The Paxos Algorithm for Replicated State Machines

By: Ronald Macmaster, Gaurav Nagar, Hari Kosuru, Taylor Schmidt

Abstract

The Paxos algorithm solves the consensus problem for replicated state machines. In other words, it guarantees that all participating nodes will agree on a single result. It also allows for nodes that have failed to recover and rejoin the algorithm, a more difficult problem than simple permanent failure of nodes. In this paper we describe our Java implementation of the Paxos algorithm, which we use to implement a fault-tolerant distributed online store system.

Introduction

Replicated state machines represent a fundamental implementation for fault-tolerant distributed computing services. Servers that host such services are not always reliable, so distributed computing services must account for possible server failures (faults). A replicated state machine duplicates computation and state across multiple servers to backup data in the event of a failure.

However, the consensus problem illustrates the primary challenge to implementing a fault-tolerant replicated state machine. Entire server clusters must agree upon committing and aborting operations as well as the order of committed operations. Without the possibility of server faults, a simple centralized leader algorithm easily solves this problem. The classical Paxos algorithm provides a solution to the consensus problem in the presence of server faults. Not only this - it also allows for crashed servers to recover, come back online, and rejoin the algorithm while still maintaining a consistent distributed state.

The following sections describe our specific research project, a replicated state machine for an online e-commerce store inventory. We provide a description of the project, some design alternatives, and some significant findings. Additionally, we provide some supplementary educational material regarding the Paxos algorithm in Appendix A.

Project Description

For our project, we implemented a distributed online e-commerce store inventory with the following specifications. There is a store which has a list of items, with a count for each item. This store is distributed across a number of nodes and each node should have a consistent global state as far as the client is concerned. In order to maintain this consistent global state, we will use the Paxos algorithm in order to come to a consensus among servers about which actions to take on the store. A client has the following (mostly self-explanatory) commands available to it:

```
$ purchase <user-name> <product-name> <quantity>
$ cancel <order-id>
$ search <user-name>
$ list
```

`order-ids` are returned when a client uses the `purchase` command and in the case that a command is not possible, the server returns a message informing the client as such.

The Paxos Algorithm

The Paxos algorithm was first described in a paper written by Leslie Lamport. It contains three main phases, each of which involve coming to a specific quorum of a majority of nodes. As a precondition, we will assume that we have selected a Leader node that is in some sense “in charge” of the algorithm. In the supplementary material, we see this is a bit of a fuzzy

requirement, but it serves to simplify instruction about the algorithm's mechanics. In this paper, I will only list the phases and the statements they contain. It will be a very high level description of the algorithm, so more detail about what each phase actually entails is presented in Appendix A, the supplementary material.

In the first phase, the leader attempts to gain permission from the other nodes (called Acceptors) to propose a value to be accepted. All of the nodes attempt to form a quorum of agreement around this **`prepare`** command. In the second phase, if the Leader successfully achieved permission to propose a value, it requests that the other nodes **`accept`** a value of its choosing (the supplementary material explains how it chooses this value), and waits for another majority quorum of agreement from the other nodes. In the case of our application, values take the form of the commands available to the client, which are listed in the previous section. Then in the final phase, assuming we successfully reached a quorum of acceptance around this value, the Leader sends a command instructing *all* of the nodes to **`learn`** the value that was accepted by a quorum in the previous phase. At this point, all of the nodes have agreed on a client command and can execute that command on their respective replicated states, thus maintaining consistency among all the nodes in the system. There are a couple of other guarantees that this algorithm makes. It allows that any (or all) of the nodes can crash, and the state between nodes will remain consistent. Not only this, but also if a crashed node later comes back online, it guarantees that this node will eventually receive the same changes as all the other live nodes, further maintaining consistency. While not required for consistency, selecting a leader at the beginning of the algorithm makes the probability of deadlock much lower. In other words, it is still conceivable that deadlock could occur, but in practice it becomes unlikely enough as to be useful for practical applications.

Design Alternatives considered during implementation

Additionally, another challenge that the team faced in implementation of the Paxos algorithm was involving the leader election algorithm. Initially, the team proposed an algorithm analogous to flooding: each server would propose its own server number as the leader and broadcast it to all other servers. Then, each server would take the maximum of the process identification numbers it receives from other servers in this broadcast phase. Ultimately, since our network obeyed a fully-connected network topology, after one broadcast each server must have received process identification from each server in the network. As a result, this algorithm ensured that every server would acknowledge that the server with the highest process identification number in the network would be the leader. While this algorithm was efficient in the context of our problem, implementation was an arduous task, specifically with figuring out how to incite this broadcast process. Alternatively, the team considered implementing a simpler variant: any server that receives a request from the client believes itself to be a leader and all other servers to be acceptors. The implication of this algorithm was that there could exist a case where a server could be both a leader ("proposer") and an acceptor simultaneously. However, since the Paxos protocol does not concern itself with the leader election algorithm being used, this algorithm satisfied the leader election portion of our implementation.

Results

Our Paxos-based Replicated State Machine can handle up to N / 2 server faults where N is the initial number of servers. Since a similar exercise was completed in class involving replicated state machines, our benchmark was primarily the implementation of replicated state machines utilizing Lamport's Mutex Algorithm. A major benefit the group saw in using Paxos instead of Lamport's Mutex Algorithm was that the constraints on the communication channels were

relaxed. Specifically, the Lamport's Mutex Algorithm forced the communication channel to be FIFO. As a result, teams had to use TCP in order to implement replicated state machines with the Lamport's Mutex Algorithm. However, in Paxos Algorithm, the team used UDP for communication channels.

Additionally, the Paxos implementation handles fault-tolerance whereas the Lamport Mutex implementation of the server shopping problem could not handle any servers crashing. Primarily, this occurred in the Lamport Mutex implementation because the group kept track of number of alive servers to modify the number of acknowledgements needed to enter the critical section. Since Paxos Algorithm only requires a majority quorum and each server initially knows the total number of servers, the implementation is independent of the number of alive servers. Namely, each server will attempt to send to all other servers, regardless of whether they are alive or dead.

Conclusion

All in all, the Paxos algorithm is the most widely used protocol in distributed consensus utilized in industry through projects such as Google's Chubby. The team's implementation of the Paxos Algorithm was stratified into three stages: prepare, accept, and learn. The prepare phase allows for values to be proposed by the leader ("proposer") and requires each acceptor to grant a promise. If there exists a quorum of promises returned to the leader for the proposed value.. Then, the proposer sends an acceptance message to all of the remaining servers. Upon receiving another quorum of returns on these acceptance messages, the learner initiates a broadcast to all servers for the command to be executed. If however, a leader receives a higher sequence number from an acceptor, then it must have previously agreed upon a majority that this current leader has not yet seen. In this particular scenario, it begins to propose this new value. Until the server's original value is not decided, it will continue proposing its value.

References

[1] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32 (2) : 374–382, April 1985.

[2] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32 (4) : 51-58, November 2001.

[3] Vijay K. Garg. "Chapter 15: Agreement" in *Concurrent and Distributed Computing in Java*, 2nd edition, Hoboken, NJ: Wiley & Sons, 2004, pp. 209-223.