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

Systems Research: Enforce Correctness
- Adds latencies of 8-50ms; increases page loads by 200ms-1s.

Networking Research: Choose Performance
- “Cold Reset”
  - User applications exhibit buggy behavior.
  - Attacks go undetected.

Industrial systems: Awkward Compromise
- No guarantee of correct recovery = same troubles as cold reset.
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.

Input Logger

Master Middlebox

Output Logger
Rollback Recovery

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

Backup

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

Input Logger

Master Middlebox

Output Logger
Rollback Recovery

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

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

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

Three Part Algorithm:

1. **Snapshot**
2. **Log**
3. **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

Input NIC “hashes” incoming packets to cores.

All packets from same flow are processed by same core.
Local state: only relevant to one connection.

Accessing local state is *fast* because only one core “owns” the data.
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:
- **Snapshot**
- **Log**
- **Rollback**

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\geq5.
Parallelism + Shared State

MB Rule: allow new connections, unless $A \geq 5$. 
FTMB logs all accesses to shared state using Packet Access Logs.
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
Checking for Safe Release

Input NIC

$fe

Core 1

Core 2

Core 3

Core 4

{34}

X

Y

Output NIC

Output Logger
Do I have all PALs so that I can replay the system up to and including this packet?
Checking for Safe Release

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

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

...and \{Blue, X, 2nd\} and \{Black, X, First\}
Checking for Safe Release

Can depend on PALs from different cores, for variables packet never accessed!
Classical Dependency Tracking Approaches: \(O(#\text{cores} \times # \text{variables})\) cross core reads and writes

FTMB: \(O(#\text{cores})\) cross core reads
Checking for Safe Release

FTMB is $O(#\text{cores})$ and read-only, making it fast.
Ordered Logging and Parallel Release

Key Insight: Packet cannot depend on a PAL that does not exist yet.
Ordered Logging and Parallel Release

PALs are written to output queues immediately when created.
Ordered Logging and Parallel Release

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!
Each core keeps a counter tracking the “youngest” PAL it has created. On release, packet reads counters across all cores. (O(#cores) reads)
Ordered Logging and Parallel Release

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 <= all counters, can release packet!
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

Three Part Algorithm:

- Snapshot
- Log
- Check

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, $O(#\text{cores})$ cross core reads per release.
Recap

Three Part Algorithm:
1. **Snapshot**
2. **Log**
3. **Check**

- **VM Snapshots**
- **Ordered Logging and Parallel Release**
What I’m not talking about today

❖ Our prototype based on Xen, Click, and DPDK.
❖ Our tool to instrument shared state with PALs automatically.
❖ How replay works in detail.
❖ Alternative output commit approaches.
❖ Integration with commercial IDS systems.
❖ See our paper or come talk to me after talk!
Performance Highlights
Latency

Remus [NSDI 2008]: 50,000us overhead
Colo [SOCC 2013]: 50,000us overhead
Pico [SOCC 2013]: 8000us overhead
FTMB: 30us overhead

Throughput

None higher than 200kpps
FTMB: 1.4-4Mpps
5-30% reduction over baseline throughput

Recovery Time

100s of ms
FTMB: increases recovery time by 50-300ms.
Still fast enough not to trigger TCP timeouts or errors!
FTMB Outside of Academia

AT&T has submitted FTMB to ETSI: founders of NFV

FTMB is in trials at two major NFV & IDS vendors.
Thank you!

FTMB: Correct Recovery and Performance

- Obeys output commit using ordered logging and parallel release.

- 30us latency overhead
- 5-30% throughput reduction

I am on the job market for a tenure-track faculty position!

justine@eecs.berkeley.edu | @justinesherry
Backupland
Recovery Guarantees

If the loggers and backup go down, Master remains online with no downtime.

If Master and one more server fail, system cannot recover.
Can we mask downtime to users?
and 115 seconds respectively. Applications do not sensitive to the precise value.

Is stateful failover valuable to applications?

To see the impact of stateless recovery on real applications, we tested several applications over the wide area with a NAT to access the same variable, the frequency of output (for us, order of messages at a given process vs. threads that ‘race’ to access the same variable). The literature includes a wide range of protocol proposals (we refer the reader to Elnozahy et al. [18] for an excellent survey); however, to our knowledge, there is no available system implementation that we can put to the test for our application context. The literature includes a wide range of protocol proposals (we refer the reader to Elnozahy et al. [18] for an excellent survey). The literature includes a wide range of protocol proposals (we refer the reader to Elnozahy et al. [18] for an excellent survey).

We briefly discuss the three lines of work relevant to FTMB, found in the literature. The second are solutions for rollback recovery from the state of non-determinism (primarily the arrival and sending of messages) and where the per-packet for us), and where the per-output commit. We discussed these solutions in broad terms and compared FTMB to them experimentally.

The final class of solutions are the multicore record-and-replay. The only application with little adoption due to the complexity of its algorithm. This approach and compared FTMB to them experimentally.