A Timestamp Based Consensus Algorithm

Zhaoguo Wang Shanghai Jiao Tong

1 Basic Protocol

1.1 The Execution Algorithm

2 Transaction Protocol

Algorithm 1: LEADER::PROCESSREQ (γ) :

// The replica who recieved γ from client is considered as
// the leader of γ and γ is a tuple of <op, key, value>.
PreAccept Phase:
// Increase the instance number of replica L
$i_L = i_L + 1$
record = KVStore[γ.key]
// Increase the record's timestamp
record.ts = *record*.ts + 1
logs[L][i_L] = (ballot$_L$, γ, *record*.ts, *preAccepted*)
// Each record has a queue which contains the access requests.
// Each entry of the queue includes request's in the log and its
// status (pending/executable)
append *{L, i_L, pending}* to *record*.queue
send *PreAccept(ballot$_L$, L, i_L, γ)* to all

if recieve *PreAcceptOK* with the same timestamp from all replicas **then**
  for each *reply* **do**
    // Barrier is a local vector variable. Each entry
    // contains a log position of the request.
    put *reply*.lastReq into *barrier*
  goto Commit Phase

if receive *PreAcceptOK* from *f+1* replicas **then**
  record.ts = the maximal replied timestamp
  logs[L][i_L].ts = *record*.ts
  logs[L][i_L].state = *accepted*
  lastReq = the last entry of *record*.queue
  put lastReq into *barrier*
  goto Accept Phase

Accept Phase:
send *Accept(ballot$_L$, L, i_L, γ,* record.*ts)* to all
if receive *AcceptOK* from f+1 replicas **then**
  for each replica *R* with reply *reply$_R$* **do**
    put *reply$_R$*.lastReq into the *barrier*
  logs[L][i_L].state = *committed*
  goto Commit Phase

Commit Phase
send *Commit(ballot$_L$, L, i_L, γ, ts, barrier)* to all
ReadToProcess(γ, barrier)

Algorithm 2: FOLLOWER::PREACCEPT (ballot$_L$,i,γ) :

if ballot$_R$ > ballot$_L$ **then**
  Ignore the request
ballot$_R$ = ballot$_L$
record = KVStore[γ.key]
record.ts = *record*.ts + 1
logs[L][i_L] = (ballot$_L$, γ, *record*.ts, *preAccepted*)
lastReq = the last entry of *record*.queue
append *<L, i_L, pending>* to *record*.queue
return *preAcceptOK*, *record*.ts, lastReq

Algorithm 3: FOLLOWER::ACCEPT (ballot$_L$, L, i_L,γ,ts) :

if ballot$_R$ > ballot$_L$ or log[L][i_L] is *committed* **then**
  Ignore the request
ballot$_R$ = ballot$_L$
record = KVStore[γ.key]
if *ts* > record.*ts* **then**
  record.ts = ts
logs[L][i_L].ts = *record*.ts
logs[L][i_L].state = *accepted*
lastReq = *record*.queue's last entry
if γ *is not in* record.*queue* **then**
  append *<L, i_L, pending>* to *record*.queue
return *AcceptOK*, *record*.ts, lastReq

Algorithm 4: FOLLOWER::COMMIT (ballot$_L$, L, i_L, γ, ts, barrier) :

if ballot$_R$ > ballot$_L$ or log[L][i_L] is *committed* **then**
  Ignore the request
logs[L][i_L].state = *committed*
record = KVStore[γ.key]
if γ *is not in* record.*queue* **then**
  append *<L, i_L, pending>* to *record*.queue
ReadyToProcess(γ, barrier)
execute γ in the order of *ts*

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

```
1  record = KVStore[γ.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     δ = log[R][i_R].cmd
11     if δ.key is not equal to γ.key then
12         continue
13     req_δ = δ's corresponding entry in record.queue
14     wait until req_δ.state is executable
15     req_γ = γ's corresponding entry in record.queue
16     req_γ.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_Q
4  γ = logs[L][i_L].cmd
5  record = KVStore[γ.key]
6  req_γ = γ'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=[γ_1,...,γ_n]) :

```
1  Generate a TS with current wall clock
2  for γ_i, i ∈ 1...n do
3      send ProcessReq(γ_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 (γ, TS) :

// The replica who recieved γ from client is considered as
// the leader of γ and γ is a tuple of <op, key, value>.
PreAccept Phase:
// Increase the instance number of replica L
$i_L = i_L + 1$
record = KVStore[γ.key]
// Increase the record's timestamp
record.ts = max(*record*.ts + 1, TS)
logs[L][i_L] = (ballot$_L$, γ, *record*.ts, *preAccepted*)
// Each record has a queue which contains the access requests.
// Each entry of the queue includes request's in the log and its
// status (pending/executable)
append *{L, i_L, pending}* to *record*.queue
send *PreAccept(ballot$_L$, L, i_L, γ)* to all

wait until *receive* PreAcceptOK *from* f+1 *replicas*
recordTS = the maximal replied timestamp
send *recordTS* to coordinator

wait until recieve *finalTS* from coordinator

if recieve *PreAcceptOK* with the timestamp of *FinalTS from all replicas* **then**
  for each *reply* **do**
    // Barrier is a local vector variable. Each entry
    // contains a log position of the request.
    put *reply*.lastReq into *barrier*
  goto Commit Phase

if receive *PreAcceptOK* from *f+1* replicas **then**
  logs[L][i_L].ts = *FinalTS*
  logs[L][i_L].state = *accepted*
  lastReq = the last entry of *record*.queue
  put lastReq into *barrier*
  goto Accept Phase

Accept Phase:
send *Accept(ballot$_L$, L, i_L, γ,* record.*ts)* to all
if receive *AcceptOK* from f+1 replicas **then**
  for each replica *R* with reply *reply$_R$* **do**
    put *reply$_R$*.lastReq into the *barrier*
  logs[L][i_L].state = *committed*
  goto Commit Phase

Commit Phase
send *Commit(ballot$_L$, L, i_L, γ, ts, barrier)* to all
ReadToProcess(γ, barrier)

Algorithm 10: FOLLOWER::PREACCEPT (ballot$_L$,i,γ) :

if ballot$_R$ > ballot$_L$ **then**
  Ignore the request
ballot$_R$ = ballot$_L$
record = KVStore[γ.key]
record.ts = *record*.ts + 1
logs[L][i_L] = (ballot$_L$, γ, *record*.ts, *preAccepted*)
lastReq = the last entry of *record*.queue
append <*L, i_L, pending*> to *record*.queue
return *preAcceptOK*, *record*.ts, lastReq

Algorithm 11: FOLLOWER::ACCEPT (ballot$_L$, L, i_L,γ,ts) :

if ballot$_R$ > ballot$_L$ or log[L][i_L] is *committed* **then**
  Ignore the request
ballot$_R$ = ballot$_L$
record = KVStore[γ.key]
if *ts* > record.*ts* **then**
  record.ts = ts
logs[L][i_L].ts = ts
logs[L][i_L].state = *accepted*
lastReq = *record*.queue's last entry
if γ *is not in* record.*queue* **then**
  append <*L, i_L, pending*> to *record*.queue
return *AcceptOK*, *record*.ts, lastReq

Algorithm 12: FOLLOWER::COMMIT (ballot$_L$, L, i_L, γ, ts, barrier) :

if ballot$_R$ > ballot$_L$ or log[L][i_L] is *committed* **then**
  Ignore the request
logs[L][i_L].state = *committed*
record = KVStore[γ.key]
if γ *is not in* record.*queue* **then**
  append <*L, i_L, pending*> to *record*.queue
ReadyToProcess(γ, barrier)
execute γ in the order of *ts*

Algorithm 13: Function::ReadyToProcess (γ, barrier) :

```
1  record = KVStore[γ.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     δ = log[R][i_R].cmd
11     if δ.key is not equal to γ.key then
12         continue
13     req_δ = δ's corresponding entry in record.queue
14     wait until req_δ.state is executable
15     req_γ = γ's corresponding entry in record.queue
16     req_γ.state = executable
```

Algorithm 14: 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 15: Follower::Prepare (L, i_L, $ballot_Q$) :

```
1  if ballot_R > ballot_Q then
2      Ignore the request
3  ballot_R = ballot_Q
4  γ = logs[L][i_L].cmd
5  record = KVStore[γ.key]
6  req_γ = γ's corresponding entry in record.queue
7  lastReq = the last entry of record.queue
8  return prepareOk, logs[L][i_L], lastReq
```