# Distributed consensus: Paxos

Shankar

May 10, 2022

# Distributed consensus service

- Distributed system: nodes interacting via message-passing
- Users at nodes can repeatedly submit values
- System provides users a consensus-log $\lambda$ of submitted values
  - each submitted value appears at most once in $\lambda$
  - each node $j$ maintains a prefix of $\lambda$
  - as values are submitted, $\lambda$ and the prefixes grow
  - users informed when their submitted values are rejected

- Typically desired in a failure-prone environments
- Eg, state machine (eg, filesystem) replicated at each node
  - values are commands
  - each state machine executes the log

# Failure model

- Nodes can fail
  - each node has some non-volatile memory
  - when a node fails, it does nothing
  - when a node recovers, its volatile state is re-initialiazed and its non-volatile state is preserved
  - this is a "fail-stop" ("non-Byantine") failure model
- Messages can be lost, duplicated, reordered
- At any time, at least half the nodes are working
- So for a value to be committed and always available, it must be in the non-volatile memory of a majority of nodes

# Paxos background

Each execution of Paxos proceeds in sessions (aka "rounds"). A node starts a session to append new values to the log. In the session, the node goes through a "prepare phase" followed by an "accept phase".

Prepare phase: The node sends prepare requests to other nodes, and receives prepare responses from them, each containing the responder's log (actually a log suffix). Upon receiving responses from a majority of nodes, it merges them into its log (which then equals the current consensus log), appends its new values to the log, and enters the accept phase.

Accept phase: The node sends accept requests with its log to other nodes, and receives accept responses from them, each signifying that the sender has incorporated the log. Upon receiving acks from a majority of nodes, the log is committed as the new consensus. (For brevity, our accept requests contain the entire log; it is easy to have it include only the needed suffix.)

At any time, multiple sessions can be ongoing. Each session has a unique session number, chosen by the node starting the session. At any time, a node participates in the highest-numbered session it knows of.

# Paxos node

Each node is defined by a set of variables and a set of rules.
Some variables are in non-volatile memory.

At any point, the values of the variables define the state of the node.

The rules define how the state of a node can change.
A rule has an enabling condition ("EC") predicate.
A rule is atomically executed only when its EC holds.
An input rule is initiated by the node's environment.
A non-input rule is initiated by the node.

Let M be the minimum number of nodes for a majority, ie, $(1 + \#\,\text{nodes})/2$.

# Node $j$ non-volatile variables

`csn` $\leftarrow$ `0` // current session number

Session number of this node's current session. Gets only increasing values. The session numbers generated by different nodes have no element in common (other than `0`). This is typically achieved by using the node id when generating a session number. (csn can be a $[int, j]$ tuple, in which case "`0`" would be `[0,`$j$`]`.)

`log` $\leftarrow$ `[[0,None]]`

Non-decreasing sequence of [session number, value] pairs. Given any log entry, the value field never changes but the session number field may increase (intially it indicates the session in which the value was generated).

`cx` $\leftarrow$ `0` // commit index

`log[0..cx]` is committed. The sequence of value fields of `log[0..cx]` is the prefix of $\lambda$ available at this node, ie, the "log" that this node's user sees.

# Node $j$ volatile variables

`role` $\leftarrow$ "follower"

Can be "follower", "candidate" or "leader". It is "candidate" if this node is in the prepare phase. It is "leader" if this node is in the accept phase. Otherwise it is "follower", ie, node did not initiate its current session.

`prs` $\leftarrow$ `{}` // prepare responses

Exists only when this node is a candidate. Map from node ids to log suffixes. `prs[k]` exists iff a prepare response was received from node k, in which case it equals the log suffix in the response.

`ars` $\leftarrow$ `{}` // accept responses

Exists only when this node is a leader. Map from node ids to commit indices. `ars[k]` exists iff an accept response was received from node k, in which case it equals the commit suffix in the response.

# Messages exchanged between nodes

Messages are tuples with the following fields:

*type*: PREQ, PRSP, AREQ, ARSP
  // prepare request/response, accept request/response
*src* // sender's node id
*sn* // sender's current session number
*cx* // sender's commit index
*log* // sender's log or suffx of log

**Prepare request**: [PREQ, *src*, *sn*, *cx*] // sent when candidate

**Prepare response**: [PRSP, *src*, *sn*, *log*] // response to PREQ
*log* is the sender's log suffix (starting at the requested *cx*)

**Accept request**: [AREQ, *src*, *sn*, *cx*, *log*] // sent when leader
*log* is the sender's log

**Accept response**: [ARSP, *src*, *sn*, *cx*] // response to AREQ

# Rules – 1

```
input rule submit(v):
  IA: role = "leader"
  log.append([csn,v])

rule become_candidate():
  EC: role = "follower"
  role ← "candidate"
  csn ← next(csn,j)
  prs ← {j:log[cx..]}

rule become_leader():
  EC: role = "candidate",
      prs.size ≥ M
  role ← "leader"
  log[0..cx].
      append(getmax(prs))
  ars ← {j:log.size}
```

```
rule commit():
  EC: role = "leader", ars.size ≥ M
  y ← max {n such that ars has M
       entries that are at least n}
  cx ← max(y, cx)

rule restart():
  EC: True
  role ← FOLLOWER

rule send_PREQ(k):
  EC: role = "candidate", k not in prs
  send(PREQ,j,csn,cx) to k

rule send_AREQ(k):
  EC: role = "leader",
      (k not in ars) or ars[k] < cx
  send(AREQ,j,csn,cx,log) to k
```

# Rules – 2

```
input recv(msg):
  if msg.sn > csn:
    role, csn ← "follower", msg.sn
  if msg.type = PREQ and msg.sn = csn:
    send(PRSP,j,csn,log[msg.cx+1..]) to msg.src
  elif msg.type = AREQ and msg.sn = csn:
   log ← msg.log    // if msg.log is a suffix, use extendlog
    cx ← max(cx, msg.cx)
    send(ARSP,j,csn,log.len) to msg.src
  elif msg.type = PRSP, role = "candidate", msg.sn = csn:
    prs[msg.src] ← msg.log
  elif msg.type = ARSP, role = "leader", msg.sn = csn:
    if msg.src not in ars:
      ars[msg.src] ← msg.cx
    else:
      ars[msg.src] ← max(ars[msg.src], msg.cx)
```

// Instead of ignoring msgs of old sessions, can send reject responses.

# Helper functions

```
helper getmax(prs):
  for k in prs, k≠ j:
    x ← min(prs[k].len, prs[j].len)
  for i in 0..x-1:
    if prs[j][i].sn < prs[k][i].sn:
       prs[j][i].val ← prs[k][i].val
    if prs[j].len < prs[k].len:
       prs[j].append(prs[k][i..])
  for in in 0..prs[j].len-1:
    prs[j][i].sn ← csn

helper extendlog(log, ix, rlog):  // currently not used in rules
  minlen ← min(log.len, rlog.len)
  for i in ix..minlen:
    if log[i].sn ≠ rlog[i].sn:
      break
  if i < minlen:
    log.remove(i..)
  log.append(rlog[i..])
```