Distributed Agreement

Javier Palomares
Cockrell School of Engineering
The University of Texas at Austin
Austin, TX
javierp@utexas.edu

Matt Molter
Cockrell School of Engineering
The University of Texas at Austin
Austin, TX
mmolter@utexas.edu

***Abstract*—This paper covers the topic of Distributed Agreement. Specifically, the Paxos algorithm by Lamport [1], a Byzantized version of Paxos [2], and a weighted non-Byzantine and Byzantized Paxos incorporating ideas from Garg [3].**

I. Introduction

This paper serves as a recap of Paxos, how to incorporate weights into Paxos, how to Byzantize Paxos, and how to incorporate weights into the Byzantine Paxos algorithm. In section II, we recap the Paxos algorithm as presented by Lamport in [1] for those unfamiliar with the algorithm. Those familiar can skip this section. In section III, we introduce weighted Paxos. In section IV, we cover how we Byzantized Paxos per Lamport's recommendations in [2]. In section V, we introduce the modifications to Byzantine Paxos in order to make it weighted as in the algorithms presented by Garg in [3]. In section VI, we describe our results. In section VII, we present conclusions.

II. Introduction to Paxos

This section serves as an introduction to Paxos for those unfamiliar with the algorithm. For those familiar, this section can be skipped.

Paxos as described by Lamport is a distributed, non-Byzantine fault tolerant consensus algorithm. The model assumed is therefore an asynchronous, non-Byzantine model.

You begin with a set of processes which can propose values, and a single value amongst the proposed must be chosen. If no value is proposed, no value is chosen, and if a value is chosen, all processes can learn of the selection. This leads to the below safety requirements per Lamport:

1) Only a value that has been proposed may be chosen
2) Only a single value is chosen
3) A process never learns that a value has been chosen unless it actually has been.

Lamport does not specify liveness requirements other than that some value proposed is eventually chosen and that if a value is chosen, a process can eventually learn of the selection. We have chosen to omit the learning part of this algorithm for simplicity. It should not affect the correctness of the first two phases, propose and accept.

We will not delve into the derivation of the algorithm, and will be operating off of a modified version of Paxos which Lamport refers to as $PCon$ in [3]. As in vanilla Paxos, there are three classes of agents: $proposers$, $acceptors$, and $learners$. The actions they perform should be self-explanatory. Paxos performs numbered ballots, each of which is orchestrated by a leader (the proposer). If N is the number of acceptors (which can include the proposer itself), where $N > f$, the number of failed servers, a quorum is any $N-f$ acceptors. A simple way to require that any two quorums have a non-empty intersection (required for safety), we require that $N > 2f$. If a quorum of acceptors vote for a value, then that value is considered chosen.

In the language of the algorithm, a proposer can make a proposal ballot number b (proposal number n in [1]) and value v. The proposal number is used to determine which proposal an acceptor will accept, and the value is the value which will be accepted. There are two-phases to the commit, $prepare$ and $accept$.

The below are the properties we are required to maintain per $PCon$. Those familiar with the vanilla Paxos algorithm will notice several small differences.

P1. An acceptor can vote for a value v in ballot b only if v is safe at b

P2. Different acceptors cannot vote for different values in the same ballot.

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b

P3b. If some acceptor in the quorum has voted, let c be the highest-numbered ballot less than b in which such a vote was cast. The value voted for in ballot c is safe at b. (By P2, there is only one such value.)

The vanilla Paxos algorithm is as below.

Phase $1a$ The ballot-b leader sends a $1a$ message to the acceptors.

Phase $1b$ An acceptor responds to the leader's ballot-b $1a$ message with a $1b$ message containing the number of the highest-numbered ballot in which it has voted and the value it voted for in that ballot, or saying that it has cast no votes.

Phase $2a$ Using the $1b$ messages sent by a quorum of acceptors, the leader chooses a value v that is safe at b and sends a $2a$ message containing v to the acceptors.

Phase $2b$ Upon receipt of the leader's ballot-b $2a$ message, an acceptor votes for v in ballot b by sending a $2b$ message.

The modifications required to create $PCon$ are below. We require the addition of a $1c$ action as well as a modification to phase $2a$.

Phase $1c$ Using the $1b$ messages from a quorum of acceptors, the leader chooses a set of values that are safe at b and sends a $1c$ message for each of those values.

Phase $2a$ The leader sends a $2a$ message for some value for which it has sent a $1c$ message.

These modifications result in the splitting up of the old phase $2a$ message into two parts. What does this accomplish? There are two items; 1) it allows for multiple safe values and 2) it is important in reconfiguration by removing dependence on acceptors in lower level ballots. This informs future leaders of safe values so that they do not have to learn about votes cast in previous ballots before participating in the algorithm.

In order to choose a value to send in the $1c$ message, it must satisfy P3a above, in addition to a new P3c below.

P3c If a ballot-c message with value v has been sent, for some $c < b$, and (i) no acceptor in the quorum has voted in any ballot greater than c and less than b, and (ii) any acceptor in the quorum that has voted in ballot c voted for v in that ballot, then v is safe at b.

III. Weighted Paxos

Before we delve into adding weights to Paxos, we'll begin by introducing the motivation behind adding weights. In a typical Paxos system, equal weights are given to all nodes. This can be viewed as all nodes are seen as equally trustworthy by all other nodes. This results in the classic result in which the total number of nodes N must be $> 2f+1$ or $f < N/2$. This can be improved upon by assigning different weights (levels of trust) to different nodes and modifying the algorithm as will be discussed below.

To begin with, we have an abstract notion of trust, which we represent with weights, $w[i]$ for each node P_i, where $0 \leq w[i] \leq 1$. The weights are normalized such that the sum of all weights adds up to 1. Each node has its own level of trust which is known across the system, the assignment of which is beyond the scope of this paper. Suffice it to say that a higher weight assigned to a node means that we have a higher level of trust in that node and vice versa. We now wish to tolerate a weight of failed nodes of at most ρ.

In the case of Paxos, stating that $\rho < 0.5$ is equivalent to a generalized version of $f < N/2$ in which we normalize for the number of nodes N. This means that we require a quorum with combined weight of > 0.5 to accept in order to achieve consensus. Now that we have introduced the concept of weighted agreement, let's use an example to see how this can be used to improve Paxos.

First, visualize a system with 4 processes, P_1, P_2, P_3, P_4. In traditional Paxos, all four would have equal weight of $0.25, 0.25, 0.25, 0.25$, and if any two processes crash, agreement becomes impossible. Now, assume that the four processes respectively have weights $0.3, 0.3, 0.2, 0, 2$, representing higher level of trust in P_1 and P_2. For a quorum to be achieved, it we simply need a weight of > 0.5. The careful reader can see that in this case, if P_3 and P_4 crash, agreement is still possible, meaning we can tolerate failures $\geq N/2$ nodes.

This does not mean that any process failure can be tolerated in the above example. The failure of various other combinations still result in failure to achieve agreement. For example, if P_1 and P_3 were to fail, agreement remains impossible. However, the result remains very powerful, particularly in large systems. For example, you could spend large amounts of money to configure several nodes to be extremely fault tolerant, and then have a larger number of cheaper, commodity machines. Assigning higher weights to the more fault tolerant and lower weights to the commodity machines, you could lower your cost to achieve a similar level of fault tolerance.

In terms of the required properties, the quorum is replaced with a weighted quorum. No further modifications are necessary. In terms of the actions vs the vanila Paxos algorithm, again, the quorum is replaced with a weighted quorum and no further modifications are necessary.

Practically, implementing the weighted Paxos is simple if you already have a functioning Paxos algorithm. Each server already keeps an identical list of peers for communication. We simply added an attribute for $weight$ to our $Peer$ object which previously stored $ServerID$, $IPAddress$, and $Port$. On receiving a response from an acceptor, instead of incrementing a counter by 1, we added the $weight$ from the responding $Peer$ to our counter. Once our counter reaches weight > 0.5, we have achieved quorum and Paxos can move on. Similar behavior is required for both propose and accept, and the modification is the same.

IV. Byzantizing Paxos

Byzantizing Paxos requires several modifications. Let us first deal with Byzantine acceptors. The naive method recognizes that a simple majority quorum is no longer satisfactory in order to guarantee a quorum of real acceptors. Let us first examine the case in which $f < N/4$. Instead of a quorum, we require what Lamport refers to as a $Byzquorum$ in order to emulate a proper quorum. To define a Byzquorum, imagine that you have a set of N processes which would normally require a quorum q in order to choose a value. However, you are operating under the assumption that you now have f $byzacceptors$ which may act in a malicious manner. To guarantee that you still have a valid quorum, you must now have a Byzquorum of $q + f$ byzacceptors, which in the case of $f < N/4$, means you require $3f + 1$ acceptors to reach a proper quorum. This satisfies P3a above, but P3b remains an issue. This is because there is no method of determining if a single message is from a real or Byzantine acceptor. We get around this by assuming that $N > 3f$, which means that any two quorums have at least $f + 1$ acceptors in common. This leads to the below modifications to P3a and P3b.

P3a'. If there is no ballot numbered less than b in which $f + 1$ acceptors have voted, then all values are safe at b.

P3b'. If there is some ballot c in which acceptors have voted and there is no higher-numbered ballot less than b in which $f + 1$ acceptors have voted, then the value v voted for in c is safe at b

These modifications follow naturally from the distrust of f processes. To guarantee that at least one acceptor which is genuine, we require the expected faulty number of processes f in addition to the previously required single true acceptor. Since a Byzantine process can also act like a real process, this guarantees that we can tolerate at least f faults. The problem with this method and vanilla Paxos is that it leads to a solution where we must have $> 4f$ acceptors. Lamport gets around this using $PCon$ in addition to other modifications involving digital signatures. We did not implement this due to lack of time and believing the digital signature portion was beyond the scope of this course, so our solution requires $> 4f$ acceptors.

Also notice that this solution so far accounts only for malicious acceptors, and not a malicious leader. A malicious leader could send multiple $2a$ messages for different values to different acceptors, leading to the choice of multiple values in future ballots. The solution is to eliminate the $2a$ action and replace it with a $2av$ action.

In the $2av$ action, when a leader sends a $2a$ message, each acceptor that receives it then emulates the execution of the $2a$ action. To do this, it sends the value to all other acceptors in order to confirm that the same value was sent to all acceptors. Once an acceptor receives a byzquorum of $3f + 1$ $2av$ messages, then it knows that the value was sent to a quorum of acceptors and can proceed with the $2b$ action responding to the leader.

To derive a general Byzantine Paxos, we add f fake acceptors to $PCon$ in order to create $BPCon$. As stated above, the $2a$ action is removed. Instead, a leader requests to its acceptors that they emulate a $2a$ action, and they perform the $2av$ action as above to provide a $2b$ response. The acceptor is allowed to perform a $2av$ action iff it has received the corresponding $1c$ message and has not already performed a $2av$ action.

In order for the leader to send a $1c$ message, the acceptors must know that the message is legitimate. In order to determine this, they perform a $2av$ action when performing the $1b$ action as well to verify that they all received the same value and that they can consider that value as safe when told by the leader.

Now, with the information sent to the leader by acceptors in the $1b$ message, along with the properties it must maintain, the leader can determine safe values and inform the acceptors of these safe values. Again, the acceptors broadcast to all other values and wait for ack in order to determine that a malicious leader is not sending different values to different acceptors.

V. Weighted Byzantized Paxos

In order to create our final Weighted Byzantine Paxos, the modifications to Byzantized Paxos consist primarily of changing the requirements for a byzquorum. Previously, in order to guarantee a proper quorum when $f < N/4$, the required number of nodes for a byzquorum was $4f+1$, subbing in $f < N/4$, the number of nodes in a quorum increases to $N/2 + 1 + N/4$, or $3N/4 + 1$. When converting to weights for a $\rho < 1/3$, this translates to a weight of at least $3/4$ to achieve a byzquorum.

Now that we have introduced the concept of weighted Byzantine agreement, let's use an example to see how this can be used to improve Byzantized Paxos as we did with vanilla Paxos.

First, visualize a system with 3 processes, P_1, P_2, P_3. In traditional Byzantine Paxos, all three would have equal weight, and if any single process were to lie, agreement becomes impossible. Now, imagine the three processes respectively have weights $0.5, 0.3, 0.2$, representing levels of trust. For a byzquorum to be achieved, the weights of real acceptors need to sum to greater than $1 - \rho$, or $3/4$. It can be seen that if the liar is P_3, this has created a case where 1/3 of processes can be malicious, yet agreement can still be achieved.

This does not mean that any process failure can be tolerated in the above example. The failure of P_1 or P_2 still results in an unsolvable system. However, the result remains very powerful, particularly in dynamic systems. For example, by assigning long running processes a higher weight, and newly joined processes lower weight, you can remain Byzantine fault tolerant even in the face of a large number of malicious processes joining your system.

VI. Final Algorithm

The failure model assumed is asynchronous, with malicious leaders and acceptors possible. One restriction we placed on our model for simplicity is that no process can pretend to be another process, eliminating the need for message authentication; we believe this is beyond the scope of this course. The result as previously mentioned allows for only $f < N/4$ fake acceptors.

The basic algorithm is very similar to that in $BPCon$ as proposed by Lamport, but with generalized weights instead of the assumption that all processes have equal weight. We will refer to it from here on as $WBPCon$ for Weighted Byzantine Paxos consensus. The required properties maintained are below.

WBP1. An acceptor can vote for a value v in ballot b only if v is safe at b

WBP2. Different acceptors cannot vote for different values in the same ballot.

WBP3a. Each message in S asserts that its sender has not voted.

WBP3b. If there is some ballot c in which acceptors have voted and there is no higher-numbered ballot less than b in which acceptors with combined weight $\geq \rho$

have voted, then the value v voted for in c is safe at b

WBP3c. For some $c < b$ and some value v, (a) each message in S asserts that (i) its sender has not voted in any ballot greater than c and (ii) if it voted in c then that vote was for v, and (b) there are $\geq \rho$ weighted $1b$ messages from byzacceptors saying that they sent a $2av$ message with value v in a ballot $\geq c$.

The actions performed are as below.

Phase $1a$ The ballot-b leader sends a $1a$ message to the acceptors.

Phase $1b$ An acceptor responds to the leader's ballot-b $1a$ message with a $1b$ message containing the number of the highest-numbered ballot in which it has voted and the value it voted for in that ballot, or saying that it has cast no votes.

Phase $1c$ Using the $1b$ messages from a quorum of acceptors with weight $> 1 - \rho$, the leader chooses a set of values that are safe at b and sends a $1c$ message for each of those values.

Phase $2a$ Using the $1c$ messages previously sent, the leader chooses a value v that is safe at b and sends a $2a$ message containing v to the acceptors.

Phase $2av$ Upon receipt of the leader's ballot-b $2a$ message, an acceptor broadcasts its value v to all other acceptors

Phase $2b$ Upon receipt of $> 1 - \rho$ weighted $2av$ messages, an acceptor votes for v in ballot b by sending a $2b$ message.

VII. Results

As previously stated, adding weights to the vanilla Paxos algorithm results in an increase in tolerance for failed nodes. We tested with 4 servers with weights as described in section III above. This means we first tested P_1, P_2, P_3, P_4 with weights $0.25, 0.25, 0.25, 0.25$ respectively. We confirmed that the failure of any two nodes resulted in a failure to reach consensus. We then tested with weights $0.3, 0.3, 0.2, 0.2$ respectively. Bringing down any single node resulted in identical fault tolerance to the base Paxos algorithm and the system was able to reach agreement. Once the base performance was compared to vanilla Paxos, we began bringing down two servers at a time. First, we confirmed that the failure of P_3 and P_4 allowed for us to continue running. Then that any other combination of two servers resulted in identical performance to the base Paxos algorithm, failing to reach agreement. Then, we applied the weights $0.4, 0.2, 0.2, 0.2$. This allowed any combination of 2 processes from P_2, P_3, P_4 to be brought down and consensus can still be reached. Losing just P_1 also allowed us to continue to reach consensus. However, losing P_1 and any other process resulted in failure. Lastly, we assigned a weight to P_1 high enough to reach agreement on its own, simulating a centralized algorithm. In this case, we assigned $1, 0, 0, 0$ respectively, though any combination where $w[1] > 0.5$ is sufficient to model this behavior. The behavior is similar for the Weighted Byzantized Paxos algorithm. As long as the weights are distributed in such a way that the weights of the real acceptors sum to $> 3/4$, there can be an arbitrary number of fake acceptors. A mildly interesting result of Byzantizing Paxos that was not stated explicitly in [2] is that you reduce the tolerance for non-Byzantine faults. This is due to the need to emulate a proper quorum in the face of $N/4$ Byzantine faults. If $> N/4$ processes crash, there is no way to tell if the remaining processes are Byzantine or non-Byzantine. If enough remaining processes are Byzantine, then there is no guarantee of liveness.