Rollback-Recovery for Middleboxes

Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, Scott Shenker
Middlebox Recovery: fail over to a back-up device after a middlebox goes offline, without interrupting connectivity or causing errors.
Key Challenge:
Correctness vs Performance
Systems Today: Correctness xor Performance

- Cold restart:
  - Fast, no overhead
  - Leads to buggy behaviour for stateful MBs, like missed attack detection

- Using snapshot/checkpoint:
  - Correctness guaranteed, no modification to MB
  - But adds latencies of 8-50ms; increases page loads by 200ms-1s

- Active-active implementation:
  - Cannot guarantee correctness either because of non-determinism
1980’s FT Research: “Output Commit”

Before releasing a packet: has all information reflecting that packet been committed to stable storage?

- Necessary condition for correctness.
- Typically implemented with check every time data is released.
- Middleboxes produce output every microsecond; release operates in parallel.
FTMB: “Fault-Tolerant Middlebox”

Correct Recovery and Performance

- Obeys output commit using ordered logging and parallel release.
- 30us latency overhead
- 5-30% throughput reduction
FTMB implements Rollback Recovery.

Three Part Algorithm:
- Snapshot
- Log
- Check

Input Logger  Master Middlebox  Output Logger
Rollback Recovery

Three Part Algorithm:
- Snapshot
- Log
- Check

Every k milliseconds, snapshot complete system state.
Rollback Recovery

Three Part Algorithm:

- Snapshot
- Log
- Check

Can now restore system to *stale state* at recovery time.
Rollback Recovery

Three Part Algorithm:
- **Snapshot**
- **Log**
- **Check**

Backup

Will restore last 100ms of system state using replay, which requires logging.

Input Logger

Master Middlebox

Output Logger
Rollback Recovery

Three Part Algorithm:
- **Snapshot**
- **Log**
- **Check**

Check to make sure we have all logged data required for replay at Output Logger.
Rollback Recovery

On Recovery, restore and replay.
Rollback Recovery

Three Part Algorithm:
- **Snapshot**
- **Log**
- **Check**

Snapshottting algorithms are well-known. We used VM checkpointing.
Rollback Recovery

Three Part Algorithm:

- **Snapshot**
- **Log**
- **Check**

Open Questions:

1. What do we need to log for correct replay?
   - A classically hard problem due to nondeterminism.

2. How do we check that we have everything we need to replay a given packet?
   - Need to monitor system state that is updated frequently and on multiple cores.
Quick Intro: Middlebox Architecture
Middlebox Architecture

Diagram showing an architecture with Input NIC, Core 1, Core 2, Core 3, Core 4, and Output NIC.
Middlebox Architecture

Input NIC “hashes” incoming packets to cores.

All packets from same flow are processed by same core.
Middlebox Architecture: State

Local state: only relevant to one connection.

Accessing local state is fast because only one core “owns” the data.
Middlebox Architecture: State

Total number of HTTP flows in the last hour

List of active connections permitted to pass.
Reading shared state is slower. Writing is most expensive because it can cause contention!
Rollback Recovery

Three Part Algorithm:

Snapshots

Log

Check

Open Questions:

(1) What do we need to log for correct replay?
   - A classically hard problem due to nondeterminism.

(2) How do we check that we have everything we need to replay a given packet?
   - Need to monitor system state that is updated frequently and on multiple cores.
Parallelism + Shared State

MB Rule: allow new connections, unless $A \geq 5$.

Total number of HTTP flows in the last hour

List of active connections permitted to pass.
Parallelism + Shared State

MB Rule: allow new connections, unless A>=5.
Parallelism + Shared State

MB Rule: allow new connections, unless $A \geq 5$. 
FTMB logs all accesses to shared state using **Packet Access Logs (PAL)**.
Parallelism + Shared State

Packet Access Log
- RED
  - accessed
  - A
  - FIRST

Packet Access Log
- BLACK
  - accessed
  - A
  - SECOND
Rollback Recovery

Open Questions:
(1) What do we need to log for correct replay?
   - Packet Access Logs record accesses to shared state.

(2) How do we check that we have everything we need to replay a given packet?
   - Need to monitor system state that is updated frequently and on multiple cores.
Checking for Safe Release

Input NIC

Core 1

Core 2

Core 3

Core 4

Output NIC

$fe

{34}

X

Y
Checking for Safe Release

Input NIC

$fe

Core 1

Core 2

Core 3

Core 4

X

Y

Output NIC

Output Logger
Checking for Safe Release

Do I have all PALs so that I can replay the system up to and including this packet?
If black packet were released now, would only need PAL \{X, Black, First\}
If blue packet were released now, would need its own PALs, and \{X, Black, First\}
Checking for Safe Release

Accessed Y after X!

Red packet needs its own PAL, and \{Blue, Y, First\}

…and \{Blue, X, 2nd\} and \{Black, X, First\}
Can depend on PALs from different cores, for variables packet never accessed!
Checking for Safe Release

Input NIC

Core 1

Core 2

Core 3

Core 4

Output NIC

Output Logger

$fe \{34\}

X

Y

FTMB is $O(\#\text{cores})$ and read-only, making it fast.
Key Insight: Packet cannot depend on a PAL that does not exist yet.
Key Insight: Packet cannot depend on a PAL that does not exist yet. PALs are written to output queues immediately when created.
When packet arrives at output queue, all PALs it depends on are already enqueued; or are already at output logger.
Ordered Logging and Parallel Release

What we want: “flush” all PALs to Output Logger. Then we’re done!

Problem: synchronizing behavior across all cores is expensive!
Ordered Logging and Parallel Release

Each core keeps a counter tracking the “youngest” PAL it has created. On release, packet *reads* counters across all cores. (O(#cores) reads)
Output logger keeps counter representing max PALs received.
Receive packet: reads marker to compare against other cores’ counters.
Ordered Logging and Parallel Release

If marker $\leq$ all counters, can release packet!
Master: Output PALs

[Pkt A]

Largest PAL sequence numbers are stored in dependency vector VOR_i for packet

Output Logger

10Gbps Ethernet

Pkt B

VOR_i compared against PALs at Output Logger

[56, 77, 63, 77]

[53, 76, 57, 75]

[45, 76, 60, 70] ≥?
Ordered Logging and Parallel Release

- Parallel! Threads are never blocked on each other to make progress.
- Cross-core accesses are *read only*.
  - Further amortized by batching.
- Linear: order # threads reads to perform.
- Fine-grained. Can make this decision with every packet release.
Rollback Recovery

Open Questions:

1. What do we need to log for correct replay?
   - Packet Access Logs record accesses to shared state.

2. How do we check that we have everything we need to replay a given packet?
   - Ordered logging and parallel release are read-only, O(#cores) cross core reads per release.
Recap

Three Part Algorithm:

1. **Snapshot**
   - VM Snapshots
2. **Log**
   - Ordered Logging and Parallel Release
3. **Check**

Backup

Input Logger

Master Middlebox

Output Logger
Replay

- Replica starts from the last available snapshot.
- The recorded packets are fed by the Input logger.
- The threads of the replica use the PALs to drive nondeterministic choices.
  - When acquiring the lock that protects a shred variable, the PALs come into play.
  - It checks whether it can access the lock, or it has to block waiting for some other thread that came earlier in the original execution.
- On output, packets are passed to the OL
  - The OL discards them if a previous instance and been already released.
- A thread exists replay mode when it finds that there are no more PALs for its shared variables.
The threads of the replica use the PALs to drive nondeterministic choices.

When acquiring the lock that protects a shred variable, the PALs come into play.

It checks whether it can access the lock, or it has to block waiting for some other threads that came earlier in the original execution.
Performance Highlights
Latency

<table>
<thead>
<tr>
<th>Latency</th>
<th>Throughput</th>
<th>Recovery Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remus [NSDI 2008]: 60,000us overhead</td>
<td>FTMB: 1.4-4Mpps 5-30% reduction over baseline throughput</td>
<td>FTMB: increases recovery time by 50-300ms. Still fast enough not to trigger TCP timeouts or errors!</td>
</tr>
<tr>
<td>Colo [SOCC 2013]: 8000us overhead</td>
<td>None higher than 200kpps</td>
<td>100s of ms</td>
</tr>
<tr>
<td>Pico [SOCC 2013]: 8000us overhead</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Latency under HA Solutions

Median latency: 70us

CDF of Packets

Latency (us)
Latency under HA Solutions

![Graph showing latency distributions and checkpointing times.]

- **App-Layer Checkpointing:** 8.5 ms
- **VM Checkpointing:** 50 ms

**Graph Legend:**
- Blue: MazuNAT (Baseline)
- Yellow: Pico (reported)
- Red: MazuNAT, under Remus
Latency under HA Solutions

CDF of Packets

FTMB: 100us

Latency (us)

MazuNAT (Baseline)
MazuNAT, with FTMB and Snapshots
Pico (reported)
MazuNAT, under Remus
Latency

CDF of Packets

Latency (us)

MazuNAT (Baseline) with I/O Loggers
w/ FTMB w/o Snapshots
w/ FTMB + Snapshots

Tail comes from periodic VM suspension for snapshots
5%-30% overhead, depending on the application.
Thank you!

**FTMB: Correct Recovery and Performance**

- Obeys output commit using ordered logging and parallel release.
- **30us** latency overhead
- **5-30%** throughput reduction