Replication for Fault Tolerance

https://en.wikipedia.org/wiki/Byzantine_fault

The problem of byzantine fault can be explained with the generals analogy.

Byzantine fault

A byzantine fault is any type of error that occurs in a system that might cause it to send wrong messages or even to shutdown.

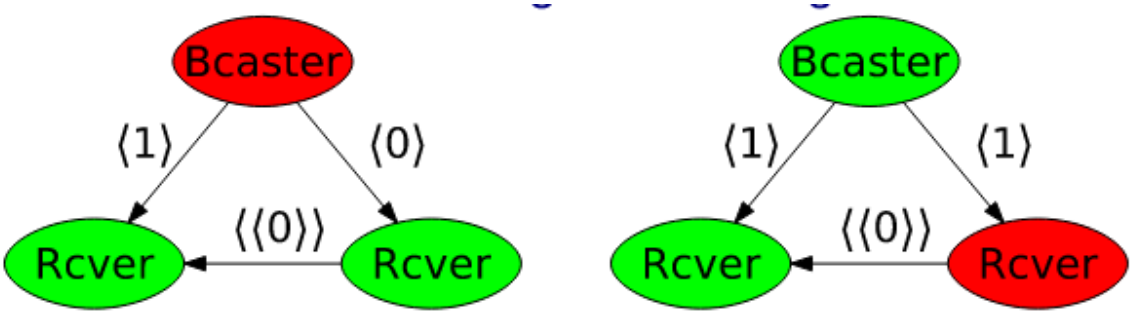
Byzantine failures

Byzantine failures is the lost of a system service due to a byzantine fault in **systems that require consensus.**

Byzantine fault tolerance is only concerned with broadcast consistency (Byzantine General Problems BGP). But how can we define a reliable broadcast:

- **Agreement** All non-fault processes delivers the same messages
- **Validity** If the broadcaster is non-faulty, then all non-faulty processes deliver the message sent by the broadcaster.

When the broadcaster is not reliable, other peers may communicate with each other to reach in agreement. However, there's no solution for this case in a system with 3 processes. The result reached from this analysis is that the number of reliable processes must be the majority.



If the Broadcaster is not reliable, the receiver from the left will receive a wrong value from the receiver from the rigth. The same occurs when is the receiver from the right the is faulty.

Byzantine quorums

The fundamental property of these quorums is that if one operation depends on others, it means that the quorums must overlap.

Formalization

For a **set of servers** U**,** a quorum system is a non-empty set of subsets of U, where every pair intersects each other. Each set in Θ is a quorum.

$$\Theta \subseteq 2^U$$

The thing is that **some servers** in U may have byzantine failures (e.g report a wrong version number). How can the coordinator know what is the last version number?

Now consider that the equation on the right what we call the *fail-prone system.* It's a non-empty set of *subsystems* of U,

$$\beta \subseteq 2^U$$

where $\mathrm{B} \in \beta$ represents a set with all faulty servers.

Here we suppose that the cliets do not fail. Here let's suppose that the network communication doesn't fail and the bounds of message delay is not known.

Problem definition

Let's approach the problem in the following way:

- We want to perform read and write operations on a variable x replicated by all the systems U. Thus, each server stores a **timestamp** and a **value** for the variable x.

The protocol requires that the clients are able to choose different timestamps for the write operation. Thus, a client c chooses its timestamp from a set of timestamps T_c that doesn't overlap with other sets of clients (e.g appending the client id to an integer). On this way we guarantee that the timestamps are all different.

Access protocol

Write

For a client c to write some value v to an arbitrary quorum the following approach needs to be executed:

- c first queries all the servers from a quorum Q to obtain a set of timestamps $A = \{< t_u >\}$
- Then the client chooses a timestamp from the set T_c **greater than the highest timestamp in A and any timestamp that was chosen in the past.**
- After choosing the timestamp, the client send the pair $< v, t >$ to the target quorum Q'.

Remember the write protocol must modify the entire object, not just a part of it.

Read

Reading is an easier task. For a client read a value x:

- It queries the servers to obtain a set of value/timestamp $A =< v_u, t_u >_{u \in Q}$ for some quorum Q.

- Then it will apply a deterministic function $Result(A)$ to find the most recent version.

It is just like the normal quorums read protocol.

Asynchrony

Basically we must receive a **reply from all the servers.** In an asynchronous scenario this is the only way of receiving the response from all the non-faulty servers. And quorums are defined as there's at least one quorum of **non-faulty** servers at any time.

> I think we can say the last sentence, because we can group the failures by computers that have similar characteristics such as servers with the same hardware or same software platforms. These machines have correlated probability of being captured by an error.

Begin and end of events

This section is kind of obvious, but let's write it for the sake of formality.

- A **write operation** starts when the client **initiates the operation** and **ends when all correct servers in some quorum have processed the update.**
- A **read operation** starts when the client **iniates the operation** and **ends when the client resolve the latest version and value using Result().**

And **op1 preeceds op2** if op1 ends before the start of op2.

op1 and p2 are concurrent if neither op1 preeceds op2, nor op2 preceeds op1.

Byzantine masking quorums

The *masking quorum system* is the one that can be used to mask the faulty behaviors of data in repositories.

Formality

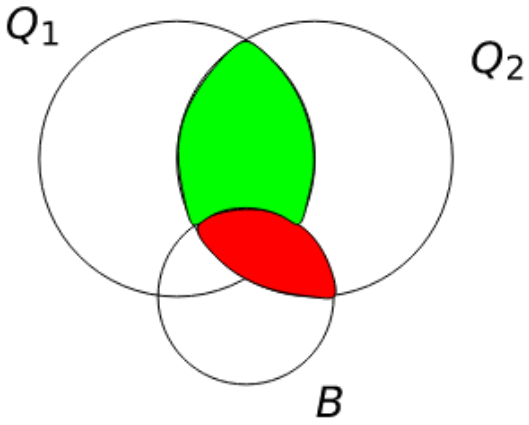
We have:

- Q_1 is the latest write quorum
- B set of byzantine nodes
- Q_2 read quorum

1) $(Q_1 \cap Q_2) \setminus B$ are the servers with **correct up-to-date values**

2) $Q_2 \setminus (Q_1 \cup B)$ are the write servers with **correct out-of-date values**

3) $Q_2 \cap B$ are the **values written wrongly**



To the client **read the correct value** he must be able to identify the most up-to-date value/timestamp pair as one returned by a set of servers that could all be faulty. As we have seem in Byzantine Failure section, the number of non-faulty servers must be the majority. This leads us to the following definitions:

- **M-consistency:**

$$\forall Q_1, Q_2 \in \Theta; \forall B_1, B_2 \in \beta : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$$

Translation: it basically says that **for the non-faulty servers of writing, there must have at least one non-faulty server at read** (Remember: the client is non-faulty, thus he does not read wrongly. If the servers are already non-faulty at write, are non-faulty at read).

⇒ The question is, how the client can the client know what are the up-to-date values?

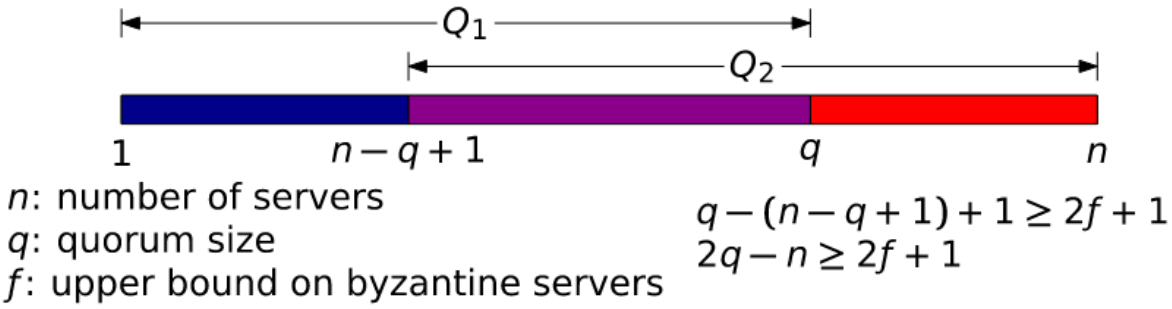
There's a bound of maximum servers can failure. Let's call if f.

> This is because, by research, the probability of more than f servers having failures becomes really low. For example, imagine that a probability of a server failing is 0.01. The probability of n servers have failure is 0.01^n.

In this scenario, the **M-consistency** says that we need at least $f + 1$ up-to-date non-faulty servers **(the majority!)**.

Thus in the end, the intersection must be at least $2f + 1$.

We can put this measure in function of the size of the write quorum and the number of servers.



- **M-availability:**

$$\forall B \in \beta : \exists Q \in \Theta : B \cap Q = \emptyset$$

Translation: We must have at least one quorum that's not faulty.

When a transaction for a quorum fails the client will try to contact another quorum to proceed with the operation. At some moment the client will have success, because there's at least one quorum that does not contain faulty servers.

Assuming that all servers are peers, we have that this equation says:

$$n - f \geq q$$

We must have enough peers to create a quorum with size q.

$$2q - n \geq 2f + 1 \rightarrow$$
$$2(n - f) - n \geq 2f + 1 \rightarrow$$
$$n \geq 4f + q$$

$$\texttt{let } n = 4f + 1 \rightarrow$$
$$q = 3f + 1$$

The conclusion

Basically, increasing the number of servers in the system worsens the performance, since it requires a larger number of quorums. The availability also get's worse. But this increases the fault tolerance.

Just to proving what we have seen:

▼

$w \geq f + 1$ (1) this ensures that writes survive failures
$w + r > n$ (2) this ensures that reads see most recent write
$n - f \geq r$ (3) this ensures read availability
$n - f \geq w$ (4) this ensures write availability

- $2n - 2f \geq r + w$ from (3) and (4)
- $2n - 2f > n$ from (2)
- $n > 2f$
- Let:
  $n = 2f + 1$
  Then: $w = f + 1$, from (1) and (4)
  Then: $r = f + 1$, from (2) and (3)

Read operation for byzantine quorums based on size (suspeito)

We have seen how the read operation works. Now let's update it to what we have seem.

Now to the client write x in a quorum we must do the following:

- c queries the servers until it has a response from a set Q of $3f + 1$ different servers, where $A = \{(v_u, t_u) : u \in Q\}$ be the **set** of value/timestamp pairs received from at least $f + 1$ servers.
- As we have seem before, the result of the value is returned by $Result(A)$. This will return the highest timestamp in the set A or the $\perp$ if no such value.

Safety - with concurrent write

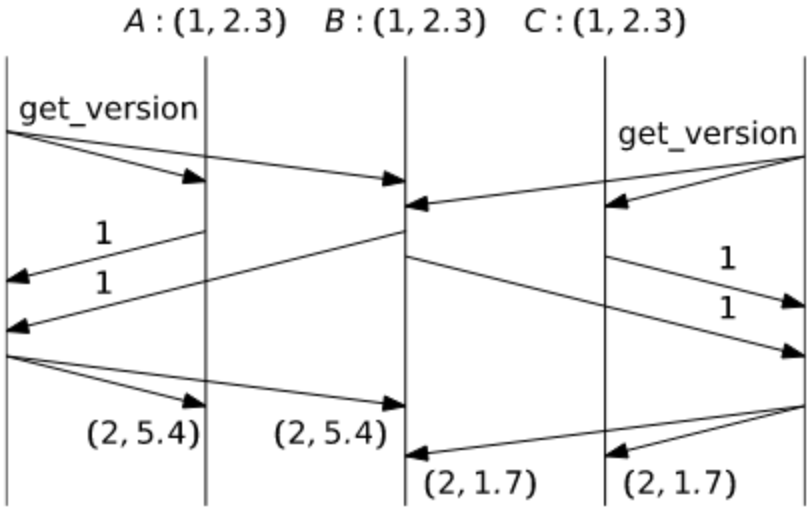
The begging of a write operation **may** happen at the same of another write operation.

The read operation is not concurrent to any write.

We mus ensure that after all the write operations that have happening concurrent, the correct server must have applied the (v, t_{max}) with the highest timestamp to their local copy.

The thing is that **timestamps are totally ordered and every write operation** has **a different timestamp**. Since (v, t_{max}) is always received by a quorum of servers, by the write protocol, the client can always send the correct timestamp and this will be applied by the correct servers.

We have seem that applying concurrent writes using number of versions is naive:



The write protocol uses a **never give up** policy, which ensures delivery of sent messages, once the system becomes synchronous and communication problems fixed.