Dynamic Plain Paxos

Denis Rystsov
rystsov.denis@gmail.com
url: http://rystsov.info/files/dynamic-plain-paxos.pdf
draft v1.3

19 July 2015 (attested by Bitcoin)

Abstract

The classic Paxos consensus algorithm requires a static set of $2F + 1$ processes to tolerate F transient failures. Dynamic Plain Paxos is an extension and a drop-in replacement for the classic Paxos algorithm that allows to change the membership during the reaching of consensus. It satisfies the requirements of the real world systems to replace a permanently failed node or to extend the size of the cluster to tolerate more transient failures.

The article describes a generic way to prove the correctness of paxos-based distributes systems during the change of configuration and uses it to prove the correctness of membership changes.

1 Paxos

Classic Paxos is the key algorithm for building distributed systems in the same sense as b-tree is the key algorithm for building databases. With the classic Paxos algorithm, it is easy to build a distributed system that:

- consists of a fixed set of $2F + 1$ servers
- uses asynchronous unreliable non-byzantine messaging
- tolerates up to F server failures
- has an API for initializing a value and getting it with a strong consistency guarantee

The API consists of two methods:

```
void put(T value);
T? get();
```

Initially the value is not set and each server responds with *null* to every *get* method call (or fails with an error). To set a value, a client connects to any of the servers and calls the *put* method with the value. The method returns nothing (or fails with error). Afterwards, the client may call *get* on the same or on any other server to check if the value was set. Once *get* method returns some value on any server, there is a guarantee that every subsequent call on any server returns the same value or fails. This provides strong consistency.

Please see the original paper "Paxos Made Simple" [1] for the description of the Paxos algorithm and its proof. Proper understanding of the original paper is a prerequisite for reading this article.

Although Paxos looks very simple, it can be used as a foundation for more complex things. For instance, a distributed write-once key/value storage can be built by enhancing the algorithm to run an independent paxos instance per each key. A distributed state machine (Multi-Paxos) can be built by turning keys into a sequence of ordered numbers and values into commands.

1.1 Dynamic Membership

Unfortunately, pure Paxos is unapplicable to real life problems because it works only for a fixed set of servers, so the only way to tolerate permanent failures is to back up each server. It means that $2F+1$ additional servers are required to tolerate up to F permanent failures. It is too expensive.

Several papers suggest enhancements to the Paxos protocol that solve this problem. "Cheap Paxos" [2] and "Vertical Paxos" [3] are among them.

"Cheap Paxos" describes an enhancement on the Multi-Paxos protocol level. But it is harder to implement a distributed state machine (Multi-Paxos) than it is to implement a distributed write-once key/value storage than it is to implement a write-once distributed constant (classic Paxos). So Cheap Paxos forces the system to be much more complex.

"Vertical Paxos" solves the problem within individual consensus instances, since it operates on the classic paxos level. The author finds Vertical Paxos to be harder than classic Paxos since it introduces new entities like read and write quorums.

The aim of this article is to present an understandable enhancement of the classic Paxos protocol that allows dynamic change of membership. One may argue that there already exists designed-to-be-understandable consensus protocol Raft [4]. But, like Cheap Paxos, it operates on the Multi-Paxos level. So it has essential complexity that cannot be removed. Dynamic Plain Paxos works on the Paxos level, so potentially it may be more understandable than Raft.

2 The model of a distributed system

The original paxos paper describes a distributed system as a set of processes of three kinds: proposers, acceptors and learners. In this paper a triplet of these processes is called a node and the distributed system is a set of connected nodes. It is convenient to think of a node as a black box that consumes/produces messages, has a state and a configuration.

Definitions of the proposers and acceptors match the definitions from the original paper but learners act different — they are responsible for serving the *get* requests by implementing the following algorithm:

1. learner starts the first phase of the propose procedure
2. if a value was accepted by a quorum then leaner returns the value
3. otherwise if at least one acceptor returned a value then leaner finishes the propose procedure and restart the algorithm
4. in case all acceptors returned empty value after the first step leaner starts the first phase of the propose procedure again
5. if the same acceptors didn't change their mind and returned empty value again then leaner also returns empty value
6. otherwise it finishes the propose procedure and restart the algorithm

This algorithm guarantees that once a leaner returns a value any subsequent call to any learner returns the same value. It is easy to see that when a leaner returns a value then the value is replicated no less than on the quorum of the nodes (otherwise it wouldn't tolerate the desired number of failures).

3 Proof techniques

The basic idea behind the proof and the algorithm is very simple. We present the change of the cluster's configuration as a sequence of changing the configuration of each node, and go on to show that each step in the sequence is correct, and therefore the whole sequence is correct.

First, let us make two observations.

Observation 1. If two nodes generate the same output to the same input and one of them is part of the paxos cluster, then we can replace one with the other. The replacement does not influence the behavior of the cluster. Such nodes are called equivalent.

Observation 2. The behavior of the node can be controlled by dropping or delaying the messages it sends and receives. This does not affect the correctness of the cluster since Paxos works on top of unreliable network and is designed to tolerate such intrusions.

When combined, these observations produce a powerful way of changing the configuration of a node without breaking the correctness of the system.

4 The algorithm and the proof

Each step in the algorithm is reversible and makes an atomic transition of configuration without affecting the consistency. Hence we don't consider a possibility when a connection issue blocks the execution of the algorithm because we always can postpone the succeeding steps of the algorithm until the connection issue is resolved. We also ignore a failure of a node because we can rollback to the original configuration to handle it.

Let us demonstrate how to make a transition between a 3-node ABC paxos cluster (quorum size is 2) to a 4-nodes $ABCD$ cluster (quorum size is 3). Please see the appendix for a generic algorithm.

Consider the 3-node ABC cluster.

Step 1. We connect to the each node of the cluster and add a filter that affects the input messages. The filter groups the acceptor's responses by the ballot number and delays the delivery until the size of the group is 3. *Nodes with the filter do not affect the correctness of the cluster since the filter operates on the network layer and this intrusion fits the second observation. If we demonstrate that the system*

with quorum of size 2 and the filter is equivalent to the system with quorum of size 3, then we will be able to to increase the quorum size.

Statement. The behavior of the proposers with quorum size 2 and the filter matches the behavior of the proposer who think the quorum size is 3. *Proof. A proposer with quorum of size 2 and the filter does not make progress and does not issue output messages until it receives three responses from the acceptors. A proposer with quorum of size 3 behaves in the same way. QED.*

Acceptors do not depend on the quorum size by definition, but learners do. So we cannot increase the quorum size to 3 because it may break the leaner's invariant and the consistency. A learner might have returned a value before the increase when it was replicated only on 2 nodes, so if we increase the size of the quorum the leaner's invariant will be broken since we can't undo the served response.

Step 2. We connect to any node and execute the proposal procedure with any value, but abort it if all the responds to the prepare message are empty. The procedure will raise the replication factor of a value to 3 if any value has been accepted up to this point. *After the second step is complete, the system has either accepted a value and replicated to all three nodes or has not accepted it yet. As a consequence, the node with the filter and with quorum of size 2 is equivalent to the node with quorum of size 3.*

Step 3. We connect to each node and change the size of quorum from 2 to 3. Since the nodes are equivalent, it does not affect the correctness.

Consider a 4-node $ABCD$ cluster (quorum size is 3).

Step 6. We shut down the D node.

Step 5. We connect to the ABC nodes and add a filter that blocks the messages from and to the D node. *It does not affect the correctness, see the second observation.*

Statement. The ABC nodes behave in the same way (generate the same messages) as if they do not know about the D node. *Proof. Any message issued to the D node from an ABC node will not be ever received. It means if an ABC node stops sending messages to D it does not affect the correctness of the system. One can follow the original paxos paper and check that an ABC node stops sending messages to D but keeps the other behavior if we ask it to forget about D. When combined it proves the statement. QED.*

Step 4. We connect to the ABC nodes and remove the mention of D from the configuration on each node. *It doesn't affect the correctness, see the last statement.*

The $ABCD$ cluster after the 4th step has exactly the same behavior as the ABC cluster after the 3rd step. The state of each cluster is a valid state of the another cluster, so we can invert and apply the steps from 4 to 6 to the ABC cluster after the 3rd step and it will do the transition and keep the consistency. The simplest way to invert the 6th step is to assume that D node has never been active and start it with the default empty state.

5 Conclusion

The article described a framework for proving the state transition in a paxos cluster and used it to prove the algorithm for changing the membership during the reaching consensus in a distributed system. The author believes that the Dynamic Plain Paxos algorithm itself and its proof are simpler then the alternatives like Raft and Vertical Paxos.

Since the Dynamic Plain Paxos is a drop-in replacement for Paxos then any distributed systems based on static Paxos can be rewritten to use dynamic Paxos in theory without changing the system's architecture.

6 Acknowledgments

Thanks to Gleb Smirnov (@gvsmirnov) and Sergei Vorobev (@xvorsx) for multiple reviews and helpful suggestions.

7 Appendix

Appendix contains an explicit generic algorithms to add a node to a cluster and to remove it. The algorithms are listed without proofs because they are either identical to the to the $ABC \to ABCD$ case or straightforward.

7.1 Adding a node

Let us consider a task of change the size of a cluster from $2n+1$ to $2n+3$ nodes. First we break the task into two phases. The first phase makes a transition from $2n+1$ to $2n+2$ nodes and it is identical to the $ABC \to ABCD$ case:

1. add a filter to nodes which groups the responses from the acceptors by the ballot number and defers the delivery until the size of the group is $n+2$ *(hereafter if a step affects several nodes it means it doesn't matter whether we execute it serially or in parallel)*
2. try to get a value to increase its replication factor
3. increase quorum size on nodes
4. add an inactive empty node to configuration on nodes
5. turn on the new node

Second phase increases the size from $2n+2$ to $2n+3$ nodes. It is a straightforward way to do it since it doesn't affect the quorum size:

6. add an inactive node to configuration on nodes
7. turn on the new empty node

7.2 Removing a node

To remove a node we invert the algorithm we use to add it. Hence the algorithm also has two phases. First phase decreases the size of a the system from $2n+3$ to $2n+2$:

1. add a filter to nodes which blocks all messages to and from a node we want to remove

2. make nodes to forget about the node
3. remove the filter on nodes
4. shut down the node

Second phase reduces the size of the system from $2n + 2$ to $2n + 1$:

5. add a filter to nodes which blocks all messages to and from a node we want to rm
6. shut down the node
7. make nodes to forget about the node
8. remove the filter on nodes
9. add a filter to nodes which groups the responses from the acceptors by the ballot number and defers the delivery until the size of the group is $n + 2$
10. decrease the quorum size on nodes
11. remove the filter on nodes

Attesting

This paper is attested by Bitcoin. It means that anyone can check when it was created. To do it a person should calculate a Bitcoin address based on the sha256 hash code of the paper and check when a transfer to the address was made.

To find out the address you should use the following command

```
sha256sum -b dynamic-plain-paxos.pdf | awk '{print $1
   }' | python hashToAddress.py
```

The sources of hashToAddress.py are listed on the next page

```
import hashlib

def digest(method, hash):
  h = hashlib.new(method, hash.decode("hex"))
  return h.hexdigest()

ripemd160 = lambda x: digest("ripemd160", x)
sha256 = lambda x: digest("sha256", x)

__b58chars = "123456789" + \
  "ABCDEFGHJKLMNPQRSTUVWXYZ" + \
  "abcdefghijkmnopqrstuvwxyz"
__b58base = len(__b58chars)

def b58(hash):
  data = hash.decode("hex")
  long_value = 0
  for (i, c) in enumerate(data[::-1]):
    long_value += (256**i) * ord(c)
  result = ""
  while long_value >= __b58base:
    div, mod = divmod(long_value, __b58base)
    result = __b58chars[mod] + result
    long_value = div
  result = __b58chars[long_value] + result
  nPad = 0
  for x in data:
    if x != '\0': break
    nPad += 1
  return (__b58chars[0]*nPad) + result

footprint = raw_input()
footprint = "00" + ripemd160(footprint)
footprint = footprint + sha256(sha256(footprint))[:8]
print b58(footprint)
```

References

[1] Leslie Lamport, *"Paxos Made Simple"*. 2001.

[2] Leslie Lamport and Mike Massa, *"Cheap Paxos"*. 2003.

[3] Leslie Lamport, Dahlia Malkhi, Lidong Zhou, *"Vertical Paxos and Primary-Backup Replication"*. 2013.

[4] Diego Ongaro, John Ousterhout *"In Search of an Understandable Consensus Algorithm"*. 2013.