2 - Context

2.1 ALGORITHM OVERVIEW

Today all the leader-based state machine replications protocols use as baseline the PBFT, which is the first BFT protocol that offers realistic utilization in practice. This is the main reason why it has been chosen for comparison with Prime. But before we describe the main building blocks of PBFT algorithm we will explain shortly previous work on byzantine fault tolerance and state machine replication. Let's start by presenting the main mechanisms used in PBFT and then dive deep into details for each one of them and describe the impact that they have on the overall architecture of the algorithm. Doing so we will create a better idea of how PBFT is built up. PBFT has incorporated the following mechanisms:

1. Event ordering
2. 3-phase protocol
3. View change protocol
4. Garbage collection
5. State transfer
6. Proactive recovery

2.1.1 Event ordering

Event ordering in a distributed system was first introduced by Lamport in the paper 'Time, Clocks, and the Ordering of Events in a Distributed System'. He was the first to solve the problem of clock synchronization on distributed systems, which plays a critical part in building distributed system protocols. It guarantees that the state will be replicated correctly across the machines(processes) by the simple fact that these machines will execute all the events

in the same order and so if every machine first was on State (0) eventually all machines will be on the same State(n) after executing “n” events.

We, as humans, describe the relationship between two events using physical time. So, if event "A” happened at 15:39 and "B” at 15:40 we say that event A happened before (earlier than) B. Perfectly synchronization of real clocks across all the machines is hard problem and maybe impossible, so Lamport takes a different road on ordering events. Rather than using continuous tick clocks he uses discrete tick clocks. Before describing further this concept let's give a clear view of the system.

Our system is composed of a set of machines (S= {M1, M2..., Mi}) each of them having a set of events (EM1= {E1, E2, E3}, EM2 = {E2, E4} …, EM3= {Ei, ..., Ej} where i<j), totally ordered. A message sent or received from a machine, or a local execution step is considered to be an event.

Discrete ticks or Logical clocks are a way to timestamp each event. A local counter can be used by each machine to generate timestamp(t). First every machine starts with timestamp t=0, then:

- On event occurring at the local machine increment timestamp t=t+1;
- On request to send a message(m) increment timestamp t=t+1 and send (m, t);
- On receiving (m, t’) t = max (t, t’) +1;

These logical clocks, also called Lamport clocks have the nice property:

If event “a” happens before event “b” then timestamp of event “a” is less than timestamp of event “b”. However, if timestamp of event “a” is less than timestamp of event “b” doesn’t not imply that event “a” happens before “b”. Also, nothing prevents having two events with the same timestamp. But if we assume that each machine has a unique identifier that is comparable, and we combine the timestamp of an event with the machine identifier in which that event occurred

then we will uniquely identify this particular event. So, using Lamport clocks we can define a total order of events:

(a=>b) <=> (T(a)<T(b) or (T(a)<T(b) and M(a)<M(b)))

The Lamport clocks still don't detect the concurrent events. We need vector clocks to do it. So rather than mixing all the machines together on a single counter we will actually have a separate counter for each machine that holds the number of events that have occurred on the specific machine. Below are the steps of algorithms for incrementing the vector of timestamp T:

- On event occurring at the local machine "i" increment timestamp T[i] = T[i]+1;
- On request to send a message(m) increment timestamp t=t+1 and send (m, t);
- On receiving (m, t') t = max (t, t') +1;

Total order broadcast or atomic broadcast is extension of event ordering. It guarantees:

- reliability of messages, all messages delivered to all machines,
- total order, all machines execute messages in the same order.

Designing an algorithm for atomic broadcasts is relatively easy if it can be assumed that computers will not fail. For example, if there are no failures, atomic broadcast can be achieved simply by having all participants communicate with one "leader" which determines the order of the messages, with the other participants following the leader.

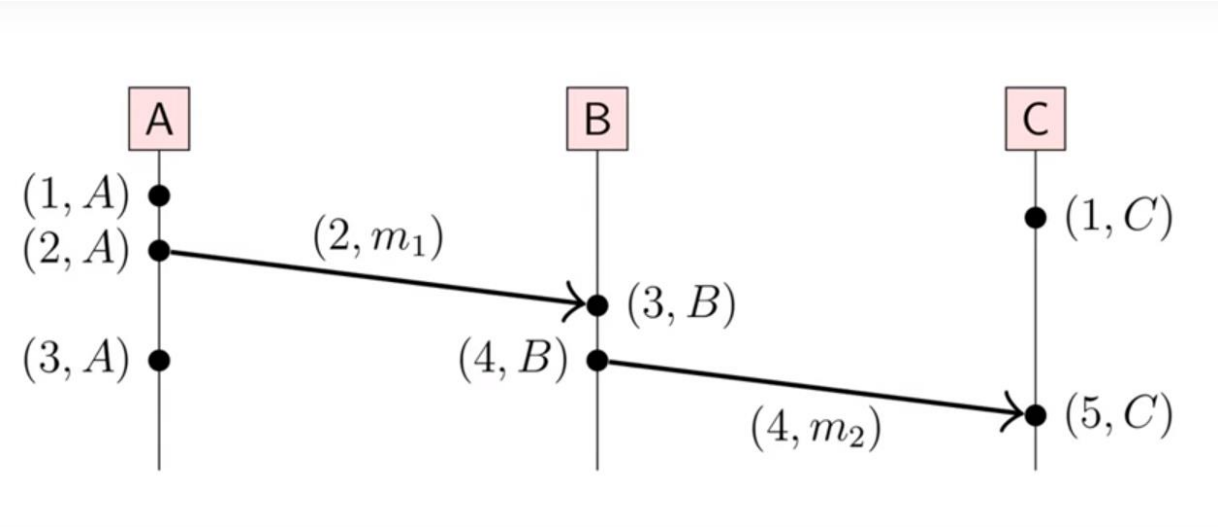

Figure 1: Lamport clocks example

2.1.2 Byzantine Consensus

Traditionally consensus is formulated: several machines want to come to agreement about a single value. Once one node decides on a certain value, all other nodes will decide the same value. Consensus and atomic broadcast are formally equivalent.

Viewstamped Replication [Liskov 1988] solved the problem of consensus in the assumption of only benign faults. This algorithm used the mechanism of view, where a view refers to the configuration of the system in which one of machines is leader. The views represent consecutive integers. Each view begins with an election in which one or more candidates attempt to become leader. If a candidate wins the election, then it serves as leader for the rest of the view. The leader decides the ordering for execution of client operations. It does this by assigning the next available sequence number to an operation and sending this assignment to all other machines. But the leader may be faulty. Therefore, replicated machines verify the sequence numbers assigned by the leader and use timeouts to detect when it stops. They trigger view changes to

select a new leader. View-change is used when the current leader fails to deliver. The view change protocol provides liveness by allowing the system to make progress when the primary fails. Client operations are synchronized using strict 2-phase locking.

Figure 2: view-change, or leader-election

PBFT uses a 3-phase protocol to atomically broadcast client operations. This protocol takes five rounds of communication. In the first round the client sends the operation op [timestamp, client-id, signature] to the leader. In the second round the leader assigns a unique sequence number to the operation, and multicasts the pre-prepare message m [op, seq-number, view-number] to all replicas. A replica accepts a pre-prepare message if:

- It is in the same view with leader.
- It has not yet accepted a pre-prepare with same view and same sequence number.
- The sequence number is between a low-high water mark.

In the third round, replica accepts a pre-prepare message and enters the prepare phase. Then it broadcasts to all replicas a prepare message. A replica accepts a prepare message if:

- It is in the same view with leader.
- The sequence number is between a low-high water mark.

The fourth round starts when the replica has 1 accepted pre-prepare and 2f accepted prepare messages from different replicas. This replica broadcasts a commit message to all other replicas. A replica accepts a commit message if:

- It is in the same view with leader.
- The sequence number is between a low-high water mark.

The response is the fifth round where one of replicas after accepted 2f+1 commits message performs the operation and sent reply to the client.

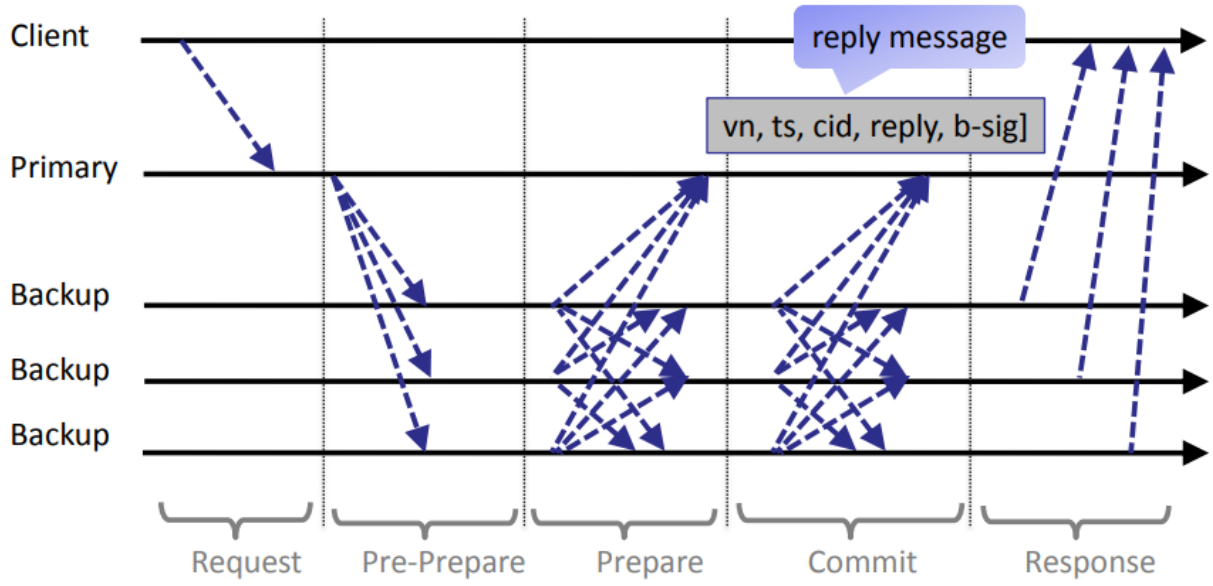

Figure 3: 3-phase protocol

2.1.3 Garbage collection

Since machines log every message eventually there will come a time that the size of logs could grow very large and cause an out of memory error. To prevent this effect the PBFT introduced garbage collection and checkpoint mechanisms. A checkpoint is created every time a machine receives a request with a sequence number multiplied by K, for example 128. After applying checkpoint to its current state, the machine multicasts a message <checkpoint, sequence number, digest, machine id>. On receiving 2f other matching checkpoint messages, the machine logs this checkpoint as stable checkpoint and discards all the messages that preceded the sequence number.

Previously we mentioned that each replica maintains low and a high-water mark to define the range of sequence numbers that may be accepted. The low water mark is equal to the sequence number of the last stable checkpoint and the high-water mark is H = h + L, where L is often set

to 2K. The high-water mark H is big enough to prevent machines from stalling waiting for a checkpoint to become stable.

2.1.4 Proactive recovery

Proactive recovery is essential for providing security in the presence of byzantine failures. It narrows down the window of vulnerability to maybe just a few minutes during normal operation. The recovery happens periodically, and it's not triggered by any failure detection mechanism. The recovery protocol makes faulty replicas behave correctly again to allow the system to tolerate more than f faults over its lifetime. The recovering replica starts by discarding the keys it shares with clients and it multicasts a new-key message to change the keys it uses to authenticate messages sent by the other replicas. Next, runs an estimation protocol to calculate an upper bound, Hm, on the high-water mark. After this point participates in the protocol as if it were not recovering but it will not send any messages above Hm until it has a correct stable checkpoint with sequence number greater than or equal to Hm. Next it sends a recovery request, which is treated as a normal request, it is assigned a sequence number and goes through all three phases. When other replicas process the recovery request, they send back new keys. The recovery replica waits for 2f+1 replies, and then computes the recovery point and a valid view. While it is recovering, the machine uses the state transfer mechanism to fetch out-of-date or corrupted data.