Paxos overview

Abstract

Paxos algorithm implements "state machine" approach where a service is made fault tolerant by running multiple copies (replicas) of it on independently failing nodes and assuring that all replicas receive exactly the same sequence of inputs and pass through exactly the same sequence of internal states. To reach this goal, Paxos uses a distributed non-blocking consensus protocol---Synod protocol.

Introduction

Lamport [0] and Schneider [4] popularized a "state machine" approach as a unified method to add fault-tolerance to a distributed service. In this method a service running on a node has to be represented as an automaton accepting a sequence of "commands" from the clients across the network and moving from one state to another as a result of command processing. It's important that automaton state transitions depend on nothing but the sequence of input commands (e.g., they do not depend on the absolute time measurements).

To add fault tolerance to such service it is replicated:

- identical instances of service state machines are ran on multiple nodes;
- clients send their commands to the elected leader and the leader broadcasts commands to all instances;
- because state depends only on the sequence of input commands, all instances execute precisely the same state transitions;
- voting among instances is used by clients to determine the outcome of the commands.

For this to work, broadcasting has to satisfy following conditions:

- all non-faulty instances receive all the commands, and
- all instances receive commands in the same order.

(See [1] for a straightforward exposition of a state machine approach and its relationship to other distributed architectures.)

At the heart of the Paxos algorithm is a distributed fault-tolerant consensus algorithm ("the Synod algorithm", described below) that allows instances to agree on a common value proposed by the leader. This algorithm is executed by the leader for every incoming command to reach an agreement on what command to execute next. That is, Synod algorithm uses (N, COMMAND) pair as a value, and agreement on this value means that all service instances promise to execute COMMAND as N-th command in the input sequence.

It might look that the description above is contradictory, because it assumes the existence of a leader and to select a leader participants have to agree on a common value (leader identity). Indeed, selection of a _unique_ leader is equivalent to distributed consensus problem, but Synod algorithm guarantees correctness even in the presence of multiple concurrent leaders. If multiple nodes claim to be leaders they can live-lock Synod progress for some transition time, until single leader is eventually elected, but they cannot violate its correctness. This is the crucial difference between the Synod algorithm and other would-be consensus algorithms like Three Phase Commit protocol that break in the face of multiple leaders. Note, that Paxos doesn't specify how the leader is elected, any existing algorithm, e.g., [2], can be used.

It must be added that Paxos alone is not sufficient to implement a fault tolerant distributed server as it doesn't deal with recovery of the failed replica.

The Synod Algorithm

In the simplest form of Synod algorithm there is a set of "proposers"---nodes that might be leaders, and a set of "acceptors"---nodes that are to agree on a common value. Acceptors might coincide with the state machine instances, or can form a different set of nodes. The role of acceptors is to make consensus fault tolerant. That is, there are two "levels" of fault tolerance in the Paxos: replicated state machines make service fault tolerant and acceptors make state machine synchrony fault tolerant.

The algorithm is executed as a sequence of numbered "ballots", each ballot involving two phase communication between its originating proposer and some subset of acceptors. The ballot is either successful, in which case a common value is agreed upon, or it fails due to proposer failure, network failure, a failure of sufficiently large number of acceptors, or a conflict with another ballot, in which case a new ballot is eventually started. In the non-faulty case there is only one ballot.

Certain sub-sets of the set of all acceptors are designated as "quorums". It is required that any two quorums have a non-empty intersection. For example, one can define quorum as any set containing a majority of acceptors (any two majorities obviously intersect).

The idea of Synod algorithm is that a proposer first tries to propose its preferred value for some quorum of acceptors (this is called "phase 1" of the ballot). If acceptors and network are alive, the proposer receives enough responses from the acceptors and learns from them whether the proposal can be accepted without conflicting past and ongoing activity of other proposers. If there is a conflict, proposer abandons its own value and instead is forced to uphold certain value that it learns from the acceptor responses and that is not in conflict. In any case, proposer moves into "phase 2" where it sends messages to acceptors asking them to accept its value (either original or forced). Once a quorum of acceptors successfully replies to this proposal, the consensus on value has been reached, and proposer can broadcast this value to state machine instances.

For every instance of Synod algorithm each proposer maintains on its stable storage a record

```
    struct proposer_state {
        /** maximal ballot number ever used by this proposer in
            this instance of Synod algorithm. */
        integer max_ballot;
    } proposer_state;
```

And every acceptor maintains a record

```
    struct acceptor_state {
        /** number of highest numbered ballot ACCEPTED by
            this acceptor. */
        integer max_accepted_ballot;
        /** value of highest numbered ballot ACCEPTED by
            this acceptor. */
        value_t max_accepted_value;
        /** number of highest numbered ballot PREPARED by
            this acceptor. */
        integer max_prepared_ballot;
    } acceptor_state;
```

An operation of synchronously committing a record to the stable storage is denoted by fsync(record). Message M with arguments (A0, ..., AN) sent from node P to node Q is denoted as

```
    M(A0, ..., AN) : P -> Q
```

A statement "On M(A0, ..., AN) : P -> Q { ... }" is executed on Q when a matching message is received there.

For simplicity we assume that the number of proposers is fixed and known in advance. This allows to easily generate unique ballot numbers: i-th proposer generates numbers in the sequence

```
    i, i + nr_proposers, i + 2 * nr_proposers, ...
```

It's assumed that proposer_state.max_ballot is initialized with a suitable "proposer number".

With these preliminaries Synod algorithm can be described as:

```
/** Synod algorithm for proposer P initially trying to
   propose value our_value. */
```

```
Proposer(node_t P, value_t our_value):
        value_t chosen;
        integer max_ballot;

        chosen = our_value;
        max_ballot = -1;
        Phase1: { /* Here execution starts when Synod is invoked or
                     when a proposer fails and restarts. */

                /* select a unique ballot number */
                proposer_state.max_ballot += nr_proposers;
                fsync(&proposer_state);
                /* select some quorum of acceptors */
                Q0 = a_quorum(acceptors);
                for_each(A in Q0)
                        PREPARE(proposer_state.max_ballot) : P -> A;

        }
        Phase 2 {
                On PREPARE_ACK(ballot, value) : A -> P {
                        if (ballot != NIL) {
                                /*
                                 * Acceptor A already accepted a ballot, find
                                 * highest numbered ballot accepted anywhere
                                 * in the quorum.
                                 */
                                if (ballot > max_ballot) {
                                        max_ballot = ballot;
                                        chosen     = value;
                                }
                        } else {
                                /* Acceptor A hasn't yet accepted any ballot. */
                        }
                }
                On PREPARE_NACK : A -> P { restart };
                On timeout for PREPARE { restart };
                /*
                 * Proposer received successful replies from the quorum. If
                 * none of acceptors accepted a value in any other ballot,
                 * phase 2 proposes original value our_value; otherwise the value
                 * from the highest numbered accepted ballot is proposed.
                 *
                 * Propose the value.
                 */
                Q1 = a_quorum(acceptors); /* This might be different from Q0 */
                for_each(A in Q1)
                        ACCEPT(proposer_state.max_ballot, chosen) : P -> A;

                On ACCEPT_ACK(ballot) : A -> P {;}
                On PREPARE_NACK : A -> P { restart };
                On timeout for PREPARE { restart };
                /*
                 * ACCEPT_ACK messages were received from the
                 * quorum. Consensus has been reached.
                 */
        }

Acceptor(node_t A):
        On PREPARE(ballot) : P -> A {
                if (ballot > acceptor_state.max_prepared_ballot) {
                        /*
                         * Acceptor received new highest numbered
                         * ballot. Remember this and reply. Acceptance of a
                         * ballot N extracts from acceptor a promise to not
                         * accept any ballot with number less than N.
                         */
                        acceptor_state.max_prepared_ballot = ballot;
                        fsync(&acceptor_state);
                        PREPARE_ACK(acceptor_state.max_accepted_ballot,
                                acceptor_state.max_accepted_value) : A -> P
                } else
                        /*
                         * Be faithful to this node's previous promise to not
                         * accept lower numbered ballots.
                         */
                        PREPARE_NACK : A -> P
        }
        On ACCEPT(ballot, value) : P -> A {
                if (ballot >= acceptor_state.max_prepared_ballot) {
                        acceptor_state.max_accepted_ballot = ballot;
                        acceptor_state.max_accepted_value  = value;
                        fsync(&acceptor_state);
                        ACCEPT_ACK(ballot) : A -> P
                } else
                        ACCEPT_NACK : A -> P
        }
```

The Synod algorithm is safe (i.e., guarantees that if a value is agreed upon, it is agreed upon consistently) under almost any failure: acceptor failing at any point, proposer failing at any point, network partitioning happening, messages being lost, duplicated, reordered, multiple nodes claiming to be the leader, etc. It handles "Byzantine failures" (i.e., situations where node runs arbitrary, including malicious, code) provided acceptors are redundant enough.

It's easy to note that proposer persistent state (`struct proposer_state`) is used only to generate unique ballot numbers. If other means to this end (like a monotonic clock surviving node failures) are present, no stable storage is necessary on a proposer.

Lustre

It is not clear how Synod relates to the Lustre recovery. In the existing Lustre code there is no need for any kind of consensus, because there is no replication. One might argue that a client and a server must reach a consensus on the results and effects of all operations that server reported as executed. Viewed from this point, a server acts like a proposer, and a client---as an acceptor, forcing server to accept values from its replay queue. Obviously this is a very degenerate case of a consensus problem. Similarly, a client and a server reach a consensus on the results of the last successfully committed request ("reconstruction" in Lustre terms). In this case, a client is a proposer and a server is an acceptor. Again, this is an extremely degenerate case.

Speaking about future Lustre code, Paxos-like solutions might be appropriate in the following places:

- raid1 for data: assuming well-behaved Lustre clients, only DLM lock owner sends conflicting (overlapping) writes, so there can be no more than one proposer at any time, making point of Paxos somewhat moot: much simpler algorithm can be used;
- on the other hand, DLM locks acquisition can nicely be expressed as a consensus problem. Currently Lustre solves it by acquiring locks synchronously in a well-defined order.

*** Question: can Paxos be used as a dead-lock avoidance mechanism guaranteeing that resources are acquired in the same order?

- agreement on global epochs. To prune undo logs "a global stability" of an epoch has to be detected. This condition is detected by "stability algorithm" that fits well into state machine approach. Fortunately, the core of this algorithm is MIN function that has very good properties (viz., idempotency) that enable much more light weight and scalable replication than direct application of Paxos would produce.
- fsync. To implement faithful cluster-wide fsync a set of servers must agree on a set of operations to synchronize. In this case there is a natural coordinator: a client, or a server where fsync RPC was sent to.
- write-back cache client. With global epochs a write-back cache client effectively inquires servers about their current epoch number (by sending lock ENQUEUE requests), finds the maximum of the received epochs numbers and its own epoch number, and then issues reintegration requests tagged with this maximum. This looks suspiciously similar to the two phases of the Paxos algorithm, except that no majority of any kind is involved, and server reply doesn't imply any kind of promise. The latter is direct consequence of having exactly one "ballot". The situations when one of the servers fails to execute updates of a distributed operation were never discussed in any detail, the reason being that it is always possible to invoke global recovery and to roll-back all inconsistent changes. Alternatively, reintegration can be re-tried, leading, as it seems, to the full Synod algorithm.

Concluding remarks

Paxos seems to be suited for fault tolerant maintenance of relatively small and slowly changing sets of critically important data. This is the view adopted by Chubby implementors and Lampson in [1] too.

Paxos guarantees absolutely identical sequence of execution histories of all state machine instances, which is usually much more than one needs: instances can execute operations in any order as long as _observable_, rather than internal, state is consistent between them (and state is observed through operations). One possible solution is to make state machines finer-grained (e.g., a state machine per file stripe rather than per file system). This works well for data, but breaks for meta-data, where a single operation can involve arbitrary objects.

Alternatively, a replication mechanism with weaker guarantees can be employed (see "virtual synchrony", "process groups", and CBCAST by Birman [3]).

References

[0] Lamport, Paxos made simple, http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#paxos-simple
[1] Lampson, How to Build a Highly Available System Using Consensus, http://research.microsoft.com/en-us/um/people/blampson/58-Consensus/Abstract.html
[2] Aguilera, et al., Stable Leader Election, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.4817
[3] Birman, The process group approach to reliable distributed computing, http://portal.acm.org/citation.cfm?id=163303&coll=portal&dl=ACM
[4] Schneider, Implementing fault-tolerant services using the state machine approach: A tutorial, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.4762