Remus: High Availability via Asynchronous Virtual Machine Replication

CS 736 - 4/13/17
Remus Design Goal:

Provide high availability against failures for unmodified applications

- Why high availability?
- What types of failures?
- Why unmodified applications?
Commercial High Availability Systems

Vendors:

[Stratus Technologies logo]

[Uptime. All the time.

[Tandem Computer Systems mug]
Commercial High Availability Systems

Implementation:
## Commercial High Availability Systems

### Implementation:

Supports high-volume transaction processing applications which require continuous availability:

- **Lockstepped system with fully redundant hardware:**
  Proprietary hardware and software enable fully redundant subsystems, which run in parallel, avoid performance degradation in the event of hardware failure.

- **OpenVOS operating system:**
  POSIX compliant fault-tolerant architecture specifically designed for rapid, high-volume transaction processing, maximizes availability and performance.

- **Active service architecture:**
  Self-monitoring and self-diagnosing solution automatically initiates the next day delivery of correct repair parts.

- **Hot swappable customer replaceable units (CRUs):**
  Subsystems that are designed to be replaced by IT generalists are automatically resynchronized when reinserted.

- **Secure kernel:**
  Proprietary, closed source kernel manages access to file system objects on a per-user basis.
# Commercial High Availability Systems

## Use Cases:

- Automation/IoT
- Finance/Banking
- Healthcare
- Manufacturing
- Government
- Telco’s
- Theme Parks (Disney)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Automation</td>
<td>Companies of all sizes rely on solutions from Stratus to ensure around the clock protection of their video monitoring, access control and other building security and automation systems.</td>
</tr>
<tr>
<td>Building Automation &amp; Security</td>
<td>Companies of all sizes rely on solutions from Stratus to ensure around the clock protection of their video monitoring, access control and other building security and automation systems.</td>
</tr>
<tr>
<td>Financial Services</td>
<td>Retail banking, corporate banking and capital markets companies alike depend on Stratus to guarantee 24/7/365 availability of their essential services and systems.</td>
</tr>
<tr>
<td>Healthcare</td>
<td>Hospitals, clinics and medical practices count on Stratus to ensure 24/7/365 access to clinical and administrative applications essential to safeguarding patient health and quality of care.</td>
</tr>
<tr>
<td>Manufacturing</td>
<td>Manufacturing companies across major industry sectors are trusting Stratus to prevent their MES, plant automation and data historian applications from downtime.</td>
</tr>
<tr>
<td>Public Safety &amp; Government</td>
<td>Public Safety Answering Points (PSAPs) and first responders rely on Stratus to ensure their most critical applications are available 24/7/365.</td>
</tr>
<tr>
<td>Retail</td>
<td>Leading retailers and distributors turn to Stratus to ensure availability of transaction processing, ordering and materials handling system.</td>
</tr>
<tr>
<td>Telecommunications</td>
<td>The world’s top network operators, service providers and solution developers depend on Stratus to provide around-the-clock availability of their applications to maximize revenues and reduce churn.</td>
</tr>
</tbody>
</table>
Cloud-Based High Availability Systems

Examples:

- Facebook
- Google Docs
- Bing
Cloud-Based High Availability Systems

Platform:

- Commodity server hardware (Intel, etc.)
- Commodity server software (Linux, Windows Server, etc.)
- Custom software services (GFS, HDFS, Hadoop, Bing, etc.)
Cloud-Based High Availability Systems

Infrastructure:

- Redundant compute (servers)
- Redundant multipath networks (fat tree)
- Redundant storage (GFS, HDFS)
- Redundant locations (multi-datacenter, PoPs)
Cloud-Based High Availability Systems

Software:

- Fully custom software
- Software load-balancing
- Software failure detection and retry/recovery
  - Software replica selection

Examples:

- GFS, HDFS, Hadoop, Spark, YARN, etc.
# High Availability Design Space:

<table>
<thead>
<tr>
<th>Custom software</th>
<th>Custom hardware</th>
<th>Commodity hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmodified</td>
<td>Commercial</td>
<td>Cloud-based</td>
</tr>
<tr>
<td></td>
<td>Remus</td>
<td></td>
</tr>
</tbody>
</table>
Hypervisor-Based High Availability

Big Idea:

- Run unmodified software (not fault tolerant)
- Run commodity server hardware (not fault tolerant)
- Use the hypervisor to transparently replicate state
- Goal: all observable output should be identical to running a single machine that does not fail
Hypervisor Availability Question:

What types of failures can be tolerated?

- Hardware?
- Hypervisor?
- OS?
- Application?
Hypervisor High Availability Strawman #1

Run app in a hypervisor using reliable shared storage:

- Restart elsewhere on crash
  - Just like local boot
- Live state is lost during crash
- Not transparent
Hypervisor High Availability Strawman #2

Run two VMs and duplicate all inputs to both VMs (identical inputs to both VMs)

- VM states will be identical if computation is deterministic

Challenge: determinism is hard

- Interrupts
- Multiprocessors
- Cache consistency
- Timestamps
Hypervisor High Availability Strawman #3

- Run a primary VM and a backup VM
- Only run compute on the primary VM
- Periodically replicate complete system state to the backup VM
- Block all output for an epoch until replication completes

This is the Remus approach!
Hypervisor High Availability Challenges

Primary challenge: Performance!

- VM memory replication
- Output blocking (delay)
- Frequent pausing (delay)
  - E.g., Do op -> replicate -> ACK -> release
  - Network latency bounds performance if the primary cannot execute ahead (asynchronously)
- Large data volume to replicate
  - E.g., Rehashing is a small operation that can change lots of data
Remus Overview

1. Run a primary app inside a VM
2. Periodically pause VM and send a snapshot to a backup
3. All output from primary is buffered until the snapshot completes
   a. Primary does not need to pause after every output (keeps executing)
4. Disk writes are immediately propagated to backup
   a. Disk writes are buffered until the next memory snapshot
5. The backup VM does not execute
Remus’ Failure Model

- Single machine (hardware) failures can be tolerated
  - All input from current epoch is lost!
    - Relies on TCP retransmissions
  - All progress from current epoch is lost!
- Dual failure requires reboot (just like a normal crash)
  - Persistent data is still not lost
- Output equivalence (but delayed)
Remus Implementation: Leverage Live Migration

- Traditional live migration
  - Stop VM at primary
  - Migrate state to backup
  - Start execution at backup
    - Spurious ARP to reprogram the network

- Remus migration
  - Pause VM at primary
  - Migrate state to backup
  - Restart VM at primary
Remus Implementation: Snapshots/Checkpoints

- Execution is divided into epochs. At the end of an epoch, do:
  a. Pause VM (End epoch i)
  b. Copy changed state into buffer
  c. Unpause VM (Start epoch i+1)
  d. Send buffer to backup
  e. Wait for ACK from backup to primary
  f. Release all output from epoch i
VM Memory snapshots

Tracking memory accesses

- Start by marking all guest pages as read-only
  - Shadow page tables
  - Nested page tables
- Page faults allow for tracking dirty set

Avoiding races:

- Live migration copies dirty pages in rounds to limit the pause time
- The VM must be paused for the final round
Remus Mem Snapshots

- Optimize communication path in Xen
- All of the guest memory is mapped before being copied
  - Other choice is to map dirty pages on demand
- Copy dirty memory to a staging buffer before sending over the network
  - Memory BW >> Network Bandwidth
Remus and Networking

- Buffer packets for output equivalence
  - Why?
- Hypervisor buffers packets in software
- Hypervisor copies packets
  - Costs CPU cycles and memory bandwidth
  - Reduces shared pages
- Buffered packets are lost on failure!
  - Is this ok?
  - What about packets that were received?
  - How does this impact the network?
Remus and Disks

- Disk writes are always mirrored
  - Need to recover on dual failure!
- All writes are write-through
  - memory -> disk +
    app -> network -> backup memory
  - Avoid backup being unable to catch up
- The backup buffers writes in memory until memory is snapshotted
- Only one disk is valid at any time
  - If the primary crashes mid-epoch, then the backup starts execution and then crashes, is either disk consistent?
Remus Failure Detection and Eventual Repair

Failure detection:
- Use a heartbeat
- Start VM on backup
- Migrate network connections to the backup
  - Spurious ARP

Primary (or backup) repair:
- All memory needs to be copied
- All disk needs to be copied

ARP Packet Format

*Note: The length of the address fields is determined by the corresponding address length fields*
Questions:

Do you believe that Remus is consistent after failure?

Do you believe it provides output equivalence?

- What happens if the primary crashes in the middle of releasing network packets?
- What happens to the disk when both the primary and backup crash?
Questions:

How should Remus be evaluated?

How can Remus hurt performance?
Evaluation (p1)
Evaluation (p2)
Evaluation (p3)
Known Optimizations

● Deadline scheduling
  ○ Lots of dirty pages -> long checkpoint
  ○ Avoid the primary “running ahead” by counting number of dirty pages

● Page compression
  ○ Copying entire pages for small changes is inefficient

● Copy-on-write checkpoints
  ○ Copy into Dom0/Hypervisor memory
**Issues**

- **One backup per server**
  - Can a single server backup multiple VMs?

- **Network topology**
  - Crossover cables between NICs literally implies that the server and backup are hardwired

- **Network Semantics**
  - Network consistency isn’t guaranteed
  - Dropping RX and TX packets is sometimes unacceptable
Middleboxes

Packets are the only input and output of middleboxes

- Packets cannot be dropped or duplicated to ensure output equivalence
- Middleboxes persist state across packets
- Middleboxes are extremely latency sensitive (10-100us)
Fault Tolerant Middleboxes

- Like Remus, FTMB relies on snapshotting
  - Waiting per-epoch to release packets is too high of latency!
- FTMB then replays the log to recover state changes from after most recent snapshot
- “must ensure that, before the middlebox transmits a packet \( p \), it has successfully logged to stable storage all the information needed to recreate internal state consistent with an execution that would have generated \( p \)”

Figure 2: Architecture for FTMB.
Highly Parallel Middleboxes

Multi-threads make logging plus replay difficult

Figure 1: Our model of a middlebox application

Figure 3: Four threads (black lines) process packets A, B, C, D. As time goes (left to right), they access (circles) shared variables X, Y, Z, T generating the PALs in parentheses. The red tree indicates the dependencies for packet B.
Fault Tolerant Middleboxes

- **Input Logger**
  - Save packets to enable replay when restoring snapshots

- **Master**
  - Modified middlebox code
  - Logs shared variable and non-determinism as PALs

- **Output Logger**
  - Releases packets in batches only after dependencies have been logged
  - Needed because master is highly parallel (multi-threaded)

- Loggers must “fail open” for this solution to work

**Figure 2: Architecture for FTMB.**