A Timestamp Based Consensus Algorithm

Zhaoguo Wang Shanghai Jiao Tong

1 Basic Protocol

2 Transaction Protocol

1.1 The Execution Algorithm

```
Algorithm 1: Leader::ProcessReQ (\gamma):
 1 // The replica who recieved y from client is considered as
 2 // the leader of \gamma and \gamma is a tuple of \langle op, key, value \rangle.
 3 PreAccept Phase:
 4 // Increase the instance number of replica L
 5 i_L = i_L + 1
 6 record = KVStore[\gamma.key]
 7 // Increase the record's timestamp
 8 record.ts = record.ts + 1
 9 logs[L][i_L] = (ballot_L, \gamma, record.ts, preAccepted)
10 // Each record has a queue which contains the access requests.
11\, // Each entry of the queue includes request's in the log and its
12 // status (pending/executable)
13 append \{L, i_L, pending\} to record.queue
14 send PreAccept(ballot_L, L, i_L, \gamma) to all
16 if recieve PreAcceptOK with the same timestamp
     from all replicas then
17
       for each reply do
18
           // Barrier is a local vector variable. Each entry
19
           // contains a log position of the request.
20
           put reply.lastReq into barrier
21
       goto Commit Phase
22
23 if receive PreAcceptOK from f+1 replicas then
24
       record.ts = the maximal replied timestamp
25
       logs[L][i_L].ts = record.ts
26
       logs[L][i_L].state = accepted
27
       lastReq = the last entry of record.queue
28
       put lastReq into barrier
29
       goto Accept Phase
30
31 Accept Phase:
32 send Accept(ballot_L, L, i_L, \gamma, record.ts) to all
33 if receive AcceptOK from f+1 replicas then
34
       for each replica R with reply reply_R do
35
          put reply_R.lastReq into the barrier
36
       logs[L][i_L].state = committed
37
       goto Commit Phase
38
39 Commit Phase
40 send Commit(ballot<sub>L</sub>, L, i_L, \gamma, ts, barrier) to all
41 ReadToProcess(\gamma, barrier)
```

```
Algorithm 2: Follower::PreAccept (ballot<sub>L</sub>,i,\gamma):
 1 if ballot_R > ballot_L then
 2 Ignore the request
 3 ballot<sub>R</sub> = ballot<sub>L</sub>
 4 record = KVStore[\gamma.key]
 5 record.ts = record.ts + 1
 6 logs[L][i_L] = (ballot_L, \gamma, record.ts, preAccepted)
 7 lastReq = the last entry of record.queue
 8 append \langle L, i_L, pending \rangle to record.queue
 9 return preAcceptOK, record.ts, lastReq
  Algorithm 3: Follower::Accept (ballot<sub>L</sub>, L, i_L, \gamma,ts):
 1 if ballot<sub>R</sub> > ballot<sub>L</sub> or log[L][i<sub>L</sub>] is committed then
 2 Ignore the request
 3 ballot_R = ballot_L
 4 record = KVStore[\gamma.key]
 5 if ts > \text{record.} ts then
 6 record.ts = ts
 7 logs[L][i_L].ts = record.ts
 8 \log[L][i_L].state = accepted
 9 lastReq = record.queue's last entry
10 if \gamma is not in record. queue then
11 append \langle L, i_L, pending \rangle to record.queue
12 return AcceptOK, record.ts, lastReq
```

Algorithm 4: Follower::Commit (ballot_L, L, i_L, γ , ts,

1 if $ballot_R > ballot_L$ or $log[L][i_L]$ is *committed* then 2 Ignore the request 3 $logs[L][i_L]$.state = *committed* 4 $record = KVStore[\gamma.key]$ 5 if γ is not in record. queue then

6 append $\langle L, i_L, pending \rangle$ to record.queue

- 7 ReadyToProcess(γ , barrier) **8** execute γ in the order of ts

Algorithm 5: Function::ReadyToProcess (γ , barrier):

```
1 record = KVStore[\gamma.key]
 2 for each entry in barrier do
 3
       R = entry.replica
 4
       i_R = entry.instance
 5
       wait until logs[R][i_R].state is committed
 6
       if timeout then
 7
           fetch logs[R][i_R] from replica R
 8
           if can not connect with R then
 9
              recover logs[R][i_R] with PREPARE
10
       \delta = \log[R][i_R].\text{cmd}
11
       if \delta.key is not equal to \gamma.key then
12
        continue
13
       req_{\delta} = \delta's corresponding entry in record.queue
14
       wait until req\delta.state is executable
15
       req_{\gamma} = \gamma's corresponding entry in record.queue
       req_{\gamma}.state = executable
```

Algorithm 6: Candidate::Prepare (L, i_L) :

- 1 // replica Q recover instance i_L in replica L
 2 ballot_Q = ballot_Q + 1
 3 send Prepare(i_L, L, ballot_Q) to all replicas
 4 if receive replies from a majority then
 5 | let S be the set of replies w/ the highest ballot number
 6 if S contains a committed entry then
 7 | for each reply do
 8 | put reply.lastReq into the vector barrier
 9 | goto Commit Phase
- 10 else if S contains an accepted entry then
- 11 goto Accept Phase
- 12 if S contains at least one *preAccepted* entry then
- **13 goto** PreAccept Phase
- 14 else
- **15 goto** PreAccept Phase with *nop* operation

Algorithm 7: Follower::Prepare (L, i_L , ballot_Q):

- 1 if $ballot_R > ballot_Q$ then
- 2 Ignore the request
- 3 ballot_R = ballot_O
- 4 $\gamma = \log[L][i_L].\text{cmd}$
- 5 $record = KVStore[\gamma.key]$
- 6 req $_{\gamma} = \gamma$'s corresponding entry in *record*.queue
- 7 lastReq = the last entry of *record*.queue
- **8 return** *prepareOk*, logs[L][i_L], lastReq

Algorithm 8: Coordinator::ProcessTxn (T=[$\gamma_1,...,\gamma_n$]):

- 1 Generate a TS with current wall clock
- **2** for γ_i , $i \in 1...n$ do
- 3 send $ProcessReq(\gamma_i, TS)$ to the closest participating server
- **4** Wait until recieve *recordTS* from all participating leaders
- 5 finalTS = maximal recordTS among the replies
- 6 send finalTS back to all leaders

```
Algorithm 9: Leader::ProcessReq (\gamma, TS):
                                                                        Algorithm 10: Follower::PreAccept (ballot<sub>L</sub>,i,\gamma):
 1 // The replica who recieved y from client is considered as
                                                                       1 if ballot_R > ballot_L then
 2 // the leader of \gamma and \gamma is a tuple of \langle op, key, value \rangle.
                                                                       2 Ignore the request
 3 PreAccept Phase:
                                                                       3 ballot<sub>R</sub> = ballot<sub>L</sub>
 4 // Increase the instance number of replica L
                                                                       4 record = KVStore[\gamma.key]
 5 i_L = i_L + 1
                                                                       5 record.ts = record.ts + 1
 6 record = KVStore[\gamma.key]
                                                                       6 logs[L][i_L] = (ballot_L, \gamma, record.ts, preAccepted)
 7 // Increase the record's timestamp
                                                                       7 lastReq = the last entry of record.queue
 8 record.ts = max(record.ts + 1, TS)
                                                                       8 append \langle L, i_L, pending \rangle to record.queue
 9 logs[L][i_L] = (ballot_L, \gamma, record.ts, preAccepted)
                                                                       9 return preAcceptOK, record.ts, lastReq
10\, // Each record has a queue which contains the access requests.
11\, // Each entry of the queue includes request's in the log and its
12 // status (pending/executable)
13 append \{L, i_L, pending\} to record.queue
14 send PreAccept(ballot_L, L, i_L, \gamma) to all
16 wait until receive PreAcceptOK from f+1 replicas
17 recordTS = the maximal replied timestamp
                                                                        Algorithm 11: Follower::Accept (ballot<sub>L</sub>, L, i_L, \gamma,ts):
18 send recordTS to coordinator
                                                                       1 if ballot<sub>R</sub> > ballot<sub>L</sub> or log[L][i<sub>L</sub>] is committed then
19
                                                                       2 Ignore the request
20 wait until recieve finalTS from coordinator
                                                                       3 ballot_R = ballot_L
22 if recieve PreAcceptOK with the timestamp of
                                                                       4 record = KVStore[\gamma.key]
      FinalTS from all replicas then
                                                                       5 if ts > \text{record.} ts then
23
       for each reply do
                                                                       6 \mid record.ts = ts
24
           // Barrier is a local vector variable. Each entry
                                                                       7 \log[L][i_L].ts = ts
25
           // contains a log position of the request.
                                                                       8 \log[L][i_L].state = accepted
26
           put reply.lastReq into barrier
                                                                       9 lastReq = record.queue's last entry
27
       goto Commit Phase
                                                                      10 if \gamma is not in record. queue then
                                                                      11 append \langle L, i_L, pending \rangle to record.queue
28
29 if receive PreAcceptOK from f+1 replicas then
                                                                      12 return AcceptOK, record.ts, lastReq
30
       logs[L][i_L].ts = FinalTS
31
       logs[L][i_L].state = accepted
32
       lastReq = the last entry of record.queue
33
       put lastReq into barrier
34
       goto Accept Phase
35
36 Accept Phase:
37 send Accept(ballot_L, L, i_L, \gamma, record.ts) to all
                                                                        Algorithm 12: Follower::Commit (ballot<sub>L</sub>, L, i<sub>L</sub>, \gamma, ts,
38 if receive AcceptOK from f+1 replicas then
39
       for each replica R with reply reply_R do
                                                                       1 if ballot<sub>R</sub> > ballot<sub>L</sub> or log[L][i<sub>L</sub>] is committed then
40
           put reply_R.lastReq into the barrier
                                                                       2 Ignore the request
41
       logs[L][i_L].state = committed
                                                                       3 logs[L][i_L].state = committed
42
       goto Commit Phase
                                                                       4 record = KVStore[\gamma.key]
43
                                                                       5 if \gamma is not in record. queue then
44 Commit Phase
                                                                       6 append \langle L, i_L, pending \rangle to record.queue
45 send Commit(ballot<sub>L</sub>, L, i_L, \gamma, ts, barrier) to all
                                                                       7 ReadyToProcess(\gamma, barrier)
46 ReadToProcess(γ, barrier)
                                                                       8 execute \gamma in the order of ts
```

```
Algorithm
                  13:
                         Function::ReadyToProcess (\gamma,
  barrier):
 1 record = KVStore[\gamma.key]
 2 for each entry in barrier do
 3
       R = entry.replica
       i_R = entry.instance
 5
       wait until logs[R][i_R].state is committed
 6
       if timeout then
 7
          fetch logs[R][i_R] from replica R
 8
          if can not connect with R then
 9
             recover logs[R][i_R] with PREPARE
10
       \delta = \log[R][i_R].\text{cmd}
11
       if \delta.key is not equal to \gamma.key then
       continue
12
13
       req_{\delta} = \delta's corresponding entry in record.queue
14
       wait until req\delta.state is executable
15
       req_{\gamma} = \gamma's corresponding entry in record.queue
16
       req_{\gamma}.state = executable
  Algorithm 14: Candidate::Prepare (L, i_L):
 {f 1} // replica Q recover instance i_L in replica L
 2 ballot_Q = ballot_Q + 1
 3 send Prepare(i_L, L, ballot_O) to all replicas
 4 if receive replies from a majority then
 5
       let S be the set of replies w/ the highest ballot
        number
       if S contains a committed entry then
 6
 7
          for each reply do
 8
             put reply.lastReq into the vector barrier
          goto Commit Phase
10 else if S contains an accepted entry then
    goto Accept Phase
12 if S contains at least one preAccepted entry then
13 goto PreAccept Phase
14 else
15 goto PreAccept Phase with nop operation
  Algorithm 15: Follower::Prepare (L, i_L, ballot<sub>O</sub>):
 1 if ballot_R > ballot_Q then
 2 Ignore the request
 3 ballot_R = ballot_Q
 4 \gamma = \log s[L][i_L].cmd
 5 record = KVStore[\gamma.key]
 6 req<sub>\gamma</sub> = \gamma's corresponding entry in record.queue
 7 lastReq = the last entry of record.queue
```

8 return prepareOk, $logs[L][i_L]$, lastReq