Lecture 9: Byzantine Fault Tolerance

Aurojit Panda

1 Failure models revisited

So far we have focused on the fail-stop model in class. We talked about the fact that there are others for instance crash-recover, but have not really covered them. As a reminder, the fail-stop model means that

- Processes can fail.
- A failed process neither sends nor receives any messages.

We have seen with FLP and other results that the possibility of process failure impacts what protocols we can implement. However, our protocols have only needed to protect against a possible lack of messages.

In this lecture we extend the set of failures to Byzantine failures. This is the strongest failure model a protocol can be designed to work under since a failed process can **exhibit arbitrary** behavior including

- Neither sending nor receiving messages.
- Sending contradictory messages to different processes.
- Sending messages carefully crafted to trigger bugs in the protocol.

Note that our assumption of Byzantine failures will at times require changes to some of the properties we desire. For example consider **validity** in consensus. As a reminder validity means that the output all processes decide on must have been input for at least one process. But in the presence of Byzantine failures we run into a few issues:

- We cannot actually determine inputs to all processes: a failed process might claim contradictory inputs.
- A-priori we have no knowledge of whether or not a process is "failed", and therefore cannot try to limit validity to only inputs at **correct** processes.

Two possible resolutions exist for this

- One relax validity requirements, this is what the Lamport paper ends up doing.
- Enhance inputs so that processes cannot manufacture inputs. This is the approach adopted by the PBFT paper.

Both require adjusting the definition of validity to capture the approach adopted.

1.1 Why Byzantine Failures

Given all of the complexity with fail-stop, why look at Byzantine failures? There are several practical reasons for doing so:

- First, assuming Byzantine behavior allows us to avoid assumptions about how bugs manifest in programs, and how they impact processes. This approach has been adopted for use cases such as airplanes (e.g., in Boeing 777s) and space crafts (e.g., SpaceX in the Falcon) where safety is crucial. One common approach is to have multiple teams write software and use a BFT consensus protocol to agree on the next action to be taken.
- Second, recent technologies like the blockchain rely on having multiple **untrusted** computers collaborate on some task, e.g., on ordering transactions. The fact that machines are untrusted leads to the use of Byzantine fault tolerant protocols. We will **not** talk about this use case, in particular we are assuming that the set of processes are known a-priori.

1.2 Note on Nomenclature

The Byzantine empire or the Eastern Roman Empire was one of the successor states to the Roman empire. The Byzantine empire adopted Rome's bureaucracy, which had many levels and was quite convoluted. Furthermore, the Byzantine army, much like the army of the later Roman empire tended to play a big role in succession, etc. which is what leads to the names used in this paper.

2 Byzantine Generals Problem

This is a form of consensus, except we relax validity and agreement is implicitly decided based on collecting inputs from all participants and then using a deterministic function to decide on the agreed value.

The problem is thus reduced to a problem where all processes exchange messages to compute a vector of inputs (whose length is equal to the number of processes).

2.1 Correctness Criterion

This leads to two correctness criterion:

- All correct processes (loyal general) agrees on the vector of inputs.
- If process p is correct and has input i_p, then the input vector I computed by any correct process q has $I[p] = i_p$.

The first of these conditions ensures that all correct processes reach agreement, while the second ensures that a correct processes input cannot be lost (among other things this avoids trivial solutions).

Rather than solve the entire problem, we can focus on how to reach agreement on a single element of the vector, e.g. $I[P]$. In this case we can reduce the two correctness criterion above to two new ones:

- All correct process agree on the input provided by process p.
- If process p is correct, then the input agreed upon by correct processes must be the input provided by p.

2.2 Impossibility Result

Assumes:

- Network is reliable, does not corrupt messages.
- Receiver can identify sender.
- Absence of messages can be detected.

No protocol can be correct if $\frac{1}{3}^{rd}$ or more of the processes are faulty.

- Proved using indistinguishability (see lecture or paper.)

2.3 Protocol for $n \geq 3f + 1$

See paper or lecture. Need to consider both

- Safety: Based on every process broadcasting all received messages, and arguments about quorum intersection.
- Liveness: Based on an assumption that the "Absence of messages can be detected." **Question**: How would we implement this in practice?

3 Cryptography

If you look at what precedes this, a processes ability to misrepresent messages sent by other processes is a core component of our dilemma. One can think of this as a problem of **authentication**, i.e., a problem with ensuring that a message's purported sender did in fact send it.

3.1 Digital Signatures (and public key cryptography)

3.1.1 Public Key Cryptography

Two keys public k_u and private k_i. Assumption: Public key is known to all (how?), private key is only known to a designated owner. Two functions encrypt and decrypt such that $dec(enc(m, k_u), k_i) = m$ for all m. Formally decrypting with the private key is the inverse of encrypting with a the public key.

3.1.2 Digital Signatures

Often (depending on function) the keys are symmetric and we have $dec(enc(m, k_i), k_u) = m$.

Note that we have assumed that k_i is only known to one process, while k_u is known to all processes. This means that the tuple $(m, enc(m, k_i))$ can be used to **authenticate** the message. Any receiver can check that $m == dec(enc(m, k_i), k_u)$ and the assumption means that no other process could have produced $enc(m, k_i)$ thus linking process and sender.

3.2 How digital signatures help with the Byzantine Generals Problem

We can use DS to detect a failed node: essentially we can break the indistinguishability proof that led to the problem with 3f nodes and f failures:

we can in fact now distinguish between the the situation where a proposer is failed vs one where a forwarder is failed.

3.3 HMAC

While digital signatures provide a way to authenticate messages, public key cryptography has generally required more compute cycles than other primitives such as cryptographic hashing.

A cryptographic hash function H is one that takes an arbitrary length message m and produces a fixed length output. There are additional requirements on $H(m)$:

- $H(m)$ must be deterministic.
- $H(m)$ should be irreversible, and not leak any information about m. Achieving the later means that a single bit change in m should potentially result in $H(m)$ changing completely.
- Finding collisions should be hard.

The problem is that cryptographic hash functions do not require a key and cannot automatically be used to authenticate messages. Many of the natural techniques for introducing a key (e.g., computing $H(m + k)$ for key k) don't do well on security.

HMACs are a way to turn a cryptographic hash function to a keyed variant, roughly they work as follows:

1. Given key k produce $k_o = k \oplus \text{opad}$ and $k_i = k \oplus \text{ipad}$. We assume opad and ipad are widely known.
2. Compute $H(k_o + H(k_i + m))$

How to use this for authentication? Any two processes use a protocol to agree on a key. Once they have they attach HMACs to messages exchanged between them. Assuming key k is not leaked, this allows either pair to authenticate messages. By presenting key k and messages pairs can also prove that messages must have originated at either end.

4 PBFT

4.1 Differences from Byzantine Generals

- Construct RSM.

- Do not allow faulty nodes to inject false commands.
- Focus on performance

How?

- Use signatures to ensure that commands actually originate at the client.
- Minimize the use of public key cryptography to limit performance overheads. Optimized version based entirely on HMACs. Not the version presented in the paper we read, which is what we will focus on.

4.2 Normal Protocol

Each view has an associated view number v. A view number v can be mapped to a primary/leader by $vmodn$ where n is the number of processes.

Given this the way things work in the normal case is:

- Client sends a signed request m to the primary.
- Primary assigns an index i to the request, and broadcasts a pre-prepare message $\{\sigma_p(\{\text{pre-prepare}, v, i, d\}), m\}$. Here σ_p indicates a tuple signed by the primary, remember m is signed by the client. d is a hash computed from message m.

  The use of a signed m makes it so the primary cannot invent a command.
- Upon receiving a pre-prepare message each replica (follower) checks
  - That it is currently in view v.
  - m authenticates, and d is in fact the hash of m.
  - It has not previously accepted a message for index i.

  If these checks pass, the replica f broadcasts a prepare message, $\sigma_f(\{\text{Prepare}, v, i, d, f\})$ where σ_f signifies the message is signed by follower f.

  Broadcasting a prepare message allows followers to protect against faulty primaries trying to commit messages in different orders.
- Each process now waits for $2f$ prepare messages that are identical to the one it sent (module signatures) from other replicas. Overall once a replica receives $2f$ prepare messages it knows that at least $2f + 1$

processes agree that index i is assigned to message m. Since f of these might be faulty, we know that $f + 1$ non-faulty nodes agree. By the existence of at most f faulty nodes we get a quorum intersection containing at least 1 correct node for any prepare round. We say a process that has received a sufficient number of prepare messages is "prepared".

  Once a process f is prepared it broadcasts a commit message $\sigma_f(\{\text{commit}, v, i, d, f\})$.

- Each process that is prepared now waits for $2f + 1$ commit messages. Again waiting for $2f + 1$ commit messages allows the process to know that at least $f + 1$ other correct processes have prepared. Once the process receives $2f + 1$ commit messages it executes the command.

  Distinct from Raft and other non-BFT RSMs, each process that executes a command **directly** sends a response to the client. Clients are responsible for waiting for $f+1$ responses before considering their command executed. Having clients count responses from followers protects against a case where a **faulty leader** might respond to a command without actually replicating the command.

4.3 Liveness in the Face of a Byzantine Leader

What happens if a primary simply does not make any attempt to commit client messages? This problem is reolved by having clients send requests to all replicas after a timeout.

Replicas then forward the request to the primary and starts a timer awaiting a pre-prepare message. If the pre-prepare message does not arrive in time the replica attempts to change views.

Now this creates a new problem: a failed process might attempt to slow down or stall processing by triggering frequent view changes. To protect against this a view change needs to be triggered by 2f processes which (along with the 1 leader for the new view) means that 2f+1 processes agree to change views, of which at least $f + 1$ must be correct.

4.4 Making sure that the Primary is Eventually Correct

Finally, how do we know that view changes will eventually result in arriving at a correct primary. The fact that leaders are deterministically assigned to views leads to this, at most f view changes are required to arrive at a correct primary.