6.824 2015 Lecture 3: Primary/Backup Replication

Today
- Replication
- Remus case study
- Lab 2 introduction

Fault tolerance
- we'd like a service that continues despite failures!
- available: still useable despite [some class of] failures
- correct: act just like a single server to clients
- very hard!
- very useful!

Need a failure model: what will we try to cope with?
- Independent fail-stop computer failure
  - Remus further assumes only one failure at a time
- Site-wide power failure (and eventual reboot)
- (Network partition)
- No bugs, no malice

Core idea: replication
- *Two* servers (or more)
- Each replica keeps state needed for the service
- If one replica fails, others can continue

Example: fault-tolerant MapReduce master
- lab 1 workers are already fault-tolerant, but not master
  - master is a "single point of failure"
- can we have two masters, in case one fails?
- [diagram: M1, M2, workers]
- state:
  - worker list
  - which jobs done
  - which workers idle
  - TCP connection state
  - program counter

Big Questions:
- What state to replicate?
- How does replica get state?
- When to cut over to backup?
- Are anomalies visible at cut-over?
- How to repair / re-integrate?

Two main approaches:
- State transfer
  - "Primary" replica executes the service
  - Primary sends [new] state to backups
- Replicated state machine
  - All replicas execute all operations
  - If same start state,
    - same operations,
    - same order,
    - deterministic,
    - then same end state

State transfer is simpler
- But state may be large, slow to transfer
- Remus uses state transfer

```
Replicated state machine can be more efficient
  If operations are small compared to data
  But complex, e.g. order on multi-core, determinism
  Labs use replicated state machines

Remus: High Availability via Asynchronous Virtual Machine Replication
NSDI 2008

Very ambitious system:
  Whole-system replication
  Completely transparent to applications and clients
  High availability for any existing software
  Would be magic if it worked well!
  Failure model:
    1. independent hardware faults
    2. site-wide power failure

Plan 1 (slow, broken):
  [diagram: app, O/S, Remus underneath]
  two machines, primary and backup; plus net and other machines
  primary runs o/s and application s/w, talks to clients, &c
  backup does *not* initially execute o/s, applications, &c
    it only executes some Remus code
  a few times per second:
    pause primary
    copy entire RAM, registers, disk to backup
    resume primary
  if primary fails:
    start backup executing!

Q: is Plan 1 correct?
   i.e. does it look just like a single reliable server?

Q: what will outside world see if primary fails and replica takes over?
   will backup have same state as last visible on primary?
   might a client request be lost? executed twice?

Q: is Plan 1 efficient?

Can we eliminate the fact that backup *state* trails the primary?
  Seems very hard!
  Primary would have to tell backup (and wait) on every instruction.

Can we *conceal* the fact that backup's state lags primary?
  Prevent outside world from *seeing* that backup is behind last primary state
    e.g. prevent primary sent RPC reply but backup state doesn't reflect that RPC
    e.g. MR Register RPC, which it would be bad for backup to forget
  Idea: primary "holds" output until backup state catches up to output point
    e.g. primary receives RPC request, processes it, creates reply packet,
    but Remus holds reply packet until backup has received corresponding state update

Remus epochs, checkpoints
  Clients:    C1
               req1                        reply1
  Primary:    ... E1 ... | pause |  E2  release   | pause |
                           ckpt        ok           ckpt
  Backup:      ... (E0) ... |  apply  | (E1)          |
  1. Primary runs for a while in Epoch 1, holding E1's output
  2. Primary pauses
  3. Primary sends RAM+disk changes to backup (in background)
```

```
    4. Primary resumes execution in E2, holding E2's output
    5. Backup copies all to separate RAM, then ACKs
    6. Primary releases E1's output
    7. Backup applies E1's changes to RAM and disk

  If primary fails, backup finishes applying last epoch's disk+RAM,
    then starts executing

  Q: any externally visible anomalies?
     lost input/output?
     repeated output?

  Q: what if primary receives+executes a request, crashes before checkpoint?
     backup won't have seen request!

  Q: what if primary crashes after sending ckpt to backup,
     but before releasing output?

  Q: what if client doesn't use TCP -- doesn't re-transmit?

  Q: what if primary fails while sending state to backup?
     i.e. backup is mid-way through absorbing new state?

  Q: are there situations in which Remus will incorrectly activate the backup?
     i.e. primary is actually alive
     network partition...

  Q: when primary recovers, how does Remus restore replication?
     needed, since eventually active ex-backup will itself fail

  Q: what if *both* fail, e.g. site-wide power failure?
     RAM content will be lost, but disks will probably survive
     after power is restored, reboot guest from one of the disks
       O/S and application recovery code will execute
     disk must be "crash-consistent"
       so probably not the backup disk if was in middle of installing checkpoint
     disk shouldn't reflect any held outputs (... why not?)
       so probably not the primary's disk if was executing
     I do not understand this part of the paper (Section 2.5)
       seems to be a window during which neither disk could be used if power failed
         primary writes its disk during epoch
         meanwhile backup applies last epoch's writes to its disk

  Q: in what situations will Remus likely have good performance?

  Q: in what situations will Remus likely have low performance?

  Q: should epochs be short or long?

  Remus Evaluation
    summary: 1/2 to 1/4 native speed
    checkpoints are big and take time to send
    output hold limits speed at which clients can interact

  Why so slow?
    checkpoints are big and take time to generate and send
      100ms for SPECweb2005 -- because many pages written
    so inter-checkpoint intervals must be long
    so output must be held for quite a while
    so client interactions are slow
      only 10 RPCs per second per client
```

```
How could one get better performance for replication?
  big savings possible with application-specific schemes:
    just send state really needed by application, not all state
    send state in optimized format, not whole pages
    send operations if they are smaller than state
  likely *not* transaparent to applications
    and probably not to clients either

PRIMARY-BACKUP REPLICATION IN LAB 2

outline:
  simple key/value database
    Get(k), Put(k, v), Append(k, v)
  primary and backup
  replicate by primary sending each operation to backups
  tolerate network problems, including partition
    either keep going, correctly
    or suspend operations until network is repaired
  allow replacement of failed servers
  you implement essentially all of this (unlike lab 1)

"view server" decides who p and b are
  main goal: avoid "split brain" -- disagreement about who primary is
  clients and servers ask view server
  they don't make independent decisions

repair:
  view server can co-opt "idle" server as b after old b becomes p
  primary initializes new backup's state

key points:
  1. only one primary at a time!
  2. the primary must have the latest state!
  we will work out some rules to ensure these

view server
  maintains a sequence of "views"
    view #, primary, backup
    0: -- --
    1: S1 --
    2: S1 S2
    4: S2 --
    3: S2 S3
  monitors server liveness
    each server periodically sends a Ping RPC
    "dead" if missed N Pings in a row
    "live" after single Ping
  can be more than two servers Pinging view server
    if more than two, "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
  OK to have a view with just a primary, and no backup
    but -- if an idle server is available, make it the backup

how to ensure new primary has up-to-date replica of state?
  only promote previous backup
  i.e. don't make an idle server the primary
  backup must remember if it has been initialized by primary
```

```
    if not, don't function as primary even if promoted!

Q: can more than one server think it is primary?
   1: S1, S2
      net broken, so viewserver thinks S1 dead but it's alive
   2: S2, --
   now S1 alive and not aware of view #2, so S1 still thinks it is primary
   AND S2 alive and thinks it is primary
   => split brain, no good

how to ensure only one server acts as primary?
  even though more than one may *think* it is primary
  "acts as" == executes and responds to client requests
  the basic idea:
    1: S1 S2
    2: S2 --
    S1 still thinks it is primary
    S1 must forward ops to S2
    S2 thinks S2 is primary
    so S2 must reject S1's forwarded ops

the rules:
  1. primary in view i must have been primary or backup in view i-1
  2. if you think you are primary, must wait for backup for each request
  3. if you think you are not backup, reject forwarded requests
  4. if you think you are not primary, reject direct client requests

so:
  before S2 hears about view #2
    S1 can process ops from clients, S2 will accept forwarded requests
    S2 will reject ops from clients who have heard about view #2
  after S2 hears about view #2
    if S1 receives client request, it will forward, S2 will reject
      so S1 can no longer act as primary
    S1 will send error to client, client will ask vs for new view,
       client will re-send to S2
  the true moment of switch-over occurs when S2 hears about view #2

how can new backup get state?
  e.g. all the keys and values
  if S2 is backup in view i, but was not in view i-1,
    S2 should ask primary to transfer the complete state

rule for state transfer:
  every operation (Put/Get/Append) must be either before or after state xfer
    == state xfer must be atomic w.r.t. operations
  either
    op is before, and xferred state reflects op
    op is after, xferred state doesn't reflect op, prim forwards op after state

Q: does primary need to forward Get()s to backup?
   after all, Get() doesn't change anything, so why does backup need to know?
   and the extra RPC costs time

Q: how could we make primary-only Get()s work?

Q: are there cases when the lab 2 protocol cannot make forward progress?
   View service fails
   Primary fails before new backup gets state
   We will start fixing those in lab 3
```