Distributed Systems

William Schultz

August 27, 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 Consensus and Mutual Exclusion

The problem of *consensus* in a distributed system is to get a set of separate nodes to agree on a single value. That is, if one node marks a value as chosen, no other node can ever mark a different value as chosen. To understand the constraints of how we might solve this problem, we can start by thinking about this problem in a simpler setting e.g. a single (non-distributed) node. For an individual node/thread, solving consensus is trivial, since that node/thread can just write into a single register and then never change its decision. But, even when we introduce multiple concurrent clients (e.g. threads), the problem is nontrivial.

3.1 Single Machine, Concurrent Consensus with Locks

We assume we have a single register which represents our consensus "object", and we have multiple threads that can access the register. That is, they can read or write a value to the register atomically. If we have a locking primitive available, then we can solve the single register consensus problem easily. Each thread just acquires a global lock before it tries to do anything, reads the register to check if it has already been written to, and if it has, then do nothing, and if it hasn't, then go ahead and write whatever value you want. It is obvious to see that this upholds the basic correctness properties of consensus.

3.2 Single Machine, Concurrent Consensus without Locks

Now, what if we want to consider solving the above problem without locks? And why would we need do do this? Well, first, we can imagine that if we eventually want a solution that generalizes to the distributed setting, we won't be able to rely on locks as a fundamental mutual exclusion primitive, since a global lock primitive won't exist in a distributed setting. Additionally, locks necessarily present a potential impediment to system progress, if we assume that threads can fail or run slowly. That is, if locks can be taken unilaterally by some thread and only released by that thread, this presents potential liveness issues if that thread fails to make progress for some time and other nodes cannot proceed. So, coming up with a lock-less solution to the consensus problem seems a reasonable/desirable goal.

(**Side Note:**) The most trivial consensus algorithm for a single register in shared memory context is just to use a compare and swap (CAS) when writing to the register. Each process just runs `CAS(X, null, v)` to update the register only if it hasn't been set. The first writer always wins and consensus is always achieved. But isn't this kind of cheating? Like, don't we want to figure out a way to do consensus with weaker forms of atomic primitives?

If we think about the fundamental requirements of consensus in this "single register" model, it boils down to a simple high level requirement that threads must satisfy:

> R1. If a thread writes a value v to the register, then it should not differ from the most recently written value.

This is the very basic, fundamental requirement, simply stating that whenever somebody tries to write to the register they better not overwrite an already existing, different value in the register. In order to start working out a lock-less solution to the problem, let's consider how the lock-based solution satisfies this above requirement.

In the lock-based solution we can think about every thread as executing a simple request/transaction, that consists of the following steps:

```
acquire(lock)
if read(X) is not set:
    write(X, v)
release(lock)
```

where `lock` is the global lock shared by all threads, and `X` represents our register object, and `v` is some arbitrary value the thread chooses to write. How does such a procedure satisfy the above requirement R1? Well, first, the locking mechanism ensures that all operations are explicitly/globally ordered with respect to each other i.e. their order is dynamically assigned based on the order of lock acquisition. Based on this, it is clear then, that

C1. Reads by transaction T read the value written by the most recent transaction ordered earlier than T.

C2. After a transaction T that is currently holding the lock completes its read, no future writes will ever be made by transactions ordered earlier T.

Note that (C2) is an important but subtle condition that is required for safety. Simply reading the most recently written value is not sufficient to ensure correctness, since this doesn't say anything about writes that may occur *after* the read but *before* the subsequent write of the transaction. In other words, you need to protect against concurrent transaction writes that would invalidate the results of your read.

Ok, so let's try to take ideas from the lock-based solution and turn them into a lock-less solution. One main idea is that there is an implicit order/sequence assigned to all transactions in the lock-based approach. We might say this ordering is "implicit" or "on demand" because transactions don't really get ordered until they try to go ahead and acquire a lock. At that point, we can imagine them being implicitly assigned some sequence number in a global sequence of transactions based on the order of their lock acquisition. So, if we don't want to rely on locks, but we know that this global ordering notion works for a lock-based solution, can we try to develop an ordering mechanism that doesn't rely on locks?

Well, the naive approach is to basically just pre-assign global, totally ordered sequence numbers to all transactions. There might be a variety of schemes for doing this, but if we're still in a single machine context, we could imagine simply having a global atomic counter that hands out sequence numbers to transactions before they start. Alternatively, we could hand out disjoint, evenly distributed sets of sequence numbers to each thread at system initialization, that they can draw from whenever they want to start a new transaction. For simplicity, we can kind of ignore the details of how such an ordering assignment scheme works, but we can assume that there is some way to assign uniquely global, totally ordered sequence numbers to different transactions (note that Paxos similarly does this in a similar manner, by pre-assigning disjoint sets of proposal ids to to each proposer).

If we now assume that all transactions are tagged with a unique, totally ordered sequence number, we can try to use this to build a complete, lock-less solution to the single register consensus problem. As shown above, each thread can still execute a similar procedure as before, but it will do so without acquisition/release of locks and with a bit of extra checking related to their sequence numbers. As we said above, all that threads need to ensure are conditions (C1) and (C2). Let's consider them independently, starting with C2 first.

- (C2) This condition is fairly straightforward to handle. Whenever transaction T with sequence number n does a read, it can tag the register with sequence number n. Then, subsequent writes from transactions in sequence numbers k can check their sequence number against n. If $k < n$ then we can prevent the write from occurring, and if $k \geq n$, then we can allow the write to succeed. This clearly ensures that once a read occurs by transaction in sequence number n, no future writes can be made to the register by transactions ordered $< n$.

- (C1) This condition is a bit more tricky, since it is complicated by the fact that we no longer assume a global serialization order between transaction operations as we did in the lock-based solution. For example, if we make a fundamental assumption that transaction operations may be always be interleaved in arbitrary orders (based on the fact that we have no global locking/mutex primitive), how can we possibly satisfy C1 in a case like the following,

  ```
  read:2(X)
  write:1(X, v)
  ```

  where `op:seqno` indicates that `op` is an operation from transaction with sequence number `seqno`? That is, how can we ever enforce that a read from transaction at sequence number n will necessarily see the writes from transactions at sequence numbers $< n$, if it can't forcibly protect against such transactions doing writes after the transaction in n does its read? Well, we can't really be sure, if we make the fundamental concurrency/interleaving assumption above. So, this is where our solution to dealing with C2 comes into play. Instead of trying to ensure C1 exactly, we just explicitly prevent any future writes that would violate it. If some writes have already occurred in sequence numbers $< n$, then we will obviously read their effects when we read at transaction n, but then after we read at n, we just force the system to never execute a write at a transaction number $< n$ in the future. Note also that we aren't really impairing ourselves unnecessarily here. That is, once a transaction in sequence number n has started, we make the implicit assumption that it "overrules" any transactions in earlier sequence numbers, so there's not really any point in letting an earlier transaction go

ahread and write when a higher transaction number has already started anyway. So, we can view it as acceptable to just prevent/discard these "stale" writes anyway.

Ok, so now that we worked out how to ensure the two important correctness conditions above, we get to a final, lock-less procedure for each thread that looks like the following, where we now assume that alongside the register there also sits a "version" number register `V` that can also be written and read by threads.

```
// Procedure for transaction with sequence number N.
if read(V) > N:
    return
write(V, N)
if read(X) is not set:
    write(X, v)
```

But what if we can't atomically read and write the version number register? (TODO...) Perhaps [Her91] is the reference I'm looking for here. Note one of their main claims:

> From a set of atomic registers, we show that it is impossible to construct a wait-free implementation of (1) common data types such as sets, queues, stacks, priority queues, or lists, (2) most if not all the classical synchronization primitives, such as *test&set*, *compare&swap*, and *fetch&add*, and (3) such simple memory-to-memory operations as move or memory-to-memory swap.

Note the following table of consensus numbers of some objects:

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add
⋮	⋮
$2n-2$	n-register assignment
∞	compare&swap

See also [AH90] as possibly relevant.

3.3 Note on Compare and Swap (CAS)

Also, note that *compare&swap* primitive is kind of just implementing "lock like" mutual exclusion at a lower level i.e. for register `X`

```
CAS(X, old, new):
    if X == old:
        X := new
        return true
    return false
```

That is, it basically allows you to do a read and a write atomically, as if you were doing it under a "virtual" lock. It's just that in practice, this "lock" isn't explicit and failure while holding a lock isn't an issue, since the operation is truly atomic, and if such a failure occurs, the whole operation would be aborted and this kind of "virtual" lock automatically released.

(In Paxos phase 1b, do acceptors actually have to do some atomic read-write transaction in order to check if given proposal number is newer than their own...?)

4 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.

4.1 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.

4.2 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.

4.3 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.

[AH90] James Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441–461, sep 1990.

[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.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, jan 1991.

[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.