Primary Backup and Lamport Clocks

Tom Anderson

Last Time

- Primary/backup
  - Goals
  - Normal case operation
  - Failover/repair
  - Corner cases

Primary/Backup: The Goal

System behaves as if it is a single server that (almost) never fails.

Primary/Backup: The Players

- View service
  - Decide which server is primary, which is backup, as sequence of "views"
- Primary
  - Decide order of client operations with a view
  - Bring backup up to date, if needed
- Backup
  - Replicates state at the primary
- Clients
  - Can be many

View Server

Servers Ping view server
- if more than two, the others are "idle" servers

If primary is dead
- new view with previous backup as primary

If backup is dead, or no backup
- new view with previously idle server as backup

Don't switch view if backup is not up to date
- As acked by the primary in the new view

Views

- Viewserver declares new view
- Clients, old primary, new primary, new backup
  - All may still be in old view
- Include current view in RPCs
  - Allows receiver to know if caller is out of date
  - Allows receiver to tell caller if out of date
  - Allows receiver to know if receiver is out of date

Repair

- Primary fails
- View server declares new “view”, promotes backup to primary
- View server promotes “idle” server as new backup
- Primary initializes new backup's state
- Now ok if new primary fails

More Than One Primary

1: S1, S2
- Network breaks, so viewserver thinks S1 dead
- But S1 is still alive

2: S2, --
- But S1 alive and not aware of view #2, so S1 still thinks it is primary
- AND S2 alive and thinks it is primary

- Called “split brain”

More Than One Primary

Can we ensure only one server acts as primary?
- Even though more than one may *think* it is primary
- Acts as = responds to client requests

1: S1 S2

2: S2 --

- S1 still thinks it is primary
- S1 must forward operations to S2 (for backup!)
- S2 thinks S2 is primary
- S2 can reject S1's forwarded operations

Rules

1. Primary in view i must have been primary or backup in view i-1
2. Primary must wait for backup to accept/execute each request before doing operation and replying to client
3. Non-backup must reject forwarded requests
   - Backup accepts forwarded requests only if they are in the current view
4. Non-primary must reject direct client requests
5. Every operation must be before or after state transfer

Example

1: S1, S2
- View server stops hearing Pings from S1

2: S2, --
- It may be a while before S2 hears about view #2

Before S2 hears about view #2:
- S1 can process ops from clients, S2 will accept forwarded ops
- S2 will reject ops from clients who have heard about view #2 [but trigger a fetch to get new view from view server]

After S2 hears:
- if S1 receives client op, it will forward but S2 will reject
- S1 will send error to client, client will ask view server for new view
- Client will re-send op to S2

State Transfer

How does a new backup get the current state?
- e.g. the contents of the key/value store
- If S2 is backup in view i, but was not in view i-1
- S2 asks primary to transfer the state

Rules for state transfer:
- every op must be either before or after state xfer
- if op before xfer, xfer must reflect op
- if xfer before op, primary forwards op after xfer finishes

Three Approaches To State Transfer (Key-Value Store)

1. Transfer current k-v map
   - Between operations! All RPCs appear to occur either before or after the transfer.
   - Recall that "at most once" means you need to save the response you previously gave to an RPC. When backup takes over, needs to act as if it was the primary.
   - If each client has only one RPC outstanding at a time, state = map + result of last RPC from each client.
2. State can be derived from an operation log: the sequence of RPC's processed at the primary.
   - Simpler, less efficient
   - Current value of key is last value written in the log for that key
3. Transfer checkpoint plus log of recent changes to k-v map

Fast Reads

- Does the primary need to forward read (Get) requests to the backup, or can it reply directly?

Progress

Are there cases when the protocol cannot make forward progress?

Implementation Hint

In Section, transition diagram for the view server for Lab 2a. What about Lab 2b?

- State transition diagram for primary?
- State transition diagram for backup?
- State transition diagram for client?
- All of the above plus the viewserver?

Implementation Hint

Behavior of a distributed system is the cross-product of the states of the individual nodes

- state at the clients
- state at the primary and backup
- state at the view server

Transitions are message send/receive and local events

Many possible paths are allowable -- depending on which messages are delivered in which order, which nodes do which actions

Lamport Clocks

Can we make sure everyone agrees on the same order of events?

An issue if:

- Multiple clients, multiple servers
- impractical to have one node decide on ordering

Even if there are no failures

Even if messages are delivered in order sent by each client ("processor order")

Facebook Storage System

Initially:

- a few front end web servers to do application logic
- a single backend storage server

To scale, add more front ends, more back end servers:

- Each front end pulls data from multiple servers (e.g., one for privacy settings, one for pictures).
- Do users see a consistent view?

Now add some intermediate caches:

- 100+ lookups per page
- 1B+ users: 1M+ front ends, 1M+ caches, 1M+ servers

Facebook Architecture

Example: Arranging Lunch

Example: P2P Whiteboard

Example: Parallel Make

Physical Clocks

- Can we assign every event in a distributed system a unique wall clock time stamp?
- Local clocks aren't perfect
  - Crystals oscillate at slightly different frequencies
  - Typical error is ~ 2 seconds/month
- Synchronize clocks across distributed system?
  - Network messages involve delays
  - Network message delays are variable

Physical Clocks

- Lets assume a network-attached GPS
  - How close can we bound clocks across multiple systems?
- Option 1: client polls the GPS server for current time.
  - How far off will the timestamp be when it arrives back at the client?
- Option 2: repeatedly fetch the GPS time, estimate relative rate of skew of the local clock

Logical Clocks (Centralized implementation)

Send every message to a central arbiter, which assigns an order for all messages.

Problems with centralization?

Space-Time Diagram

Happens Before

Happens before: a->b, ie, b could be caused by a

1. If a and b are in same process, a occurs before b
2. Send occurs before receive
3. Transitivity

Logical clock C: Cj is the value of clock C for process j st if "a happened before b" then C(a) < C(b)

Logical Clock Update

IR1: increment on every event within a process

IR2: put current clock value Ci in each message sent by process i

On receipt at process j, Cj = max(current Cj, Ci + 1)

If apply tiebreaking rule, then we get a total order consistent with partial order.

Example

Globally Ordered Message Delivery

Proposed rule: Always deliver packets everywhere in the same total order, defined by Lamport clock.

Q: When is it ok to act on a message that arrives?

- Can you know that you have received all msgs that might ever arrive with earlier timestamp?

Updated Rule

Proposed rule: Always deliver packets everywhere in the same total order, defined by Lamport clock.

If receive a message with time T, ping neighbors to get their current time (and/or any messages that are before T).

Eliminating Waiting

Can we apply messages without waiting to see if an “earlier” message will arrive?

If temporary inconsistency is ok:

- Keep timestamped log of all operations
- Apply operations in timestamp order.
- If an "older" update arrives later, insert in log and recompute.

Using Logical Clocks

Can we use logical clocks to implement distributed mutual exclusion?

- Example of state machine replication

Correctness:

- one at a time
- everyone in order of request
- Liveness: (if every holder releases lock) eventually requester gets the lock

Logical Clocks for Mutual Exclusion

Every message timestamped with value of logical clock.

1. To request lock, multicast request Tm to everyone
2. If get a lock request message, put Tm on local queue, ack (once it is ack'ed from everyone, request is in every queue)
3. To release the lock, remove request from local queue, multicast release message
4. If get release message, remove that request from local queue
5. Process i gets lock if its request < everyone else's, and none earlier can arrive
   – Process i has seen a later timestamp from everyone else

Example

Vector Clocks

Each process keeps an array of timestamps, VC[i]

VC[i] is the latest event I know from process i

Revised update rule, at node i:

- Pass vector clock in each message
- Update local clock on message receipt (VC[i]++)
- for all i, VC[j] = max (VC[j], msg's VC[j])

VC defines partial order of events

- don't process an event until you've seen all prior events from that process