Practical Byzantine Fault Tolerance

System Model

It’s assumed an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, or deliver them out of order. It’s pretty much the same configuration from 🫂 Replication for Fault Tolerance.

Concepts

Safety

Introduction to Reliable and Secure Distributed Programming

In modern computing a program is usually distributed among several processes. The fundamental challenge when developing reliable and secure distributed programs is to support the cooperation of

https://link.springer.com/book/10.1007/978-3-642-15260-3

Concurrency: State Models and Java Programs

Chapter 7. Safety and Liveness Properties A property is an attribute of a program that is true for every possible execution of that program. Properties of interest for concurrent programs ... -

https://www.oreilly.com/library/view/concurrency-state-models/9780470093559/ch07.html

According to this book, safety properties state that the algorithm should do anything wrong. In this case.

The Safety Properties also guarantees:

- **Agreement:** *no two correct processes decide differently.*
- **Validity:** *the decided value depends on the input values of the processes.*

Liveness

Clients eventually receive replies for their requests. Thus, for every execution, a value is decided in a given protocol.

Theorem

FLP impossibility result

Consensus (computer science) - Wikipedia

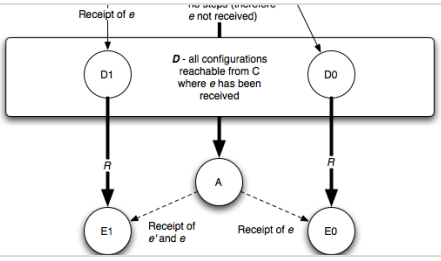
A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. This often requires coordinating processes to reach consensus, or agree on some data value that is needed during computation.

https://en.wikipedia.org/wiki/Consensus_(computer_science)#The_FLP_impossibility_result_for_asynchronous_deterministic_consensus

A Brief Tour of FLP Impossibility

One of the most important results in distributed systems theory was published in April 1985 by Fischer, Lynch and Patterson. Their short paper 'Impossibility of Distributed Consensus with One Faulty

https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/



It was proven that in deterministic consensus algorithm, **if at least one peer crashes, it becomes impossible to reach in a consensus. There's no deterministic consensus protocol that is both live and safe.**

This is due to the fact that it's not possible to distinguish a crashed process from a slow one. The algorithm here has two choices:

- If the algorithm answers, the response may not guarantee safety.
- If the algorithm give no answer, then it violates liveness.

Paxos sacrifices liveness for safety.

System model

The **network** can fail in many ways: the network can fail in deliver a message, it may send duplicate messages, messages are delivered out of order, messages are delayed.

Nodes/processes may fail arbitrarily.

This system uses cryptography techniques to prevent spoofing, detect corrupted messages, etc.

Notation

- $D(m)$ digest of a message m
- $< m >_{\sigma i}$ message m signed by process i. It is a encrypt then sign.

Service properties

The service uses **state machine replication** i.e a replicated service with a state and some operations: each replica maintains the service state and execute operations, which must be deterministic. Since the system is deterministic and all replicas starts at the same state, we can assure **safety** regardless faulty servers.

Also, the system has **safety**, since the replicated service satisfies **linearizability**: it behaves like a centralized application that executes operations atomically one at time and in the end the final state is correct. Because, if the system agrees in a response than the process is executed.

The **liveness,** however cannot be assured in an asynchronous system, since we cannot tell the difference between a process that takes too long to answer and a process that is just too slow.

The **resilience** is decided can be achieved in the following way:

- It tolerates f faulty replicas with $n = 3f + 1$ and this is optimal.
- Since f nodes can fail, we can proceed with the action by communicating with $n - f$ replicas.
- But, how can we tell apart faulty replicas and slow replicas? The worst case is: from all the replicas that didn't answer, all of them are slow non-faulty replicas. Thus, from the $n - f$ replicas that answered, f from $n - f$ replicas may be faulty. What should be the size of n in this scenario, so that we can reach the correct answer by consensus?
- The correct number of replicas is $n - 2f$ and the number of correct processes must be the majority from the $n - f$ nodes that answered. Thus $n - 2f > f \rightarrow n >$

$3f$.

The algorithm (SMR)

The replicas move through a succession of configuration called *views.* In a *view* we have the *leader(primary replica),* where its id is $id = v \ mod \ n$. Where n is the number of replicas $3f + 1$ and v is the view number.

The view changes occur upon the suspicion of the current leader.

General logic

- The client send a request to the primary
- The primary **atomically** broadcasts the request (this ensures a total order on the delivery of the messages by non-faulty replicas)
- The request is received by each replica in the view and sends a response directly to the client
- The client waits for $f + 1$ equal responses from the replicas. When this condition is satisfied the client reaches to an answer.

Client perspective

- A client *c* request to execute a server operation *o* sending `<REQUEST, o, t, c>`$_{\sigma c}$. Where *t* is the timestamp of the client's local clock. This ensure the **exactly once** semantics, since this measure is taken from the monotonic clock.
- The server atomically broadcasts the request to the replicas
- Replica `i` executes the request and sends `<REPLY, v, t, c, i, r>`$_{\sigma i}$ with the result `r` of the execution of the operation.

Case the client doesn't receive the request

- It will **broadcast** the request to all replicas.

- If the replica already received the message, it will reply again to the sender. The replicas are able to do this by keeping tracking of the last message they sent to each client.
- If the replica didn't receive the request, it will send it to the leader. If the leader does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas.

Quorums and certificates

The algorithm assures the two properties:

- **Availability:** There is always a quorum with no faulty replicas. Even if a quorum fails (somehow) the view will change until it got it right.
- **Intersection:** Any two quorums have at least a correct replica in common (they intersect in at least f+1 replicas).

A **quorum certificate** is a set with one message for each element in quorum, ensuring that relevant information has been stored.

A **weak certificate** is a set with at least $f + 1$ messages from different replica. The set $f + 1$ replies a client must receive before returning the result is a weak certificate, the **reply certificate.**

Normal-Case operation

Every replica has a **log** containing the messages the replica has accepted. This log is eventually cleaned by the garbage collector.

Te replicas obey a three-phase protocol to assure that all replicas agree in the order of the message. These protocol has the following phases:

- Pre-prepare
- Prepare
- Commit

Pre-prepare

The pre-prepare phase is initialized by the leader (primary replica), by sending in multicast a pre-prepare messages. The message has the following format:

$$\texttt{<<pre-prepare, v, n, d>}_{\sigma i}, m >$$

Where `v` is the number of the view, `n` is the message number and `d` is the message digest (`m`).

When a replica receives this message, first it verifies:

- if it belongs to the view `v`
- checks the signatures and that `d` is the digest of `m`
- the `n` is between the water marks `[h,H]` (that will be explained in the garbage collector)

Prepare

Each replica sends a prepare message in multicast and adds each `prepare` message received to the log. A message of this type is accepted only if:

- The view is the same of the current replica
- Its signature is correct
- n is between `[h,H]`

When the replica receives $2f$ $\texttt{<prepare v, n,d,i>}_{\sigma i}$ messages. it adds a `prepared` message to the log, attesting that the replica is ready to the next phase and that at least $f + 1$ non-faulty replicas agrees in the message order and that it's correct.

> The pre-prepare and prepare phases guarantee that a replica cannot obtain prepare-certificates for the same view and sequence number and requests with different digests.

Commit phase

By collecting the `prepared` messages from at least $f + 1$ non-faulty servers, a replica starts the commit phase.

By receiving a $\texttt{<commit, v, n, d, i>}_{\sigma i}$ a replica verifies that:

- It belongs to the view
- The number `n` is between `[h,H]`
- The signature is valid

After receiving $2f + 1$ commit messages with the same view, sequence number and digest, received from different replicas it adds the `commited-local` and `commited` variables to the log.

> The commit phase ensures that if a replica commited a request that request is prepared by at least f+1 non-faulty replicas.With the view change protocol it also guarantees that the non-faulty replicas agrees on the sequence numbers of a commited request. Also guarantes that any request commited at a non-faulty replica, will commit at all non-faulty replicas, possibly in different views, due to the fact that this is a deterministic system.

Request delivery and execution

After the commit phase (when the replica possesses the `commited` “certificate”, it can immediatly starts the execution of the request, except if it hasn’t finished executing processes with lower sequence number. This exception allows the replicas to execute requests in the same order as provided in the safety property.

After executing the request, each replica will send directly to the client, that will reach to the right answer.

> Replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to the client. This guarantees the exactly-once semantics.

Ensuring order across view changes

Garbage collector

To ensure **safety** a replica cannot just discards the message with a certain sequence number as soon as it executes the request. It must be sure that the other replicas had already executed it.

We need here some state synchronization, but synchronize at every request execution would not be efficient. So we can do this task after the execution of every x requests. This synchronization is useful if a replica falls behind after a partition and when it comes back alive it's necessary to update the log.

Checkpoint

Thus, periodically, i.e after every K requests, a replica **checkpoints its state** and generates a proof of correctness of that checkpoint. After generating such proof, the **checkpoint** is classified as a **stable** one. Thus, if a checkpoint is for sure correct, a replica can discards all the messages with sequence number lower than the last sequence number received in the checkpoint.

The replica not only saves the stable checkpoint, but also one of more checkpoints that are not stable and the current state.

However, to get the proof of the state, it's necessary to multicast the message.

Checkpoint proof generation

Upon a checkpoint (i.e after x requests has been received), the replica i multicast a `<checkpoint, v, n, d, i>`$_{\sigma i}$ message to all replicas, where `n` is the sequence number of the last request whose execution is reflected in the state. `d` is the digest of the state.

The **stable certificate** is received when $f + 1$ `checkpoint` messages are received from one replica (may include itself). This shows that at least one replica has produced the correct state. This proofs the correctness.

Garbage collector in action

After collecting a stable certificate, a replica can discard all `pre-prepare, prepare` and `commit` messages with sequence number lower than the sequence number of the last request whose execution modified the state. It also can discard all earlier checkpoints and respective `checkpoint` message.

Update to low and high water marks

A replica advances the low and high water marks every time it runs the checkpoint protocol.

- h is set to the sequence number `n` of the last stable checkpoint.
- H is set to $h + L$, where L is the frequency (or a small multiple of it) that the checkpoint is generated. Thus, if the checkpoint frequency is 100, it means that after 100 requests a checkpoint will be generated. This makes it unlikely for replicas to stall waiting for the checkpoint to become stable.

View change

It's necessary to change a view when it is suspected that the leader (primary) is faulty. To ensure liveness, it's necessary to change the view.

Leader failure and timer

Each peer as a counter. The timer is executed by the reception of a request and if the timer is not already running. This timer keeps running until the task is executed and while this doesn't happen we say that the replica is **waiting** for a request.

Upon timeout

Upon the timeout, the replicas starts the process of view-change.

- The replicas first stops accepting requests and only accept messages of view-change, new-view and checkpoint.
- The replica, upon the timeout, send in multicast `<view-change, v+1, n, C, P, i>`$_{\sigma i}$. The `v+1` is the number of the new view, `n` is the sequence number of last stable checkpoint known by i, `C` is the set of checkpoints to verify the correctness of `n`, `P` is a set of sets containing the pre-prepared messages of other replicas. In other words, `P` is a set of the prepared certificate for each request prepared at replica i, with sequence number greater than `n`.

The new primary

The primary of view `n+1` starts to collect requests of view-change. The **new-view certificate** is a set of $2f + 1$ valid view-change messages for view $v + 1$.

The `<new-view, v+1, V, O, N>`$_{\sigma l}$ contains the **new-view certificate `V`.** The `O` and `N` are sets of `pre-prepared` messages (without the respective request) that propagate sequence number assignments from previous views. After sending the `new-view` messages, the leader finally enters in the `v+1` view and starts accepting messages.

Computing O and N

The leader determines the sequence numbers:

- **`h`** of the latest checkpoint in `V` and the highest `H` in a message in a prepared certificate in `V.`

Then the leader creating `pre-prepared` messages between `[h,H]` to all the replicas in the view. There're two cases here:

- **There's a prepared certificate** in `V.` In this case the leader can get the message digest and create `<pre-prepare, v+1, n, d>`$_{\sigma e}$ and adds it to `O`.
- **Otherwise,** it adds `<pre-prepare, v+1, n, null>`$_{\sigma l}$ to `N`, where null is a special digest for a no-op request.

In the end the leader merges `O` and `N` to its log.

State update at New Leader

If `h` is greater than the sequence number of its latest stable checkpoint, the leader simply adds a stable certificate to the checkpoint with sequence number `h` and discards the log information.

If `h` is greater then the current state, it also updates its current state to that of the checkpoint with sequence number **`h.`**

The leader still may be missing the latest checkpoints and the latest messages commit or prepared, but these are not added to the `new-view` message, the leader can get this values from another replica. The leader can just fetch from a replica that has correctness in `V`.

Receiving new-view in replicas

The replicas verifies the **`new-view`** message:

- If it's properly signed
- If it contains a valid **`new-view`** certificate, by verifying the content in **`V.`**
- Check the correctness of **`O`** and **`N`**, by recomputing them from the **`new-view`** certificate, like the leader does

After checking these information it enters in the view **`v+1`** and adds the messages **`O`** $\cup$ **`N`** to its log. For each message in this new set, it multicasts a **`prepare`** message to the other replicas and as described earlier re-execution of client requests is prevented by using stored information about replies sent to clients.

Questions

1) What is safety? What are its properties?

▼ Answer

Means that the program holds the property of linearizability: it behaves like a central program that executes the operations in order.

2) Why safety is insufficient to guard against faulty clients?

3) What is liveness?

4) What is the FLP impossibility theorem? Explain.

5) Why $3f + 1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty?

The following question answer with true or false and justify:

- An asynchronous system guarantees safety.

▼ Answer

False. If the system is asynchronous we cannot tell the difference between a process that is faulty and a process that just takes to long to answer.