exist, then it would be possible for Byzantine nodes to impersonate honest nodes, and as a result correctness and liveness would be impossible to achieve. Thus, in the model, we can't allow byzantine nodes to exhibit truly arbitrary behavior, because in a sense, the protocol itself places an implicit restriction on what Byzantine behavior can be.

We then have to decide whether or not we allow Byzantine replicas to attempt to forge signatures from other replicas, and if we do, that opens up a massive space of design decisions that may or may not be worthwhile to think about. However, all of this added complexity ultimately detracts from the goal of the project, which was to create a tool that doesn't actually need to address these specifics.

3.2 Synchrony

There is no standard definition of synchrony. Some definitions use a clock synchronization algorithm and operates along clock cycles [3], while protocols like Sync HotStuff essentially operate as in partial synchrony, except replicas know of the value of Δ. As far as I was able to find, all specifications for synchronous protocols use the former definition of synchrony, which is much easier to model, as it is much more simple to determine when a message should arrive by.

One of the problems that Sync HotStuff solves is the problem that arises when messages arrive too quickly, which allows it to have much better good-case performance when compared to the alternative. Such a model is possibly not impossible to create using TLA+, however, I have not been able to come up with an idea for it that does not involve the synchrony to be heavily coupled with the details of the specific protocol itself.

4 Conclusions

TLA+ is a very useful tool, and I found it to be surprisingly intuitive. Lamport has an impressive case for why it can be a key tool in the engineering process, and industry seems to agree. On the other hand, this tool likely will never see use in the application that I have tried here, where I tried to create some sort of oblivious compiler that can take a protocol as input and modify it. It turns out that there is a massive difference between the high-level description of a protocol and what that protocol actually looks like in an application, and this gap makes it impractical to use the tool with such an abstract purpose. Currently, existing traditional methods for analyzing consensus protocols are far less complex and likely will always be more practical and reliable than using this tool in this setting.

References

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync hotstuff: Simple and practical synchronous state machine replication. Cryptology ePrint Archive, Paper 2019/270, 2019. https://eprint.iacr.org/2019/270.

[2] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

[3] M. Fitzi. *Generalized Communication and Security Models in Byzantine Agreement.* PhD thesis, ETH Zurich, 2002.

[4] J. Kim, V. Mehta, K. Nayak, and N. Shrestha. Making synchronous bft protocols secure in the presence of mobile sluggish faults. Cryptology ePrint Archive, Paper 2021/603, 2021. https://eprint.iacr.org/2021/603.

[5] L. Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers.* Addison-Wesley, 2002.