VMware FT FAQ

Q: The introduction says that it is more difficult to ensure deterministic execution on physical servers than on VMs. Why is this the case?

A: Ensuring determinism is easier on a VM because the hypervisor emulates and controls many aspects of the hardware that might differ between primary and backup executions, for example the precise timing of interrupt delivery.

Q: What is a hypervisor?

A: A hypervisor is part of a Virtual Machine system; it's the same as the Virtual Machine Monitor (VMM). The hypervisor emulates a computer, and a guest operating system (and applications) execute inside the emulated computer. The emulation in which the guest runs is often called the virtual machine. In this paper, the primary and backup are guests running inside virtual machines, and FT is part of the hypervisor implementing each virtual machine.

Q: Both GFS and VMware FT provide fault tolerance. How should we think about when one or the other is better?

A: FT replicates computation; you can use it to transparently add fault-tolerance to any existing network server. FT provides fairly strict consistency and is transparent to server and client. You might use FT to make an existing mail server fault-tolerant, for example. GFS, in contrast, provides fault-tolerance just for storage. Because GFS is specialized to a specific simple service (storage), its replication is more efficient than FT. For example, GFS does not need to cause interrupts to happen at exactly the same instruction on all replicas. GFS is usually only one piece of a larger system to implement complete fault-tolerant services. For example, VMware FT itself relies on a fault-tolerant storage service shared by primary and backup (the Shared Disk in Figure 1), which you could use something like GFS to implement (though at a detailed level GFS wouldn't be quite the right thing for FT).

Q: How do Section 3.4's bounce buffers help avoid races?

A: The problem arises when a network packet or requested disk block arrives at the primary and needs to be copied into the primary's memory. Without FT, the relevant hardware copies the data into memory while software is executing. Guest instructions could read that memory during the DMA; depending on exact timing, the guest might see or not see the DMA'd data (this is the race). It would be bad if the primary and backup both did this, but due to slight timing differences one read just after the DMA and the other just before. In that case they would diverge.

FT avoids this problem by not copying into guest memory while the primary or backup is executing. FT first copies the network packet or disk block into a private "bounce buffer" that the primary cannot access. When this first copy completes, the FT hypervisor interrupts the primary so that it is not executing. FT records the point at which it interrupted the primary (as with any interrupt). Then FT copies the bounce buffer into the primary's memory, and after that allows the primary to continue executing. FT sends the data to the backup on the log channel. The backup's FT interrupts the backup at the same instruction as the primary was interrupted, copies the data into the backup's memory while the backup is into executing, and then resumes the backup.

The effect is that the network packet or disk block appears at exactly
the same time in the primary and backup, so that no matter when they
read the memory, both see the same data.

Q: What is "an atomic test-and-set operation on the shared storage"?

A: The system uses a network disk server, shared by both primary and backup
(the "shared disk" in Figure 1). That network disk server has a
"test-and-set service". The test-and-set service maintains a flag that
is initially set to false. If the primary or backup thinks the other
server is dead, and thus that it should take over by itself, it first
sends a test-and-set operation to the disk server. The server executes
roughly this code:

```
  test-and-set() {
    acquire_lock()
    if flag == true:
      release_lock()
      return false
    else:
      flag = true
      release_lock()
      return true
```

The primary (or backup) only takes over ("goes live") if test-and-set
returns true.

The higher-level view is that, if the primary and backup lose network
contact with each other, we want only one of them to go live. The danger
is that, if both are up and the network has failed, both may go live and
develop split brain. If only one of the primary or backup can talk to
the disk server, then that server alone will go live. But what if both
can talk to the disk server? Then the network disk server acts as a
tie-breaker; test-and-set returns true only to the first call.

Q: How much performance is lost by following the Output Rule?

A: Table 2 provides some insight. By following the output rule, the
transmit rate is reduced, but not hugely.

Q: What if the application calls a random number generator? Won't that
yield different results on primary and backup and cause the executions
to diverge?

A: The primary and backup will get the same number from their random
number generators. All the sources of randomness are controlled by the
hypervisor. For example, the application may use the current time, or
a hardware cycle counter, or precise interrupt times as sources of
randomness. In all three cases the hypervisor intercepts the the
relevant instructions on both primary and backup and ensures they
produce the same values.

Q: How were the creators certain that they captured all possible forms
of non-determinism?

A: My guess is as follows. The authors work at a company where many
people understand VM hypervisors, microprocessors, and internals of guest
OSes well, and will be aware of many of the pitfalls. For VM-FT
specifically, the authors leverage the log and replay support from a
previous a project (deterministic replay), which must have already
dealt with sources of non-determinism. I assume the designers of
deterministic replay did extensive testing and gained experience
with sources of non-determinism that the authors of VM-FT use.

Q: What happens if the primary fails just after it sends output to the

external world?

A: The backup will likely repeat the output after taking over, so that it's generated twice. This duplication is not a problem for network and disk I/O. If the output is a network packet, then the receiving client's TCP software will discard the duplicate automatically. If the output event is a disk I/O, disk I/Os are idempotent (both write the same data to the same location, and there are no intervening I/Os).

Q: Section 3.4 talks about disk I/Os that are outstanding on the primary when a failure happens; it says "Instead, we re-issue the pending I/Os during the go-live process of the backup VM." Where are the pending I/Os located/stored, and how far back does the re-issuing need to go?

A: The paper is talking about disk I/Os for which there is a log entry indicating the I/O was started but no entry indicating completion. These are the I/O operations that must be re-started on the backup. When an I/O completes, the I/O device generates an I/O completion interrupt. So, if the I/O completion interrupt is missing in the log, then the backup restarts the I/O. If there is an I/O completion interrupt in the log, then there is no need to restart the I/O.

Q: How is the backup FT able to deliver an interrupt at a particular point in the backup instruction stream (i.e. at the same instruction at which the interrupt originally occured on the primary)?

A: Many CPUs support a feature (the "performance counters") that lets the FT VMM tell the CPU a number of instructions, and the CPU will interrupt to the FT VMM after that number of instructions.

Q: How secure is this system?

A: The authors assume that the primary and backup follow the protocol and are not malicious (e.g., an attacker didn't compromise the hypervisors). The system cannot handle compromised hypervisors. On the other hand, the hypervisor can probably defend itself against malicious or buggy guest operating systems and applications.

Q: Is it reasonable to address only the fail-stop failures? What are other type of failures?

A: It is reasonable, since many real-world failures are essentially fail-stop, for example many network and power failures. Doing better than this requires coping with computers that appear to be operating correctly but actually compute incorrect results; in the worst case, perhaps the failure is the result of a malicious attacker. This larger class of non-fail-stop failures is often called "Byzantine". There are ways to deal with Byzantine failures, which we'll touch on at the end of the course, but most of 6.824 is about fail-stop failures.