Tech report

Vahid HeidariPour Lakhani

December 27, 2018

1 Consensus(Computer Science)

A fundamental problem in distributed computing and multi-agent system is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation. Examples of applications of consensus include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts. The real world application include clock synchronization, page rank, opinion formation, smart power grids, state estimation, control of UAVs, load balancing and others.

2 Problem Description

The consensus problem requires agreement among a number of processes(or agents) for a single data value. Some of the processes may fail of be unreliable in other ways, so consensus protocols must be fault tolerant or resilient. The processes must somehow put forth their candidate values, communicate with one another, and agree on a single consensus value. The consensus problem is a fundamental problem in control of multi-agent systems. One approach to generality consensus is for all processes(agents) to agree on a majority value. In this content, a majority requires at least one more than half of available votes (where each process is given a vote) however one or more faulty processes may skew the resultant outcome such that consensus may not be reached or reached incorrectly. Protocols that solve consensus problem are designed to deal with limited numbers of faulty processes. These protocols must satisfy a number of requirements to be useful. For instance a trivial protocol could have all processes binary value 1. This is not useful and thus the requirement is modified such that the output must somehow depend on the input. This is, the output value of a consensus protocol must be the input value of some process. Another requirement is that a process may decide upon and output a value only once and this decision is irrevocable. A process, is called correct in an execution if it does not experience a failure. A consensus protocol tolerating halting failures must satisfy the following properties:

- Termination - Every correct process decides some value

- Validity - If all processes propose the same value v, then all correct processes decide v.
- Integrity - Every correct process decides at most one value, and if it decides some value v, then v must have been proposed by some process.
- Agreement - Every correct process must agree on the same value.

3 The Paxos Algorithm

Consensus is a fundamental problem in distributed computing. Requirements of consensus problem are:

- Termination
- Agreement
- Validity

Paxos is an algorithm for implementing fault tolerant consensus. It runs a completely connected network of n processes and tolerates up to m failures (where n ¿ 2m + 1). Algorithm solves the consensus problem in the presence of failures on an asynchronous system of processes. Paxos guarantees primarily agreement and validity and not termination. There is three roles of processes:

- Proposer
- Acceptor
- Learner

Suppose that we have 3 processes, that 2 of them are proposers and one acceptor. This is OK until the acceptor fails, then what happens? It's obvious that we need other acceptors to go forward. There is three main phases in algorithm. There is a solution for single acceptor, and it's quorum, mostly set of odd number of acceptors.

3.1 Split Votes

However making the quorum approach is a little tricky, for example, suppose we decide that each acceptor accepts all the first value it receives. And then whichever gets the majority wins. Well, as we can see in Figure 1,there might be a situation where nobody might get a majority. In this five node cluster we don't have any one value that been agreed two by three of five servers. This means that acceptors sometimes have to change their mind, after accepting one value and acceptor will occasionally have to accept different value. Another way of saying is this, that there is noway to guarantee that you can achieve an agreement in a cluster in a single round. It may take multiple rounds in order to get to agreement.

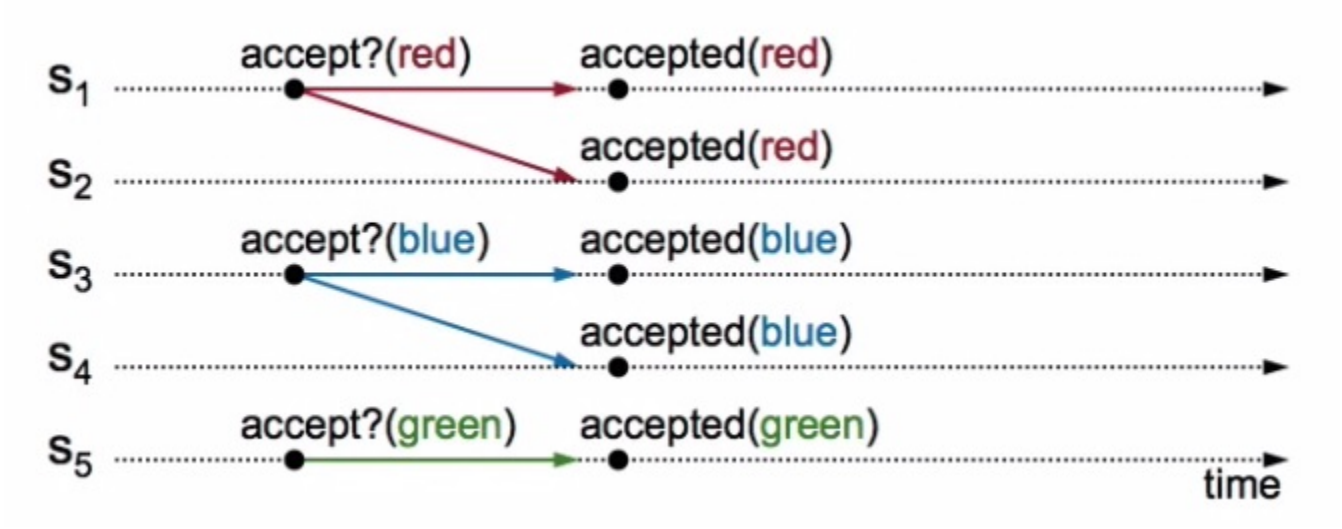

Figure 1: Split Votes

3.2 Conflicting Choices

The second example is where every acceptor accepts every value it receives and could choose multiple values. This one has two major problems, the first problem is that we can end up choosing multiple values, for example server one might propose the value red, ask other servers to accept it, and so server one, two, and three accept that value red, now that value Red chosen, since it been accepted by a majority of nodes of cluster. But then, server 5 might come along, propose a different value blue, and ask acceptor accept that value, and since they accept every value they receive, server three will now accepts blue even though it previously accepted red, and so now we've also chosen the value blue, and this violates the fundamental safety property that we have to choose only one value. The solution to this problem is that, if a second proposer comes along once a value has already been chosen, it has to abandon any value it's own and has to propose the existing chosen value. So in this scheme this means before server five can ask for acceptance of its value first has to look around and see if there is any value out there, if some other value chosen, then server five has to abandon its own value and instead use the value red, and then we'll end up with red chosen and so on.

Unfortunately, this mechanism is not enough by itself, and the figure 3 shows the example. Suppose server one comes along and it's going to propose value red. It first checks with other servers and sees no other value has been accepted anywhere, so it starts asking servers to accept its value red, but meanwhile, before any of those acceptors actually respond, another server comes along and decides to propose value blue and similarly it looks ans sees no other value is chosen out there, and so it starts sending messages to get the blue value chosen, and in this particular scheme, it happens that blue finish first, it gets server three, four, and five to accept that blue value, and so blue is now chosen, but meanwhile, the red first server continues to work and eventually since the acceptors will accept multiple values, we can see

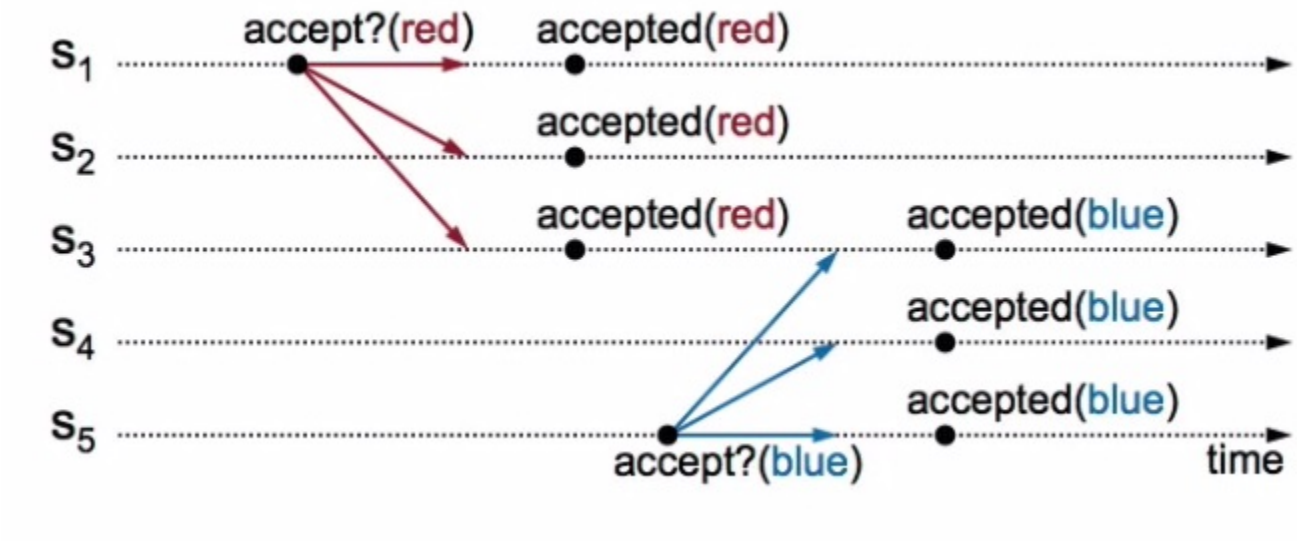

Figure 2: Conflicting Choices

it eventually gets a quorum have and the red value ends up been chosen also. Once again, we have violated our fundamental safety property. The solution to this problem is that once we've chosen one value, any competing proposals have to somehow be aborted, that is in this case, we somehow need for server three to reject that red acceptance request after its already accepted blue, and the way we'll do this we're going to place an order on proposals where newer proposals take place over old proposal, so once the blue proposal which is later has now got there, it will cut off the red proposal. So that proposal can not finish choosing a competing value.

3.3 Phase 1 (Preparing Phase):

3.3.1 .Step 1.1.

Each proposer sends a proposal (v, n) to each acceptor.

3.3.2 .Step 1.2.

If n is the largest sequence number of a proposal received by an acceptor, then it sends an ack(n, ?, ?) to its proposer, which is a promise that it will ignore all proposals numbered lower than n. However, In case an acceptor has already accepted a proposal with a sequence number n'<n and a proposed value v, it responds with an ack(n, v, n'). This implies that the proposer has no point trying to submit another proposal with largest sequence number. When there are two proposals with identical sequence numbers, the tie is broken using the process ids.

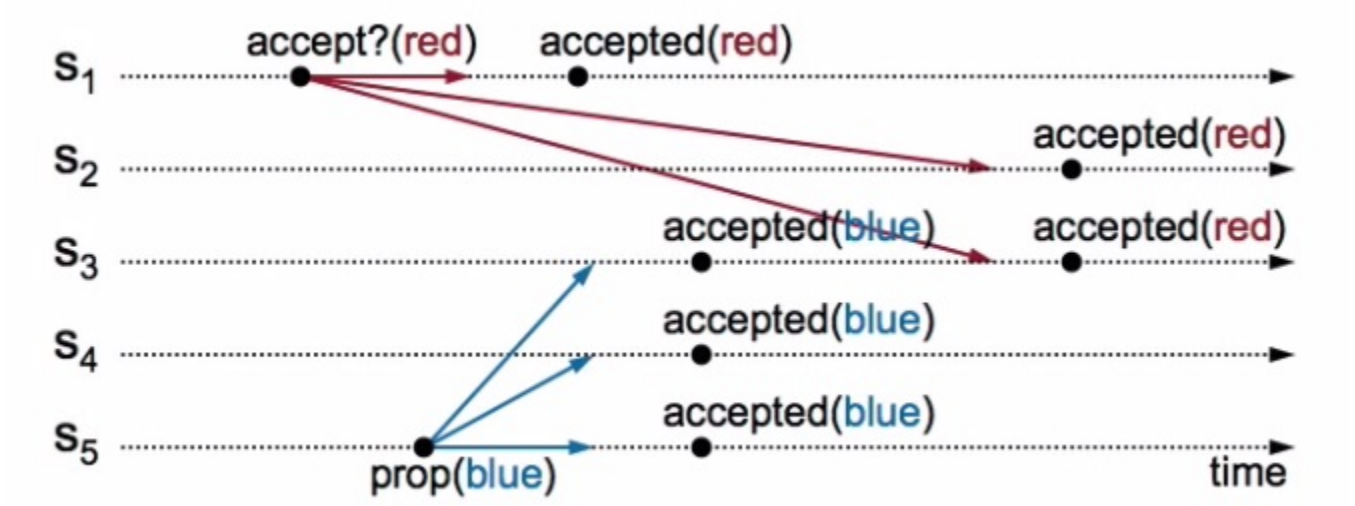

Figure 3: Conflicting Choices cont'd

3.4 Phase 2 (Accept):

3.4.1 .Step 2.1.

If a proposer receives an ack(n, ?, ?) from a majority of acceptors, then it sends accept (v, n) to all acceptors, asking them to accept this value. If however, an acceptor returned an ack(n, v, n') to the proposer in phase 1 (which means that it already accepted proposal with value v) then the proposer must include the value v with the highest sequence number in its request to the acceptors.

3.4.2 .Step 2.2.

An acceptor accepts a proposal (v, n) unless it has already promised to consider proposals with a sequence number greater than n.

3.5 Phase 3 (Final Decision):

When a majority of the acceptors accepts a proposed value, it becomes the final decision value. The acceptors multi-cast the accepted value to the learners. It enables them to determine if a proposal has been accepted by a majority of acceptors. The learner covey it to the client processes.

3.6 Observations

- An acceptor accepts a proposal with a sequence number n if it has not sent a promise to any proposal with a sequence number n'>n.

- If a proposer sends an accept (v, n) message in Phase 2, then either no acceptor in a majority has accepted a proposal with a sequence number n' <n, or v is the value in the highest numbered proposal among all accepted proposals with sequence numbers n' <n accepted by at least once acceptor in a majority of them.

3.7 Properties

There is two type of properties that should considered.

3.7.1 Safety Properties:

- Validity - Only a proposed value can be chosen as the final decision.
- Agreement - Two different processes cannot make different decisions.

3.7.2 Liveness Properties

Some proposed value is eventually chosen and if a value has been chosen, then a process can eventually learn the value.

3.8 Consider the following scenario:

- Phase 1 - Proposer 1 sends out prepare (v, n1);
- Phase 1 - Proposer 2 sends out prepare (v, n2), where n2 >n1;
- Phase 2 - Proposer 1's accept (n1) is declined, since the acceptor has already promised to proposer 2 that it will not accept any proposal numbered lower than n2. So proposer 1 restarts phase 1 with a higher number n3 >n2;
- Phase 2 - Proposer 2's accept request is now declined on a similar ground; *The race can go on forever! There are two solution for this problem, the first one, randomized delay before restarting, to give other proposers a chance to finish choosing. And the second one, is using leader election. Using basic Paxos is inefficient, with multiple concurrent proposers, conflicts and restarts are likely (higher load = more conflicts). In leader election, at any given time, only one server acts as proposer. For leader election, there is one simple approach from Lamport, Let the server with highest ID acts as leader. Each server sends a heartbeat to every other server every T ms, if a server hasn't received heartbeat from server with higher ID in last 2T ms, it acts as leader, and accepts request from clients, and acts as proposer and accetor. If server not leader, rejects client requests (redirect to leader) and acts only as acceptor.

3.9 Why is it important?

Many problems in distributed systems are equivalent (or harder than) consensus!

- Perfect failure detection
- Leader election (select exactly one leader, and every alive process knows about it)
- Agreement (harder than consensus)

So consensus is a very important problem, and solving it would be really useful.