Distributed Systems

William Schultz

August 23, 2023

1 System Models

- In a **synchronous** message passing system, there exists some known finite bound Δ on message delays. That is, for any message sent, an adversary can delay its delivery by at most Δ. So, every process that sends messages at time t gets them delivered by time $t + \Delta$. i.e., the whole system runs in lockstep, marching forward in perfectly synchronous rounds. For example, [ADD+19] provides a standard (modern) description of the synchronous model:

  > If an honest party i sends a message to another honest party j at the beginning of a round, the message is guaranteed to reach by the end of that round. We describe the protocol assuming lock-step execution, i.e., parties enter and exit each round simultaneously. Later...we will present a clock synchronization protocol to bootstrap lock-step execution from bounded message delay.

- In a fully **asynchronous** model, there is no upper bound on the delay for a message to be delivered, but we do assume that the delay is some finite value (e.g. chosen by an adversary). So, even though the message delay may be some unknown/unbounded quantity, we do assume that every message eventually gets delivered, even if the delay is unknown a priori.

  The nature of asynchronous networks also implies that there is no way to have a perfect failure detector in a fully asynchronous system, since you can't distinguish between a failed/stopped process and one whose messages are just taking a long time to get delivered.

- The **partial synchrony** model aims to find a middle ground between the two above models. The assumption is that there exists some known finite time bound Δ and a special event called GST (global stabilization time) such that:
  - The adversary must cause the GST event to eventually happen after some unknown finite time.
  - Any message sent at time x must be delivered by $\Delta + max(x, GST)$. That is, after the GST, messages are delivered within the known finite time bound Δ (i.e. the system has "reverted" to synchrony).

What are the fundamental differences between the synchronous and asynchronous models, and what exactly makes the latter harder?

2 Fault Tolerance

There are some fundamental requirements to establish bounds for fault tolerance in an omission fault model. If an arbitrary set of f nodes can fail by stopping at any time, then this means that if we want a protocol that makes progress, we would need to ensure that any "work" we do (e.g. executing operations, writing down data, etc.) is made sufficiently redundant so that it can be accessed even in the case of maximum node failure. So, this implies we need to write all data to at least $f + 1$ nodes, so that there is always at least one non-faulty node with the data we need to access.

This seems to imply that having $f + 1$ nodes might be sufficient for a protocol to be fault tolerant. But, this doesn't satisfy a progress requirement. That is, if we now need to write everything down to $f + 1$ nodes, then failure of f out of $f + 1$ nodes would clearly stall our protocol, since it can't do any work safely. So, our additional requirement is that both:

1) Write any work down to $f + 1$ nodes.

2) Always have $f + 1$ non-faulty nodes available that we can write work down on.

Thus, this naturally gives us a total node requirement of

$$n = (f + 1) + f = 2f + 1$$

That is, even in the case of f maximum node failures, we will always have $f + 1$ nodes available to us to write down our work, allowing us to make progress.

3 Byzantine Fault Tolerance

The earliest explicit reference to *Byzantine* faults appeared in [LSP82], though earlier work had touched on the same problem without referring to it by that moniker [PSL80, WLG+78]. They show in [LSP82] that when using "oral" messages (i.e. non-signed) messages, a Byzantine agreement solution requires $3f + 1$ processes, even in a synchronous communication model. They give an algorithm that solves the problem assuming $n > 3f + 1$, and also show that if we allow for "written" (e.g. digitally signed) messages, then in the synchronous model Byzantine agreement can be achieved with only $f + 1$ processes.

3.1 Model

The work on Practical Byzantine Fault Tolerance [CL99] considers an asynchronous distributed system where nodes are connected by a network which can fail to deliver messages, delay them, or deliver them out of order. Furthermore, it considers a Byzantine failure model i.e., faulty nodes may behave arbitrarily, subject only to the above restrictions. They do assume, however, cryptographic techniques that prevent spoofing and can detect corrupted messages. In other words, Byzantine processes can send any message, but we assume the identity of the sender of a message can be determined by the receiver [Lam11]. This can be achieved this with public-key signatures [RSA78], message authentication codes (MACs), etc.

3.2 Intuitions and Algorithm

If we assume a starting point of a classic 2-phase Paxos consensus approach, the following are some of the essential issues that arise and must be dealt with when we add in Byzantine faults:

1. **Leader equivocation**: if a leader is faulty (Byzantine), then it can trivially send two conflicting messages in the same view (i.e. with the same proposal number). This means that, for example, it could send out and accept message with its own proposal number but with a different value to each replica. Then, we would end up with a quorum of replicas having accepted that proposal, but they all have different values, so which one is the true value to agree upon?

2. **Wrong value adoption**: A leader (faulty or not) that accepts a wrong value (i.e. not highest among previously) chosen can lead to safety violation as considered in the standard 2-phase Paxos model.

A key idea of the algorithm is about how we deal with the issue of potentially Byzantine leaders. That is, we need to protect against leaders sending conflicting messages to different followers such that we would violate the constraints needed to ensure safety in, for example, classic asynchronous consensus Paxos in the standard omission (non Byzantine) fault model. If a leader is faulty and just went ahead and followed the standard 2-phase protocol used in Paxos (*prepare* + *accept*), then in the *prepare* phase it could tell different replicas arbitrarily different things i.e. tell them to accept one value and then change this value

Note: focus on reasoning about the protocol from ground up by looking at two special cases:

- Faulty acceptors (can't trust 1b or 2b messages of classic Paxos since acceptors can lie)

- Faulty proposers/leaders (can't trust 1a or 2a messages of classic Paxos since proposers can lie)

Need to have ways to protect against both scenarios. Generally, for the faulyty acceptor case, we need to increase our quorum sizes in a way that ensures a propsoer can gather enough responses to be sure a sufficient number of honest responses were received. For faulty proposers, acceptors will need to implement some additional safety check to guard against proposers sending dishonest messages about values to be proposers/accepted.

The essence of the algorithm is as follows:

1. Primary sends a $pre\_prepare(value, p)$ message for view/proposal number p.
2. Replica responds to the first $pre\_prepare$ message it receives from a primary.
3. Primary gathers $pre\_prepare$ responses from $n-f$ replicas, and then sends $prepare(v, proof)$ (note this message may be linear in size since it contains signed codes fro m up to n nodes.)
4. If a replica sees $prepare(value, p, proof)$ and $proof$ contains $n - f$ valid signatures for $pre\_prepare(value, p)$, then it goes ahead and accepts.
5. Primary then gathers $n - f$ $prepare$ responses from replicas.

Note that since we assume a public key infrastructure (PKI) set up between nodes of the system, any node can securely verify that a message was signed by some another node.

3.3 Notes

- Given $n = 3f + 1$ nodes, for any 2 quorums with $n - f = 2f + 1$ nodes, we are guaranteed they intersect in at least $f + 1$ nodes (just draw a picture). Note that if you talk to at least $f + 1$ nodes then you are sure you are in contact with at least one non-faulty (non-Byzantine) node.

References

[ADD+19] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o(1) rounds, expected communication, and optimal resilience. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers*, page 320–334, Berlin, Heidelberg, 2019. Springer-Verlag.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[Lam11] Leslie Lamport. Byzantizing paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, page 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, apr 1980.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[WLG+78] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.