Fault Tolerant Virtual Machines

Main Structure

- Provides a way to keep a replica backup server so that if the main server fails, we have an exact copy we can switch to
- Two options for the level at which we want the replicas to be identical:
  - Application level state
    - Application has to be designed for fault tolerance
  - Machine level state
    - Allows us to replicate any application
    - Requires us to modify machines and forward all machine events
    - We'll focus on this
- Accomplished by using two VMs and keeping their state exactly synced by communicating inputs with each other
  - Keep them in **virtual lockstep**
- Some nuance required to ensure they stay exactly in state (i.e. communicating the results of non-deterministic inputs)
- Only the primary VM advertises its presence on the VM, takes inputs, and generates outputs
  - The secondary VM will process everything but its outputs will be sent to the void
- We sync nondeterministic events like virtual interrupts by logging the timing of these events / values and sending them with the inputs to the backup
  - These logged events are sent to the secondary VM through a logging channel
  - These include things such as reading current time, network packets, disk reads
    - Network packet arrival / disk reads trigger interrupts which have to be modeled precisely
- Both VMs share storage and only the primary will issue reads / writes to it (read data will be sent to the secondary through logs)
  - Alternatively, if we had the secondary also execute reads, this would force the primary VM to have to slow down to make sure the secondary VM finishes all of its operations before the primary continues
    - It results in lower throughput but greatly reduces bandwidth

Output Rule

- When a **failover** occurs, the backup VM takes over and will likely not process things the exact same way as the original
- However, all that matters is that what the new VM produces is consistent with the outputs that the original VM sent out
  - This includes both things sent to the client, disk writes, etc.
- What we need is that the secondary VM has to have received all log events before the primary VM outputs anything
  - That way, the previous outputs are still consistent
  - As a result, the primary VM just has to wait to output anything until the log is processed and acknowledged by the secondary
  - This doesn't mean that the primary VM has to stop running, it just delays its output
- It is possible that outputs are produced twice this way, but that's fine since TCP and network infrastructure can typically handle / filter duplicate packets

Detecting Failure

- The secondary VM will go live when it stops receiving heartbeats from the primary / logging traffic
- However, brief cuts in the network could lead to both attempting to go live

    - "Split brain" issue
- To do this, we make use of the shared storage and perform an atomic operation to try to get access to the storage
- Whichever one gets it first takes over and the other one kills itself

Other Considerations

- Need to make sure that the logging channel buffer does not grow too large or else there will be a noticeable delay between outputting and failover response
    - As a result, when the execution lag grows too large, we can apply a feedback loop to slow down the primary VM
- If using a non-shared disk, then we can still make it work by syncing disk state with the logging events
    - However adds extra complexity in keeping the disks in sync upon startup / failure
    - Also needs to use some external party to avoid split-brain situation