Paxos Made Practical

David Mazi`eres
Group member: 谢少锋, 尹诚诚, 杨令, 王岩

What is Paxos protocol?

- Paxos is a simple protocol that a group of machines in a distributed system can use to agree on a value proposed by a member of the group.

Terminology

- Proposal
- Proposer
- Acceptor
- Learner
- A majority

Learner

- Machine must notify the group of its proposal number before proposing a particular value. If, after hearing from a majority of the group, the proposer learns one or more values from previous proposals, it must re-use the same value as the highest-numbered previous proposal. Otherwise, the proposer can select any value to propose.

Three rounds of the Paxos(1)

- The proposer then broadcasts the message prepare(n).
- Network environment has uncertainty and parallelism, a and b simultaneously send requests, and the election number b > a, and b arrives at c, then c will not respond a's request. (Higher numbered proposals override lower-numbered ones.)
- Otherwise Replies with prepare-result (n', v') if the highest numbered proposal it has seen is n' < n for value v'.

Three rounds of the Paxos(2)

- If at least a majority of the group accepts the prepare message (indicating it as the latest proposal), the proposer moves to the second round. It sets v to the value in the highest-numbered prepare result it received. If v is nil, it selects any value it wishes for v.
- The proposer then broadcasts the message propose(n, v). Again, each group member rejects this message if it has seen a prepare(n') message with n' > n. Otherwise, it indicates acceptance in its reply to the proposer.
Three rounds of the Paxos

- If at least a majority of the group (including the proposer) accepts the propose message, the proposer broadcasts decide(n, v) to indicate that the group has agreed on value v.

How to make Paxos practical?

- Viewstamped Replication in cohorts is a simple example which used Paxos.
- Viewstamped Replication can use in?

State machine replication

- The service is deterministic, if two instances of the same state machine start in the same initial state and receive identical sequences of requests, they will also produce identical replies.
- Having one cohort choose all the nondeterministic values for an operation and having all cohorts use those values.

The server-side replication library

- The server-side replication library provides three functions:

  ```
  id_t newgroup (char *path);
  id_t joingroup (id_t group, char *path);
  int run (char *path, sockaddr *other_cohort, buf (*execute) (buf));
  buf choose (buf request);
  buf execute (buf request, buf extra);
  ```

The Client-side replication library

- On the client side, the replication library provides a matching function to execute:

  ```
  buf invoke (id_t group, sockaddr *cohort, buf request);
  ```

  It may be necessary to tell the client library how to contact members of the group.
  The cohort argument, if not NULL, can tell the library how to contact a member of the group. [primary cohort]

Paxos implement

- Both libraries can implement by the RPC calls.
- We assume every machine has a unique identifier of type cid_t.
- We assume that any machine may reboot at any time, and that as long as it doesn’t lose its disk, this doesn’t count as a failure. A cohort uses old cid_t after each reboot;
Normal-case operation

- One cohort in a group is designated the primary while the others are backups. [logs]

1. The client sends its request to the primary cohort.
2. The primary cohort logs the request and forwards it to all other cohorts.
3. Cohorts log the operation and send an acknowledgment back to the primary.
4. Once the primary knows that a majority of cohorts have logged the operation, it executes the operation and sends the result to the client.

```
struct viewstamp_t {
    viewid_t vid;
    unsigned ts;
};
```

Viewstamp specify the order in which cohorts must execute requests within a view.

```
union execute_res switch(bool ok) {
    ...
}
```

If ok is true, the reply field contains the result of the execute function. If ok is false, the client either got the view-id wrong or sent the request to a cohort other than the primary; in this case the viewinfo field contains the current view-id and primary.

When we need view change?

- One of the cohorts may suspect that another cohort has crashed because it fails to respond to messages.
- A new cohort may wish to join the system and when the membership of the group is changed.

View-change protocol

- This is a multi-step process like Paxos that involves first proposing a new view-id, then proposing the new view.

Terminology

- The cohort proposing the view-id is called the view manager for the new view.
- The view manager should not be confused with the primary; if the view manager succeeds in forming a new view, it may or may not become the primary in that view.
- We call cohorts that receive a view change request underlings, to distinguish them from the view manager that sent the RPC.
Who take part in the proposal?

- The state of the cohort is VC_ACTIVE.
- If an execute or replicate RPC arrives when the cohort is in one of the other states, it ignores the request.

```c
struct vc_state {
    enum {
        VC_ACTIVE,  /* active in a formed view: */
        VC_MANAGER,  /* proposing a new view: */
        VC_UNDERLING  /* invited to join a new view: */
    } mode;
    /* last (or current) formed view: */
    view_t view;
    /* last committed op at any cohort */
    viewstamp_t latest_seen;
    /* highest proposed new view-id: */
    viewid_t proposed_vid;
    /* accepted new view (if any): */
    view_t *accepted_view;
};
```

A cohort’s state

---

Proposing a new view-id (1)

```c
struct viewid_t {
    unsigned hyper counter;
    cid_t manager;
};
```

The lower part of the viewid_t is cid_t which is unique for every machines.

```c
struct cohort_t {
    oid_t id;
    net_address_t addr;
};
```

```c
struct view_t {
    viewid_t vid;
    cohort_t primary;
    cohort_t backups<>
};
```

```c
struct view_change_arg {
    view_t oldview;
    viewid_t newvid;
};
```

Proposing a new view-id (2)

```c
view_change.arg
view_change.res
new_view.arg
new_view.res
init_view.arg
```

Four possible cases (1)

1. Oldview is the most recent successfully formed view that the view manager knows about.
2. Each Cohort can accept or reject in the view_change.arg
3. Oldview.vid in the view_change.request is less than view.vid in the underling’s state. Thus, at least one subsequent view has already successfully formed since the one the manager wants to change. The underling therefore rejects the view change RPC.
4. Oldview.vid ≥ view.vid, but newvid in the request is less than proposed vid in the underling’s state. Thus, another manager has already proposed a higher new view-id. Again, the underling rejects the proposed view change, but updates view ← oldview in its state if view.vid < oldview.vid.

(View Manager 知道的view不是最新的)
(View Manager 知道的view不是最新的)
Four possible cases(2)

3. oldview.vid = view.vid and newvid ≥ proposed vid, so this is the highest new view-id the underling has seen proposed. However, accepted view is non-NULL, meaning the underling has already agreed to a particular configuration for the view following oldview.vid.
(underling cohort已经同意了其它VM的请求)

4. Either oldview.vid > view.vid (in which case the underling was not even aware of the last view), or else oldview.vid = view.id and accepted view = NULL. This is like the previous case, except the underling never agreed to any particular configuration for the view after oldview.

Proposing a new view (1)

- VM aborts the view change:
  - if any of the underlings reject the view change request.
  (When reject? in 2 case)
  - if it accepts a view change from a different view manager, thereby becoming an underling.

Proposing a new view(2)

- The view manager can form a new view V if:
  1. At least one cohort in V knows all past committed operations.
  2. No other view manager can form a view that executes operations concurrently with V.
     a. A majority of cohorts in Void agree on the configuration of V.
     2. If a majority of Void previously tried unsuccessfully to form a new view V', then V contains the same cohorts as V' and a majority of them agree V' never executed and never will execute a request.

Form the Group knows all past committed operations

- The view manager first determines what cohorts new view V will include, and all cohorts that replied with accept.include me set to true.
- If this set of cohorts is not guaranteed to contain a member that knows all past operations, the view manager starts additionally conscripting cohorts that set accept.include me to false.
- If the primary from Void replied with accept.include me false, the view manager includes it in the new view, which is sufficient.
- Otherwise, the view manager starts including cohorts in order of decreasing accept.latest fields until V contains a majority of Void.

Selects a primary

- the view manager selects a primary.
- If the primary from Void is also in V, the view manager keeps the same primary.
- Otherwise, it selects the cohort in V with the highest accept.latest field, breaking ties first by preferentially selecting itself, then arbitrarily.
Once they decide the primary:

```c
struct new_view_arg {
    viewstamp_t latest;
    view_t view;
};
```

An underlying cohort transferring the missing log entries (or entire current state) from the proposed new primary. Finally, it replies:

```c
struct new_view_res {
    bool accepted;
};
```

If the set of cohorts that has accepting the new view RPC grows to include both a majority of Vold and a majority of V, it is then safe to begin executing operations in view V.

If the manager is not the primary, it notifies the new primary with an init view RPC:

```c
struct init_view_arg {
    view_t view;
};
```