```
6.824 2015 Lecture 5: Paxos
===========================

From Paxos Made Simple, by Leslie Lamport, 2001

starting a new group of lectures on stronger fault tolerance
  today:
    cleaner approach to replication: RSM via Paxos
    you'll need Paxos for Lab 3
  subsequent lectures:
    improved Paxos-like protocols (Raft)
    using replication in systems (Harp, later Spanner)

recall: RSM
  maintain replicas by executing operations in the same order
  requires all replicas to agree on the (set and) order of operations
  the point: if one server fails, can use other servers, which have state

Lab 2 critique
  primary/backup with viewserver
  pro:
    only two k/v servers needed to tolerate one failure
    handles network partition correctly (viewserver's partition wins)
  con:
    viewserver is single point of failure

network partition is the knottiest problem here
  over a network, one cannot distinguish:
    primary has crashed,
      so backup should take over
    primary is up and serving but net is partitioned,
      so backup should *not* take over
  viewserver solves this BUT is not fault-tolerant

paxos lets us build replication schemes that:
  handle partition correctly
  have no single point of failure

why paxos specifically?
  there are better protocols (Viewstamped Replication, Raft, ZAB)
  paxos is simple (as these things go)
  many other protocols are easy to view as variants of Paxos
  Paxos (and variants) are used a lot in real systems

there are two hard topics here
  1. how does Paxos work?
  2. how to use Paxos sensibly in a real system?
  #1 takes thought but is pretty well scoped
  #2 has been MUCH HARDER for people to figure out
  I'll talk about #2 first, for context

Paxos-based replication -- the big picture:
  [diagram: clients, replicas, log in each replica, k/v layer, paxos layer]
  no viewserver
  three replicas
  clients can send RPCs to any replica (not just primary)
  server appends each client op to a replicated *log* of operations
    Put, Get, Append
  numbered log entries -- instances -- seq
  Paxos agreement on content of each log entry
```

```
    note: each instance (log entry) is an entirely separate Paxos agreement
    with entirely separate proposal numbers

what does Paxos provide?
  Lab 3 interface on each server:
    Start(seq, v) -- to propose v as value for instance seq
    fate, v := Status(seq) -- to find out the agreed value for instance seq
  correctness:
    if agreement reached, all agreeing servers agree on same value
    corollary: once any agreement reached, never changes its mind
    critical since, after agreement, servers may update state or reply to clients
    (may not agree if too many lost messages, crashed servers)
  fault-tolerance:
    can tolerate non-reachability of a minority of servers
  liveness:
    will reach agreement when a majority can communicate for long enough

example:
  client sends Put(a,b) to S1
  S1 picks log entry 3
  S1 uses Paxos to get all servers to agree that entry 3 holds Put(a,b)

example:
  client sends Get(a) to S2
  S2 picks log entry 4
  S2 uses Paxos to get all servers to agree that entry 4 holds Get(a)
  S2 scans log up to entry 4 to find latest Put(a,...)
  S2 replies with that value
  (S2 can cache content of DB up through last log scan)

why a log?
  why not require all replicas to agree on each op in lock-step?
  doesn't matter if the state is small: can agree on entire state
  log is a big win if state is very large; log describes changes
  log helps replicas catch up
    if slow, miss messages, crash and restart

summary of how to use Paxos for RSM:
  a log of Paxos instances
  each instance's value is a client command
  different instances' Paxos agreements are independent
  this is how Lab 3B works

now let's switch to how a single Paxos agreement works

agreement is hard (1):
  may be multiple proposals for the op in a particular log slot
  Sx may initially hear of one, Sy may hear of another
  clearly Sx or Sy must change its mind
  thus: multiple rounds, tentative initially
  how do we know when agreement is permanent -- no longer tentative?

agreement is hard (2):
  if S1 and S2 are happy with a value, and S3 and S4 don't respond, are we done?
  agreement has to be able to complete even w/ failed servers
  we can't distinguish failed server from network partition
  so maybe S3/S4 are partitioned and have "agreed" on a different value!

two main ideas in Paxos:
  1. many rounds may be required but they will converge on one value
  2. a majority is required for agreement -- prevent "split brain"
```

```
     a key point: any two majorities overlap
     so any later majority will share at least one server w/ any earlier majority
     so any later majority can find out what earlier majority decided

Paxos sketch
  each server consists of three logical entities:
    proposer
    acceptor
    learner
  may be more than one proposer
    if multiple clients submit requests at the same time to diff servers
  each proposer wants to get agreement on its value
  proposer contacts acceptors, tries to assemble a majority
    might not get majority -> new round

basic Paxos exchange:
 proposer        acceptors
     prepare(n) ->
  <- prepare_ok(n, n_a, v_a)
     accept(n, v') ->
  <- accept_ok(n)
     decided(v') ->

why n?
  to distinguish among multiple rounds, e.g. proposer crashes, simul props
  want later rounds to supersede earlier ones
  numbers allow us to compare early/late
  n values must be unique and roughly follow time
  n = <time, server ID>
    e.g., ID can be server's IP address
  "round" is the same as "proposal" but completely different from "instance"
    round/proposal numbers are WITHIN a particular instance

definition: server S accepts n/v
  S responded accept_ok to accept(n, v)

definition: n/v is chosen
  a majority of servers accepted n/v

the crucial property:
  if a value was chosen, any subsequent choice must be the same value
    i.e. protocol must not change its mind
    maybe a different proposer &c, but same value!
    this allows us to freely start new rounds after crashes &c
    AND it allows a server to safely execute a chosen command, or reply to client
  tricky b/c "chosen" is system-wide property
    e.g. majority accepts, then proposer crashes
    no server can tell locally that agreement was reached

so:
  proposer doesn't send out value with prepare
  acceptors send back any value they have already accepted
  if there is one, proposer proposes that value
    to avoid changing an existing choice
  if no value already accepted,
    proposer can propose any value (e.g. a client request)
  proposer must get prepare_ok from majority
    to guarantee intersection with any previous majority,
    to guarantee proposer hears of any previously chosen value

now the protocol -- see the handout
```

```
proposer(v):
  choose n, unique and higher than any n seen so far
  send prepare(n) to all servers including self
  if prepare_ok(n, n_a, v_a) from majority:
    v' = v_a with highest n_a; choose own v otherwise
    send accept(n, v') to all
    if accept_ok(n) from majority:
      send decided(v') to all

acceptor state:
  must persist across reboots
  n_p (highest prepare seen)
  n_a, v_a (highest accept seen)

acceptor's prepare(n) handler:
  if n > n_p
    n_p = n
    reply prepare_ok(n, n_a, v_a)
  else
    reply prepare_reject

acceptor's accept(n, v) handler:
  if n >= n_p
    n_p = n
    n_a = n
    v_a = v
    reply accept_ok(n)
  else
    reply accept_reject

example 1 (normal operation):
  S1, S2, S3
  but S3 is dead or slow
  S1 starts proposal, n=1 v=A
S1: p1    a1A    dA
S2: p1    a1A    dA
S3: dead...
"p1" means Sx receives prepare(n=1)
"a1A" means Sx receives accept(n=1, v=A)
"dA" means Sx receives decided(v=A)
these diagrams are not specific about who the proposer is
  it doesn't really matter
  the proposers are logically separate from the acceptors
  we only care about what acceptors saw and replied

Note proposer only ever needs to wait for a majority of the servers
  so we can continue even though S3 was down
  proposer must not wait forever for any one acceptor's response

What would happen if network partition?
  I.e. S3 was alive and had a proposed value B
  S3's prepare would not assemble a majority

the homework question:
  How does Paxos ensure that the following sequence of events can't
  happen? What actually happens, and which value is ultimately chosen?
  proposer 1 wants v=X, crashes after sending two accepts
  proposer 2 wants v=Y
  S1: p1 a1X
  S2: p1     p2 a2?
```

```
  S3: p1 a1X p2 a2?
  S3's prepare_ok to proposer 2 really included "X"
    thus a2X, and so no problem
  the point:
    if the system has already reached agreement, majority will know value
    any new majority of prepares will intersect that majority
    so subsequent proposer will learn of already-agreed-on value
    and send it in accept msgs

example 2 (concurrent proposers):
S1 starts proposing n=10
S1 sends out just one accept v=X
S3 starts proposing n=11
  but S1 does not receive its proposal
  S3 only has to wait for a majority of proposal responses
S1: p10 a10X
S2: p10       p11
S3: p10       p11  a11Y
S1 is still sending out accept messages...
has a value been chosen?
could it go either way (X or Y) at this point?
what will happen?
  what will S2 do if it gets a10X accept msg from S1?
  what will S1 do if it gets a11Y accept msg from S3?
what if S3 were to crash at this point (and not restart)?

how about this:
S1: p10  a10X              p12
S2: p10         p11  a11Y
S3: p10         p11        p12   a12X
has the system agreed to a value at this point?
  after all, a majority have accepted value "X"

what's the commit point?
  i.e. exactly when has agreement been reached?
  i.e. at what point might a server have executed the command?
  after a majority has the same v_a? no -- why not?  above counterexample
  after a majority has the same v_a/n_a? yes -- why sufficient?  sketch:
    suppose majority has same v_a/n_a
    acceptors will reject accept() with lower n
    for any higher n: prepare's must have seen our majority v_a/n_a (overlap)

why does the proposer need to pick v_a with highest n_a?
S1: p10  a10A              p12
S2: p10         p11  a11B
S3: p10         p11  a11B  p12   a12?
n=11 already agreed on vB
n=12 sees both vA and vB, but must choose vB
why: two cases:
  1. there was a majority before n=11
     n=11's prepares would have seen value and re-used it
     so it's safe for n=12 to re-use n=11's value
  2. there was not a majority before n=11
     n=11 might have obtained a majority
     so it's required for n=12 to re-use n=11's value

why does prepare handler check that n > n_p?
  it doesn't have to: a proposer that fails the check in
    prepare handler will fail same check in accept handler

why does accept handler check n >= n_p?
```

```
    to ensure later proposer sees any possible chosen value
      by preventing acceptance of old value once an acceptor
      has responded to new proposer's prepare
    w/o n >= n_p check, you could get this bad scenario:
    S1: p1 p2 a1A
    S2: p1 p2 a1A a2B
    S3: p1 p2     a2B
    oops, for a while A was chosen, then changed to B!

  why does accept handler update n_p = n?
    required to prevent earlier n's from being accepted
    server can get accept(n,v) even though it never saw prepare(n)
    without n_p = n, can get this bad scenario:
    S1: p1    a2B a1A p3 a3A
    S2: p1 p2         p3 a3A
    S3:    p2 a2B
    oops, for a while B was chosen, then changed to A!

  what if proposer S2 chooses n < S1's n?
    e.g. S2 didn't see any of S1's messages
    S2 won't make progress, so no correctness problem

  what if an acceptor crashes after receiving accept?
  S1: p1  a1X
  S2: p1  a1X reboot  p2  a2?
  S3: p1              p2  a2?
  the story:
    S2 is the only intersection between p1's and p2's majorities
    thus the only evidence that Paxos already chose X
    so S2 *must* return X in prepare_ok
    so S2 must be able to recover its pre-crash state
  thus: if S2 wants to re-join this Paxos instance,
    it must remember its n_p/v_a/n_a on disk.

  what if an acceptor reboots after sending prepare_ok?
    does it have to remember n_p on disk?
    if n_p not remembered, this could happen:
    S1: p10           a10X
    S2: p10 p11 reboot a10X a11Y
    S3:     p11            a11Y
    11's proposer did not see value X, so 11 proposed its own value Y
    but just before that, X had been chosen!
    b/c S2 did not remember to ignore a10X

  can Paxos get stuck?
    yes, if there is not a majority that can communicate
    how about if a majority is available?
      if proposers immediately retry w/ higher n after accept_reject,
        they can all keep each other from getting accepts accepted
      so don't retry immediately!
      pause a random amount of time, then re-try
```