Fault Tolerant Virtual Machines.

→ In this paper, we see how replication can ensure fault tolerant. This goes in detail and is extremely detailed about how the replication is done.

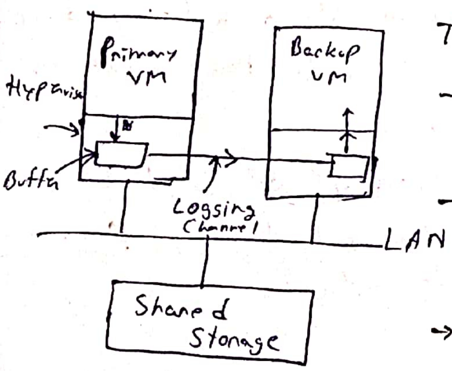
→ Replication helps with fail-stop failures i.e, type of failure which can be detected before the failing server causes an incorrect externally visible action Ex: fan broken → CPU overheat → Machine stops working.

→ ~~Bas~~ There are two approaches to share data between primary & backup: i) State Transfer → Entire state, i.e, RAM is shared with backup.
ii) Replicated State Machine → Operations are shared with backup.
The idea is if primary and backup start from same state and perform the same operation. The reach the same end state.

→ (i) is slower than (ii) due to large size & Network B/w but is simpler & robust.
→ (ii) → Less traffic, opⁿ are smaller, complex.

→ In this paper, the state we talk about is machine level (not like GFS) i.e, sharing registers, memory, interrupts

* Architecture:



Hypervisor = Virtual Machine Monitor is a emulator that emulates a OS and apps over a given hardware. The emulation is called Virtual Machine.

→ The main idea, there are two VMs which are Primary & Backup obviously running on different physical machines.

→ The primary and ~~bac~~ backup share the same ~~physical~~ storage service [we can consider it to be fault tolerant]

→ Only the primary receives external input and produces output.

→ The backup runs almost in sync with the primary with a bit of lag and does not produce outputs.

→ The backup receives its input from the logging channel. The primary sends all the operation that it is performing to the backup to ensure that it is in sync.

→ Not all operations are deterministic, something like interrupt, time stuff may be different in backup. The solution to this is, the backup perform these tasks but does not use the results. The primary sends the data of non deterministic operations to the backup and the backup just uses those results.

→ We follow the "output Rule" to ensure that no data is lost if the primary fails. Example: ~~Cli~~ Primary receives a INC opⁿ and the state was 10. If it inc and returns output then logs. Then there is a chance it fails after returning & before logging so backup gives 11 on inc which client has already got.

→ Main Questions:
(i) How non-determinism is handled?
(ii) How failure is handled like how to ensure no data is lost?
(iii) How to handle & detect failures?

(i) How non-determinism is handled? → As mentioned above [the first point of page 2]
↳ Each log entry is something like instruction #, type, data.

To Eliminate non determinism: The backup must see same events
i) in same order ii) same points in instruction stream.

Ex: FT's handling of timer interrupts:

Goal: Primary & Backup should see the interrupt at same point in instruction stream:

Primary:
i) fields the timer interrupt.
ii) reads instruction # from CPU
iii) sends "timer interrupt at inst. X" on logging channel
iv) delivers interrupt to primary & resumes it

Backup:
i) ignores its own timer hardware
ii) Sees log entry before backup gets to instruction X
iii) tells CPU to interrupt at inst X
iv) ~~Pri~~ uses the output result from ~~Primar~~ log entry.

Ex: FT's handling of network packet arrival (Input)

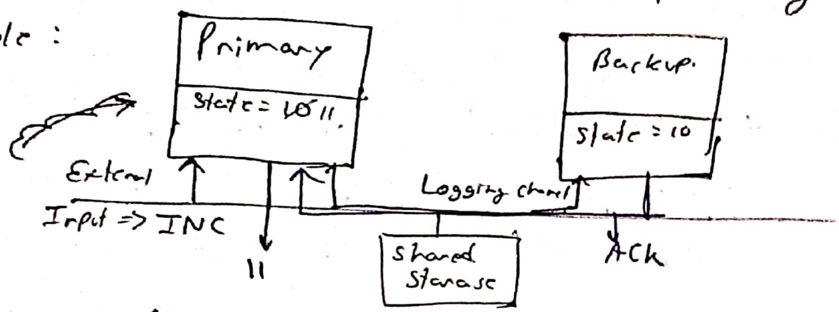
Primary: FT
i) tells NIC to copy packet data into FT's private "bounce buffer". → Discussed later.
ii) At some point NIC does DMA then interrupt
iii) FT gets the interrupt
iv) Pause primary
v) copies bounce buffer into primary memory
vi) simulates a NIC interrupt to Primary
vii) FT sends packet data & inst No to the backup.

Backup:
i) FT gets data & inst # from log stream
ii) FT tells CPU to interrupt (to FT) at inst Y.
iii) FT ~~Simulates~~ copies data to backup primary, simulates NIC interrupt in backup.

* The Output Rule:

→ The main idea is the
" The Primary VM may not send an output to the external world until the backup VM has received & acknowledged the log entry associated with the opⁿ producing output "

Example:



Sequence 1: i) Receives "INC" Input
ii) Performs Increment 10 → 11
If Primary fails here ← iii) Sends output → 11
iv) ~~fails~~ Sends logs entry to backup.

When backup takes over, its state will be 10. on receiving INC it will do 10 → 11 & return 11 → This is wrong ~~and~~ NOT WANTED.

Correct way: i) Receives "INC" Input
ii) Performs increment 10 → 11
iii) Sends log entry to backup. & waits for acknowledgement from backup.
iv) After ack; the 11 is sent as output.

* Detecting & Responding to failures:

* Detecting failures: i) By Exchanging UDP heart beat messages b/w primary & backup [for Primary side] [No response from ~~Primary~~ backup
ii) By Monitoring the logging traffic [for Backup Side] [No logging traffic]

Qn) what happens if backup fails? → The primary stops sending log entries & executes normally.

Qn) What happens if Primary fails? → The backup will have some log that it has received & acked. It will continue that execution on that is completed. It will execute normally, i.e, take input from network and produce output. For this transition from recording mode to go live mode. There is a change in network level. The MAC address of the new primary VM is automatically advertised on the network. So the physical network switches will know on what servers the primary VM is located.

Qn) What happens if network failure?

→ Like Primary is alive, the network fails so the backup is also going live. This is a split brain situation. To fix this, when ever primary &or backup tries to go live, it executes an atomic "test-and-set" operation on shared storage. If the operation succeeds, VM is allowed to go live. If it fails, the VM halts.

→ After this, a new backup VM is started and continues working normally.

* Additional Components:

i) Starting & Restarting FT VMs: It needs to be runnable from any arbitary state. They used VMotion which clones a VM to a remote host; sets up a logging channel; Source VM as primary and destination VM as ~~secon~~ backup. The whole thing is handled by a clustering service, it determines the best & server on which to run the backup VM based on resource usage & other constraints.

ii) Logging Channel: The hypervisor maintains a log buffer. P puts it in buffer & flushes asap to channel. The B consumes the entry from its buffer. ~~There can~~ The B buffer being empty is not a problem as it does not affect client. The buffer full on P affects performance. There can be mismatch in consumption rate:

P ——→ S

P: prod buffer rate x

S: buffer cons. rate Y

If $x > Y$ dynamic. Use a mech to reduce x to match Y.

If $x \leq Y$ → No problem.

iii) Special operations:

→ All opⁿ should be initiated on primary VM. Like CPU share ike, MMU change all should be done on P. The By log should be done on B.

iv) Issues for Disk IOs.

- Disk opⁿ are non blocking & can happen in ||. → Non determinism
- Disk access DMA directly to/from memory

One solution: i) Page Protection → Need to change MMU protection
↓
Expensive.

ii) Used bounce buffers:
↳ Temp buffer → same size as memory → accessed by disk operation.

→ A disk read opⁿ is modified to read the specified data ~~from~~ to bounce buffer & copy it to memory.

→ A disk write opⁿ, data sent to bounce buffer & disk write is modified to ~~read~~ write data from bounce buffer.

iii) Issues for Network Io:

Async Opⁿ → Non determinism.
i) Disable async ~~s~~ mode of opⁿ

More optimizations in paper but too much detail Section 3.5

** Alternative Design:

i) Shared v/s Non shared disk.

Usual, Only Primary ~~reads~~ writes ~~from~~ to disk → output logged to channel
→ wait for output rule
↓
Backup reads from there.

Alternative, if disk are not shared, B No need to wait for Outputrule.

Backup writes to its virtual disk.

→ useful when shared is too expensive or not accessible.

→ Disadv: i) The disks needs to be synced up when FT is enabled
ii) No shared ~~&~~ storage to write for split brain case so need a 3rd Party server here.

ii) Executing disk reads on ~~virtual~~ Backup VM. 6

→ Our design: Backup never reads from disk. → Reads from logging channel.

→ Alt design: Backup execute reads:

↳ 1 Eliminate the logging of disk read data (+)

2 Slow down backup VM's execution (−)

→ Extra retries must be done if Primary disk read succeeds but backup's disk read fails.

→ Disk read by Primary fails → contents ~~of~~ sent by logging since backup needs to undermine its read (~~to fail~~ purposely fail)

→ If Primary VM read & write to same location then the write must be delayed until backup performs ~~that~~ the read too. → Extra complexity.

→ This may be useful when bandwidth of logging channel is quite limited.