```
6.824 2018 Lecture 4: Primary/Backup Replication

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

Fault tolerance
  we'd like a service that continues despite failures
  some ideal properties:
    available: still useable despite [some class of] failures
    strongly consistent: looks just like a single server to clients
    transparent to clients
    transparent to server software
    efficient

What failures will we try to cope with?
  Fail-stop failures
  Independent failures
  Network drops some/all packets
  Network partition

But not:
  Incorrect execution
  Correlated failures
  Configuration errors
  Malice

Behaviors
  Available (e.g. if one server halts)
  Wait (e.g. if network totally fails)
  Stop forever (e.g. if multiple servers crash)
  Malfunction (e.g. if h/w computes incorrectly, or software has a bug)

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 memory and stack
    CPU registers

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 up to speed?

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
  VM-FT uses replicated state machine

Replicated state machine can be more efficient
  If operations are small compared to data
  But complex to get right
  Labs 2/3/4 use replicated state machines

At what level to define a replicated state machine?
  K/V put and get?
    "application-level" RSM
    usually requires server and client modifications
    can be efficient; primary only sends high-level operations to backup
  x86 instructions?
    might allow us to replicate any existing server w/o modification!
    but requires much more detailed primary/backup synchronization
    and we have to deal with interrupts, DMA, weird x86 instructions

The design of a Practical System for Fault-Tolerant Virtual Machines
Scales, Nelson, and Venkitachalam, SIGOPS OSR Vol 44, No 4, Dec 2010

Very ambitious system:
  Goal: fault-tolerance for existing server software
  Goal: clients should not notice a failure
  Goal: no changes required to client or server software
  Very ambitious!

Overview
  [diagram: app, O/S, VM-FT underneath, shared disk, network, clients]
  words:
    hypervisor == monitor == VMM (virtual machine monitor)
    app and O/S are "guest" running inside a virtual machine
  two machines, primary and backup
  shared disk for persistent storage
    shared so that bringing up a new backup is faster
  primary sends all inputs to backup over logging channel

Why does this idea work?
  It's a replicated state machine
  Primary and backup boot with same initial state (memory, disk files)
  Same instructions, same inputs -> same execution
    All else being equal, primary and backup will remain identical

What sources of divergence must we guard against?
  Many instructions are guaranteed to execute exactly the same on primary and backup.
    As long as memory+registers are identical, which we're assuming by induction.
  When might execution on primary differ from backup?
  Inputs from external world (the network).
  Data read from storage server.
  Timing of interrupts.
  Instructions that aren't pure functions of state, such as cycle counter.
  Races.

Examples of divergence?
  They all sound like "if primary fails, clients will see inconsistent story from backup."
  Lock server grants lock to client C1, rejects later request from C2.
    Primary and backup had better agree on input order!
    Otherwise, primary fails, backup now tells clients that C2 holds the lock.
  Lock server revokes lock after one minute.
    Suppose C1 holds the lock, and the minute is almost exactly up.
    C2 requests the lock.
    Primary might see C2's request just before timer interrupt, reject.
    Backup might see C2's request just after timer interrupt, grant.
  So: backup must see same events, in same order, at same point in instruction stream.

Example: timer interrupts
  Goal: primary and backup should see interrupt at exactly the same point in execution
```

```
    i.e. between the same pair of executed instructions
  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 special support from CPU to count instructions, interrupt after X)
  Backup:
    ignores its own timer hardware
    FT sees log entry *before* backup gets to instruction X
    FT tells CPU to interrupt at instruction X
    FT mimics a timer interrupt, resumes backup

Example: disk/network input
  Primary and backup *both* ask h/w to read
    FT intercepts, ignores on backup, gives to real h/w on primary
  Primary:
    FT tells the h/w to DMA data into FT's private "bounce buffer"
    At some point h/w does DMA, then interrupts
    FT gets the interrupt
    FT pauses the primary
    FT copies the bounce buffer into the primary's memory
    FT simulates an interrupt to primary, resumes it
    FT sends the data and the instruction # to the backup
  Backup:
    FT gets data and instruction # from log stream
    FT tells CPU to interrupt at instruction X
    FT copies the data during interrupt

Why the bounce buffer?
  I.e. why wait until primary/backup aren't executing before copying the data?
  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 event (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 restarting backup

Example: non-functional instructions
  even if primary and backup have same memory/registers,
    some instructions still execute differently
  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:
    backup also interrupts when it tries to execute that instruction
    FT supplies value that the primary got

What about disk/network output?
  Primary and backup both execute instructions for output
  Primary's FT actually does the output
  Backup's FT discards the output

But: the paper's Output Rule (Section 2.2) says primary primary must
tell backup when it produces output, and delay the output until the
backup says it has received the log entry.

Why the Output Rule?
  Suppose there was no Output Rule.
  The primary emits output immediately.
  Suppose the primary has seen inputs I1 I2 I3, then emits output.
  The backup has received I1 and I2 on the log.
  The primary crashes and the packet for I3 is lost by the network.
```

```
    Now the backup will go live without having processed I3.
      But some client has seen output reflecting the primary having executed I3.
      So that client may see anomalous state if it talks to the service again.
    So: the primary doesn't emit output until it knows that the backup
      has seen all inputs up to that output.

  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 takes over.
     The backup got some log entries from the primary.
     The backup continues executing those log entries WITH OUTPUT SUPPRESSED.
     After the last log entry, the backup starts emitting output
     In our example, the last log entry is I3
     So after input I3, the client will start emitting outputs
     And thus it will emit the output that the primary failed to emit

  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 shared disk, since backup will write same data to same block #.

  Duplicate output at cut-over is pretty common in replication systems
    Not always possible for clients &c to ignore duplicates
    For example, if output is vending money from an ATM machine

  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 shared disk breaks the tie.
     Shared disk server supports atomic test-and-set.
     Only one of primary/backup can successfully 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.

  Shared storage is single point of failure
    If shared storage is down, service is down
    Maybe they have in mind a replicated storage system

  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.tomsitpro.com/articles/vmware-vsphere-6-fault-tolerance-multi-cpu,1-2439.html

https://labs.vmware.com/academic/publications/retrace
```