6.824 2020 Lecture 4: Primary/Backup Replication

Today
  Primary/Backup Replication for Fault Tolerance
  Case study of VMware FT, an extreme version of the idea

The topic is (still) fault tolerance
  to provide availability
  despite server and network failures
  using replication

What kinds of failures can replication deal with?
  "fail-stop" failure of a single replica
    fan stops working, CPU overheats and shuts itself down
    someone trips over replica's power cord or network cable
    software notices it is out of disk space and stops
  Maybe not defects in h/w or bugs in s/w or human configuration errors
    Often not fail-stop
    May be correlated (i.e. cause all replicas to crash at the same time)
    But, sometimes can be detected (e.g. checksums)
  How about earthquake or city-wide power failure?
    Only if replicas are physically separated

Is replication worth the Nx expense?

Two main replication approaches:
  State transfer
    Primary replica executes the service
    Primary sends [new] state to backups
  Replicated state machine
    Clients send operations to primary,
      primary sequences and sends to backups
    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 over network

Replicated state machine often generates less network traffic
  Operations are often small compared to state
  But complex to get right
  VM-FT uses replicated state machine, as do Labs 2/3/4

Big Questions:
  What state to replicate?
  Does primary have to wait for backup?
  When to cut over to backup?
  Are anomalies visible at cut-over?
  How to bring a replacement backup up to speed?

At what level do we want replicas to be identical?
  Application state, e.g. a database's tables?
    GFS works this way
    Can be efficient; primary only sends high-level operations to backup
    Application code (server) must understand fault tolerance, to e.g. forward op
stream
  Machine level, e.g. registers and RAM content?
    might allow us to replicate any existing server w/o modification!
    requires forwarding of machine events (interrupts, DMA, &c)
    requires "machine" modifications to send/recv event stream...

Today's paper (VMware FT) replicates machine-level state
  Transparent: can run any existing O/S and server software!
  Appears like a single server to clients

Overview
  [diagram: app, O/S, VM-FT underneath, disk server, network, clients]
  words:
    hypervisor == monitor == VMM (virtual machine monitor)
    O/S+app is the "guest" running inside a virtual machine
  two machines, primary and backup
  primary sends all external events (client packets &c) to backup over network
    "logging channel", carrying log entries
  ordinarily, backup's output is suppressed by FT
  if either stops being able to talk to the other over the network
    "goes live" and provides sole service
    if primary goes live, it stops sending log entries to the backup

VMM emulates a local disk interface
  but actual storage is on a network server
  treated much like a client:
    usually only primary communicates with disk server (backup's FT discards)
    if backup goes live, it talks to disk server
  external disk makes creating a new backup faster (don't have to copy primary's disk)

When does the primary have to send information to the backup?
  Any time something happens that might cause their executions to diverge.
  Anything that's not a deterministic consequence of executing instructions.

What sources of divergence must FT handle?

  Most instructions execute identically on primary and backup.
    As long as memory+registers are identical,
      which we're assuming by induction.
  Inputs from external world -- just network packets.
    These appear as DMA'd data plus an interrupt.
  Timing of interrupts.
  Instructions that aren't functions of state, such as reading current time.
  Not multi-core races, since uniprocessor only.

Why would divergence be a disaster?
  b/c state on backup would differ from state on primary,
    and if primary then failed, clients would see inconsistency.
  Example: GFS lease expiration
    Imagine we're replicating the GFS master
    Chunkserver must send "please renew" msg before 60-second lease expires
    Clock interrupt drives master's notion of time
    Suppose chunkserver sends "please renew" just around 60 seconds
    On primary, clock interrupt happens just after request arrives.
      Primary copy of master renews the lease, to the same chunkserver.
    On backup, clock interrupt happens just before request.
      Backup copy of master expires the lease.
    If primary fails, backup takes over, it will think there
      is no lease, and grant it to a different chunkserver.
      Then two chunkservers will have lease for same chunk.
  So: backup must see same events,
    in same order,
    at same points in instruction stream.

Each log entry: instruction #, type, data.

FT's handling of timer interrupts
  Goal: primary and backup should see interrupt at
        the same point in the instruction stream
  Primary:
    FT fields the timer interrupt
    FT reads instruction number from CPU
    FT sends "timer interrupt at instruction X" on logging channel
    FT delivers interrupt to primary, and resumes it
    (this relies on CPU support to interrupt after the X'th instruction)
  Backup:
    ignores its own timer hardware
    FT sees log entry *before* backup gets to instruction X
    FT tells CPU to interrupt (to FT) at instruction X
    FT mimics a timer interrupt to backup

FT's handling of network packet arrival (input)
  Primary:

```
    FT tells NIC to copy packet data into FT's private "bounce buffer"
    At some point NIC does DMA, then interrupts
    FT gets the interrupt
    FT pauses the primary
    FT copies the bounce buffer into the primary's memory
    FT simulates a NIC interrupt in primary
    FT sends the packet data and the instruction # to the backup
  Backup:
    FT gets data and instruction # from log stream
    FT tells CPU to interrupt (to FT) at instruction X
    FT copies the data to backup memory, simulates NIC interrupt in backup

Why the bounce buffer?
  We want the data to appear in memory at exactly the same point in
    execution of the primary and backup.
  Otherwise they may diverge.

Note that the backup must lag by one one log entry
  Suppose primary gets an interrupt, or input, after instruction X
  If backup has already executed past X, it cannot handle the input correctly
  So backup FT can't start executing at all until it sees the first log entry
    Then it executes just to the instruction # in that log entry
    And waits for the next log entry before resuming backup

Example: non-deterministic instructions
  some instructions yield different results even if primary/backup have same state
  e.g. reading the current time or cycle count or processor serial #
  Primary:
    FT sets up the CPU to interrupt if primary executes such an instruction
    FT executes the instruction and records the result
    sends result and instruction # to backup
  Backup:
    FT reads log entry, sets up for interrupt at instruction #
    FT then supplies value that the primary got

What about output (sending network packets)?
  Primary and backup both execute instructions for output
  Primary's FT actually does the output
  Backup's FT discards the output

Output example: DB server
  clients can send "increment" request
    DB increments stored value, replies with new value
  so:
    [diagram]
    suppose the server's value starts out at 10
    network delivers client request to FT on primary
    primary's FT sends on logging channel to backup
    FTs deliver request to primary and backup
    primary executes, sets value to 11, sends "11" reply, FT really sends reply
    backup executes, sets value to 11, sends "11" reply, and FT discards
    the client gets one "11" response, as expected

But wait:
  suppose primary crashes just after sending the reply
    so client got the "11" reply
  AND the logging channel discards the log entry w/ client request
    primary is dead, so it won't re-send
  backup goes live
    but it has value "10" in its memory!
  now a client sends another increment request
    it will get "11" again, not "12"
  oops

Solution: the Output Rule (Section 2.2)
  before primary sends output,
  must wait for backup to acknowledge all previous log entries

Again, with output rule:
  [diagram]
  primary:
    receives client "increment" request
    sends client request on logging channel
    about to send "11" reply to client
    first waits for backup to acknowledge previous log entry
    then sends "11" reply to client
  suppose the primary crashes at some point in this sequence
  if before primary receives acknowledgement from backup
    maybe backup didn't see client's request, and didn't increment
    but also primary won't have replied
  if after primary receives acknowledgement from backup
    then client may see "11" reply
    but backup guaranteed to have received log entry w/ client's request
    so backup will increment to 11

The Output Rule is a big deal
  Occurs in some form in all replication systems
  A serious constraint on performance
  An area for application-specific cleverness
    Eg. maybe no need for primary to wait before replying to read-only operation
  FT has no application-level knowledge, must be conservative

Q: What if the primary crashes just after getting ACK from backup,
   but before the primary emits the output?
   Does this mean that the output won't ever be generated?

A: Here's what happens when the primary fails and the backup goes live.
   The backup got some log entries from the primary.
   The backup continues executing those log entries WITH OUTPUT DISCARDED.
   After the last log entry, the backup goes live -- stops discarding output
   In our example, the last log entry is arrival of client request
   So after client request arrives, the client will start emitting outputs
   And thus it will emit the reply to the client

Q: But what if the primary crashed *after* emitting the output?
   Will the backup emit the output a *second* time?

A: Yes.
   OK for TCP, since receivers ignore duplicate sequence numbers.
   OK for writes to disk, since backup will write same data to same block #.

Duplicate output at cut-over is pretty common in replication systems
  Clients need to keep enough state to ignore duplicates
  Or be designed so that duplicates are harmless

Q: Does FT cope with network partition -- could it suffer from split brain?
   E.g. if primary and backup both think the other is down.
   Will they both go live?

A: The disk server breaks the tie.
   Disk server supports atomic test-and-set.
   If primary or backup thinks other is dead, attempts test-and-set.
   If only one is alive, it will win test-and-set and go live.
   If both try, one will lose, and halt.

The disk server may be a single point of failure
  If disk server is down, service is down
  They probably have in mind a replicated disk server

Q: Why don't they support multi-core?

Performance (table 1)
  FT/Non-FT: impressive!
    little slow down
  Logging bandwidth
    Directly reflects disk read rate + network input rate
    18 Mbit/s for my-sql
  These numbers seem low to me
    Applications can read a disk at at least 400 megabits/second
    So their applications aren't very disk-intensive
```

When might FT be attractive?
  Critical but low-intensity services, e.g. name server.
  Services whose software is not convenient to modify.

What about replication for high-throughput services?
  People use application-level replicated state machines for e.g. databases.
    The state is just the DB, not all of memory+disk.
    The events are DB commands (put or get), not packets and interrupts.
  Result: less fine-grained synchronization, less overhead.
  GFS use application-level replication, as do Lab 2 &c

Summary:
  Primary-backup replication
    VM-FT: clean example
  How to cope with partition without single point of failure?
    Next lecture
  How to get better performance?
    Application-level replicated state machines

VMware KB (#1013428) talks about multi-CPU support. VM-FT may have switched
from a replicated state machine approach to the state transfer approach, but
unclear whether that is true or not.

http://www.wooditwork.com/2014/08/26/whats-new-vsphere-6-0-fault-tolerance/

http://www-mount.ece.umn.edu/~jjyi/MoBS/2007/program/01C-Xu.pdf

http://web.eecs.umich.edu/~nsatish/abstracts/ASPLOS-10-Respec.html