CSE535 Asynchronous Systems Multi-Paxos Final Report

JunXing Yang (108312768)

I. Part 0. State of the art

A. Abstract

This is the plan for the implementation of Multi-Paxos. Paxos is a family of protocols for solving consensus in a network of unreliable processors. The plan first describes the Multi-Paxos protocol and the problems that we face in the implementation, then proposes the algorithms for solving these problems. The plan also describes existing implementations of Multi-Paxos as well as the setups used to evaluate my implementation and compare with those existing work. A weekly plan for this project is also included.

B. Best problem descriptions

Multi-Paxos is a protocol for state machine replication in an asynchronous environment that admits crash failures [1]. The exact input and output of the protocol is as follows:

Input: Commands that sent by clients to replicas.
Output: A sequence of commands consented and performed by replicas.

There are three main goals that an implementation of Multi-Paxos desires to achieve.

The most important aspect of the protocol is correctness, which means that while concurrent commands may arrive in different orders at the replicas, all replicas receive commands in the same order and thus the replicated state machine behaves logically identical to a single remote state machine [1].

Another important feature is fault tolerance. Multi-Paxos tolerates crash-stop failures so that the system works correctly and grants progress as long as a subset of processes is alive and communicating [2].

Throughput and latency are also used to evaluate the implementation of Multi-Paxos.

C. Best algorithm descriptions

As described in [1], there is a set of processes performing specific protocol tasks, which is referred to as *replicas, acceptors, commanders, scouts* and *leaders*.

The precise pseudocode of each process is copied from [1], here we also present the overview description of each processes:

Replica: Replica receives a request for a command from a client and propose it to all the leaders. It waits for a decision from the leader, and then perform the command and send results to the client.Pseudocode shows in Fig. 1.

Acceptors: Acceptors sit waiting for messages from commanders or scouts and answer to them.Pseudocode shows in Fig. 2.

commanders: A commander sends a phase 2 message to all acceptors, and waits for responses. If a commander receives phase 2 response from all acceptors in a majority of acceptors, it notifies the replicas. Otherwise it notifies its leader about the Preemption.Pseudocode shows in Fig. 3.

scouts: A scout sends a phase 1 message to all acceptors, and waits for responses. If a scout receives phase 1 response from all acceptors in a majority of acceptors, it notifies its leader about the adoption. Otherwise it notifies its leader about the preemption. Pseudocode shows in Fig. 4.

leaders: A leader is responsible for spawning scouts, receiving proposes from replicas, and spawning commanders when notified about the success of phase 1 by scouts. Pseudocode shows in Fig. 5.

Besides the basic implementation of all kinds of processes in the protocol which ensures correctness described in Part B, there are a subset of optimizations suggested in [1] that could be included in this implementation to achieve more fault tolerance

```
process $Replica(leaders, initial\_state)$
  var $state := initial\_state, slot\_num := 1;$
  var $proposals := \emptyset, decisions := \emptyset;$

  function $propose(c)$
    if $\nexists s : \langle s, c \rangle \in decisions$ then
      $s' := \min\{s \mid s \in \mathbb{N}^+ \wedge$
          $\nexists c' : \langle s, c' \rangle \in proposals \cup decisions\};$
      $proposals := proposals \cup \{\langle s', c \rangle\};$
      $\forall \lambda \in leaders : send(\lambda, \langle \mathbf{propose}, s', c \rangle);$
    end if
  end function

  function $perform(\langle \kappa, cid, op \rangle)$
    if $\exists s : s < slot\_num \wedge$
            $\langle s, \langle \kappa, cid, op \rangle \rangle \in decisions$ then
      $slot\_num := slot\_num + 1;$
    else
      $\langle next, result \rangle := op(state);$
      atomic
        $state := next;$
        $slot\_num := slot\_num + 1;$
      end atomic
      $send(\kappa, \langle \mathbf{response}, cid, result \rangle);$
    end if
  end function

  for ever
    switch $receive()$
      case $\langle \mathbf{request}, c \rangle$ :
        $propose(c);$
      case $\langle \mathbf{decision}, s, c \rangle$ :
        $decisions := decisions \cup \{\langle s, c \rangle\};$
        while $\exists c' : \langle slot\_num, c' \rangle \in decisions$ do
          if $\exists c'' : \langle slot\_num, c'' \rangle \in proposals \wedge$
                $c'' \neq c'$ then
            $propose(c'');$
          end if
          $perform(c');$
        end while;
    end switch
  end for
end process
```

Fig. 1. Pseudocode of replica

```
process $Acceptor()$
  var $ballot\_num := \bot, accepted := \emptyset;$

  for ever
    switch $receive()$
      case $\langle \mathbf{p1a}, \lambda, b \rangle$ :
        if $b > ballot\_num$ then
          $ballot\_num := b;$
        end if;
        $send(\lambda, \langle \mathbf{p1b}, self(), ballot\_num, accepted \rangle);$
      end case
      case $\langle \mathbf{p2a}, \lambda, \langle b, s, c \rangle \rangle$ :
        if $b \geq ballot\_num$ then
          $ballot\_num := b;$
          $accepted := accepted \cup \{\langle b, s, c \rangle\};$
        end if
        $send(\lambda, \langle \mathbf{p2b}, self(), ballot\_num \rangle);$
      end case
    end switch
  end for
end process
```

Fig. 2. Pseudocode of acceptor

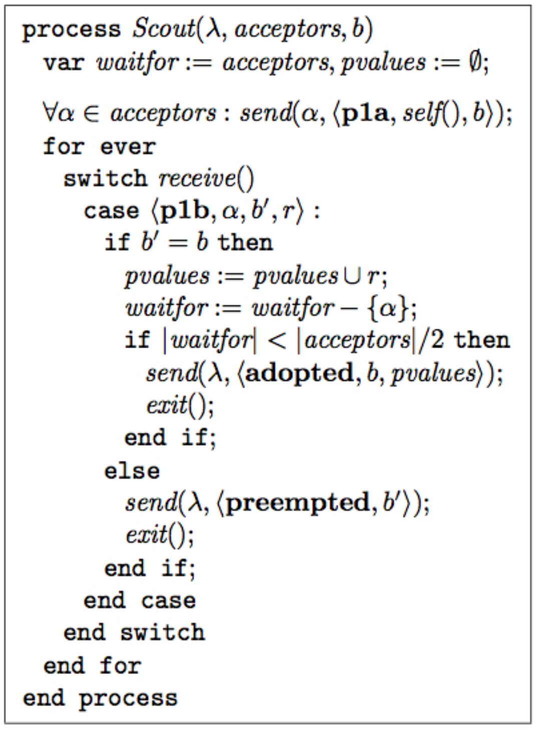
```
process $Commander(\lambda, acceptors, replicas, \langle b, s, c \rangle)$
  var $waitfor := acceptors;$

  $\forall \alpha \in acceptors : send(\alpha, \langle \mathbf{p2a}, self(), \langle b, s, c \rangle \rangle);$
  for ever
    switch $receive()$
      case $\langle \mathbf{p2b}, \alpha, b' \rangle$ :
        if $b' = b$ then
          $waitfor := waitfor - \{\alpha\};$
          if $|waitfor| < |acceptors|/2$ then
            $\forall \rho \in replicas$ :
              $send(\rho, \langle \mathbf{decision}, s, c \rangle);$
            $exit();$
          end if;
        else
          $send(\lambda, \langle \mathbf{preempted}, b' \rangle);$
          $exit();$
        end if;
      end case
    end switch
  end for
end process
```

Fig. 3. Pseudocode of commander

and performance mentioned in Part B, as described below:

a) State reduction: Acceptors only maintain the most recently accepted pvalue for each slot and return only these pvalues in a p1b message to the scout.

A leader keep track of which slots have already been decided so that it does not need to start commanders for slots for which it knows the decision,



```
process $Scout(\lambda, acceptors, b)$
  var $waitfor := acceptors, pvalues := \emptyset$;

  $\forall \alpha \in acceptors : send(\alpha, \langle \mathbf{p1a}, self(), b\rangle)$;
  for ever
    switch $receive()$
      case $\langle \mathbf{p1b}, \alpha, b', r\rangle$ :
        if $b' = b$ then
          $pvalues := pvalues \cup r$;
          $waitfor := waitfor - \{\alpha\}$;
          if $|waitfor| < |acceptors|/2$ then
            $send(\lambda, \langle \mathbf{adopted}, b, pvalues\rangle)$;
            $exit()$;
          end if;
        else
          $send(\lambda, \langle \mathbf{preempted}, b'\rangle)$;
          $exit()$;
        end if;
      end case
    end switch
  end for
end process
```

Fig. 4. Pseudocode of scout

and also does not need to maintain proposals for such slots.

The set *requests* maintained by a replica only needs to contain those requests for slot numbers higher than current slot number.

b) Garbage collection: When all replicas have learned that some slot has been decided, then it is no longer necessary for an acceptor to maintain the corresponding pvalues in its *accepted* set. Replicas could respond to leaders when they have performed a command, and upon a leader learning that all replicas have done so it could notify the acceptors to release the corresponding state.

The state of an acceptor would have to be extended with a new variable that contains a slot number: all pvalues lower than that slot number have been garbage collected.

c) Disk storage of states: In order to increase fault tolerance, the state of acceptors and leaders can be kept on stable storage (disk). This would allow such processes to recover from crashes. Considering that a process may crash partway during saving its state, an undo or redo logging similar to that in a database system can be used.

d) Efficiency: This part aims at improving the performance of the code. The idea is to use list comprehensions to replace some expensive for-loops, and minimize the number of set traversals. Some sets can be replaced by dictionaries which have an average O(1) searching cost. Another improvement will be to maintain some counters and avoid set queries.

D. Best implementations

1) Chubby: The implementation of Multi-Paxos in Chubby [3] can be considered as the best existing implementation in terms of being widely used and well-known. Chubby lock service is a fault-tolerant system at Google that provides a distributed locking mechanism and stores small files [4]. Chubby implements Multi-Paxos as the base for a framework that implements a fault-tolerant log. A number of advanced features in Chubby's Multi-Paxos is as follows:

a) Handling disk corruption: When a replicas disk is corrupted and it loses its persistent state, it won't renege on promises it has made to other replicas in the past.

b) Master leases: As long as a master has the lease, it is guaranteed that other replicas cannot successfully submit values to Paxos. Thus the master can be used to serve a read operation purely locally.

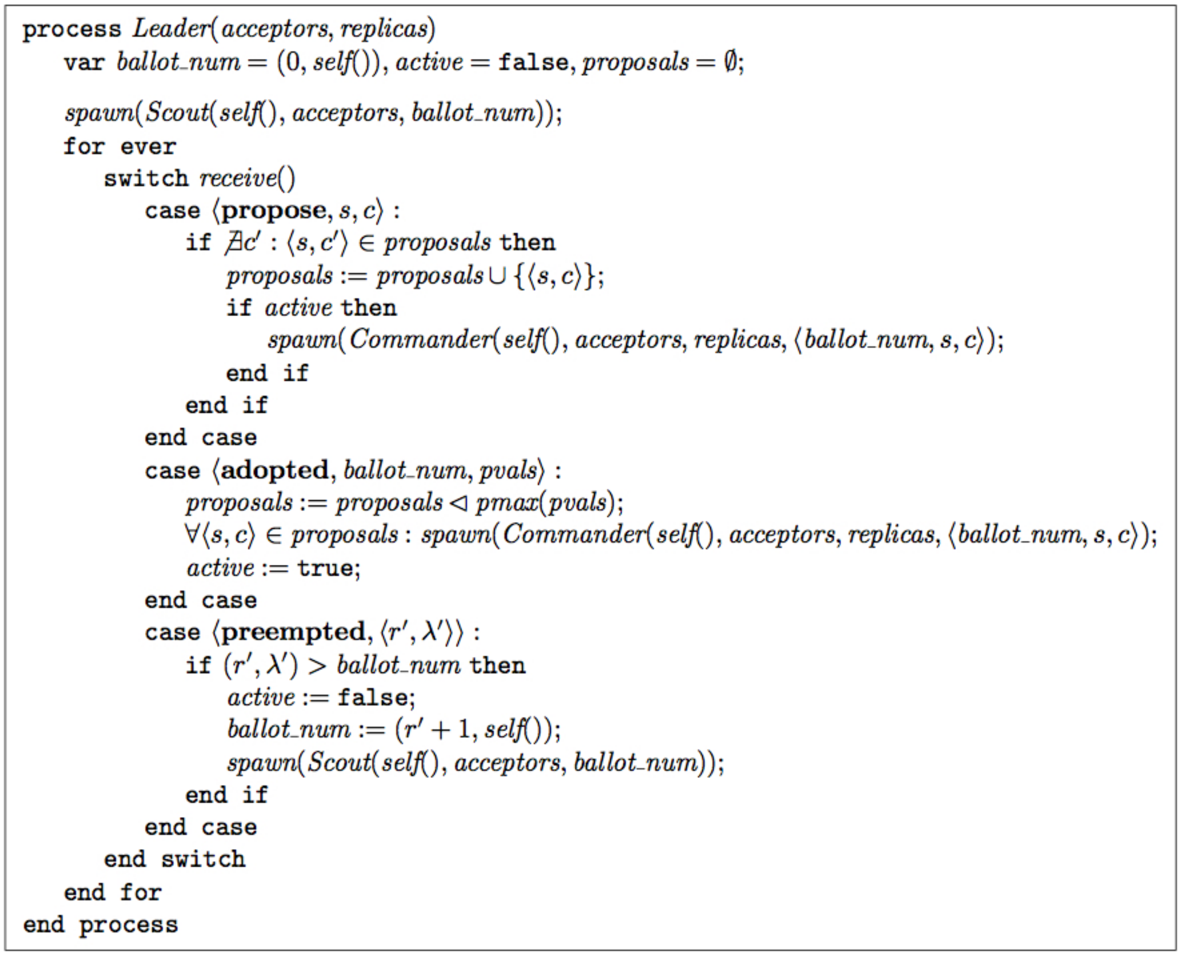
c) Group membership: Handling changes in the set of replicas.

d) Snapshots: A snapshot is a persistent representation for the data structure that the operations are applied on. Snapshots are used to minimize the amount of disk space and recovery time.

2) LibPaxos: LibPaxos [2] is one of the best implementations of Multi-Paxos in terms of open source as well as good performance. It is build on top of *libevent*, and includes leader election and various protocol optimizations, some of which are described below:

a) Phase 1 pre-execution and Parallel Phase 2: Executing phase 1 without waiting the next value from the client, which saves the two message delays required by phase 1. The leader can concurrently execute multiple phase 2 with different values to increase throughput.

b) Message and Value batching: In each packet Multi-Paxos messages are batched together. Also, the leader can use a single instance to broadcast different values to digest client values faster.



```
process $Leader(acceptors, replicas)$
    var $ballot\_num = (0, self()), active =$ false, $proposals = \emptyset$;

    $spawn(Scout(self(), acceptors, ballot\_num))$;
    for ever
        switch $receive()$
            case $\langle \textbf{propose}, s, c \rangle$ :
                if $\nexists c' : \langle s, c' \rangle \in proposals$ then
                    $proposals := proposals \cup \{\langle s, c \rangle\}$;
                    if $active$ then
                        $spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, c \rangle))$;
                    end if
                end if
            end case
            case $\langle \textbf{adopted}, ballot\_num, pvals \rangle$ :
                $proposals := proposals \lhd pmax(pvals)$;
                $\forall \langle s, c \rangle \in proposals : spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, c \rangle))$;
                $active :=$ true;
            end case
            case $\langle \textbf{preempted}, \langle r', \lambda' \rangle \rangle$ :
                if $(r', \lambda') > ballot\_num$ then
                    $active :=$ false;
                    $ballot\_num := (r' + 1, self())$;
                    $spawn(Scout(self(), acceptors, ballot\_num))$;
                end if
            end case
        end switch
    end for
end process
```

Fig. 5. Pseudocode of leader

c) Relaxed stable storage: The storage can be made volatile and thus much faster. When some number of failures is detected, leaders stop opening new instances. A snapshot is taken and the system can be then entirely restarted.

E. Project plan

1) Implementation: In the project, I will work on the implementation of Multi-Paxos with key features as described in Part B using *distalgo*.

The first thing to implement is the basic part of the protocol including *replicas, acceptors, commanders, scouts* and *leaders* as described in Part C. The pseudocode for each of these can be found in [1].

Next, implement the optimizations of the protocol including *state reduction, garbage collection, disk storage of states* and *efficiency*. The description of the algorithms for these optimizations can be found in Part C and [1].

2) Evaluation:

a) Testing consistency and fault-tolerance: The test can run in the safety mode [4]. In this mode, the test verifies that the system is consistent. It starts in safety mode and inject random failures into the system. After running for a predetermined period of time, we stop injecting failures and give the system time to fully recover. The failures may include network outages, message delays, timeouts, process crashes and recoveries, file corruptions, schedule interleavings, etc.

b) Evaluating throughput: Throughput is measured using *operations/s* or *MB/s* [4]. A typical setup for evaluating throughput is as follows (from the test of LibPaxos in [2]):
Setup:A client submits values for 5 minutes, at most 30 are sent concurrently. The size of values is random between 20 and 2000 bytes. Paxos runs with 3 acceptors, 2 leaders and 2 replicas.

3) Weekly plan: Tests will be conducted in every step of the implementation.

TABLE I
WEEKLY PLAN

Week	Plan
1	Basic components of Multi-Paxos (pseudocode from [1])
2	
3	Optimizations using incrementalization
4	State reduction
5	Garbage collection
6	Disk storage of states
7	Efficiency

II. PART 1. DESIGN

The code will be a modified version based on the original Multi-Paxos in distalgo [5]. Here we present a detailed design for each of the four improvements as mentioned in Part 0. An API documentation can be found on http://junxing.zxq.net/paxos.html

A. State Reduction

1) : Acceptors can remove old pvalues. So once an acceptor adopts a new ballot_num for a slot in receiving a P2a message the old ones with a smaller ballot_num for that slot can be removed.

2) : Leaders keep track of which slots have already been decided, and remove old proposals. In order to realize this, commanders will send decisions also to the leader. A leader will have a new message handler *OnDecision*, and uses a variable *first_slot* to maintain the first slot that it doesn't know the decision. In the handler, first_slot is updated and old proposals with a smaller slot number are removed.
In this situation, acceptors do not need to respond with pvalues (b, s, p) if $s < first\_slot$. A scout includes in its P1a message this first_slot which it gets from its leader.

3) : A replica can remove old proposals. So once an operation is performed, the proposals (s, p) will be removed if $s < slot\_num$

B. Garbage Collection

When all replicas have performed an operation, remove the corresponding pvalues in acceptors state. To achieve this, replica will send a message *Performed(s,p)* to the leaders when they have performed an operation. Leaders have dictionary *performed* with slot_num as keys, and values initially be the number of replicas. In their new message handler *OnPerformed*, they reduce the value by 1 and check whether all replicas have perform that operation. If so, they will send a message *Release(s)* to acceptors who will then remove the corresponding pvalues in message handler *OnRelease*.

C. Disk storage of states

1) : Use a list *state* to store the state of acceptors and leaders, and use a python module *pickle* to serialize and store the state to files. The state of acceptors is

$$[ballot\_num, accepted]$$

while for leaders it is

$$[ballot\_num, active, proposals, first\_slot, performed]$$

Acceptors and leaders update their states whenever it changes, and use the pickle function *dump()* to save the state to files.

2) : Recovery requires some modifications of the processes. It takes two more arguments to setup acceptors and leaders. One is *index* that identifies which acceptor or leader is recovered, because the recovered process will have different process id and port number, so the index is necessary for leaders to identify the acceptor or acceptors and replicas to identify the leader. It is also used to make sure the processes read and write the correct log file. Another one is a Boolean value *recovered* which indicates whether the process starts normally or it needs recovery. The procedure of recovery is as follows:

- open the log file
- retrieve the state using pickle function *load()*
- send message *Notify()* with its index. For acceptors, the message is sent to leaders, and for leaders it is sent to replicas and acceptors for leaders.

When receiving such a notification, processes will update the processes' list so they can communicate with the recovered one. And also:

- Leaders will spawn a new scout when an acceptor recovers
- Replicas will re-propose proposals with unknown decisions when a leader recovers

D. *Efficiency*

1) : In the original Multi-Paxos [5], it's expensive for a replica to find the minimum unused number for the proposal's slot number with multiply set traversals. Instead, a variable *max_slot* is used to keep track of the maximum slot number in proposals and decision, and return $max\_slot + 1$ as the new slot number. This could have potential waste of numbers, but in most cases will return the same result as the original one, and is much more efficient.

2) : Some sets will be replaced by dictionaries such as the set *waitfor* in commanders and scouts. It will reduce the time complexity from $O(n)$ to $O(1)$ for determining whether the majority of acceptors accept.

III. Part 2. Implementation

The implementation uses **DistAlgo v0.5** . The complete code is available on https://github.com/JunxingYang/Paxos/blob/master/paxos.da
The code size of modified and added code (not including test cases) is approximately 260 LoC with comments or 200 LoC without comments. The instructions of running the code will be described in Part 3. Note that this code needs driver programs such as test cases in Part 3 in order to run.

IV. Part 3. Testing and evaluation

To run the protocol, use test cases described below with command:

```
$python3 -m distalgo.runtime [testcase].da
```

You must have the source code of DistAlgo as well as the paxos code mentioned in Part 2 and test code described below in the same directory. The code can be found in the directory https://github.com/JunxingYang/Paxos with instructions in the README file.

A. *Test for correctness*

Test case 0 (test0.da):

- Simply run the protocol with 5 acceptors, 5 replicas, 3 leaders and 5 clients.

Result: the protocol runs with no error; the processes keep running with consensus.
The test shows that the code works well with state reduction and garbage collection, as compared to the original code [5] which fails after several seconds.

B. *Test for recovery*

1) Test case 1 (test1.da):
   - run paxos with 5 acceptors, 5 replicas, 3 leaders, 5 clients
   - 3 acceptors terminate after 2 seconds (more than half)
   - 1 acceptor recovers after 2 seconds

   Result: When 3 acceptors fail, clients won't receive results. After 1 acceptor recovers, clients receive results again.
   The test shows that the recovery for acceptors works correctly.
2) Test case 2 (test2.da):
   - run paxos with 5 acceptors, 5 replicas, 3 leaders, 5 clients
   - all leaders terminate after 2 seconds
   - 1 leader recovers after 2 seconds

   Result: When all leaders fail, clients won't receive results. After 1 leader recovers, clients receive results again.
   The test shows that the recovery for leaders works correctly.

The above tests show the implementation has complete functionalities as mentioned in Part 1 and works as expected.

References

[1] R. van Renesse, "Paxos made moderately complex," 2012. [Online]. Available: http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf

[2] M. Primi, "Paxos made code: Implementing a high throughput atomic broadcast," Master's thesis, University of Lugano, 2009. [Online]. Available: http://www.inf.usi.ch/faculty/pedone/MScThesis/marco.pdf

[3] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298487

[4] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 398–407. [Online]. Available: http://doi.acm.org/10.1145/1281100.1281103

[5] "Original multi-paxos in distalgo." [Online]. Available: http://www.cs.stonybrook.edu/~liu/cse535/orig.da