Rollback-Recovery for Middleboxes

Justine Sherry, SIGCOMM 2015

Basic Concepts

Middlebox: a network appliance owning functions other than packet forwarding.

Rollback recovery: restart from recent saved state on a back-up device

Correctness: Output Commit

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

Implemented with check every time packet is released.

Gb/s flows trigger frequent output commit

Performance vs. Correctness

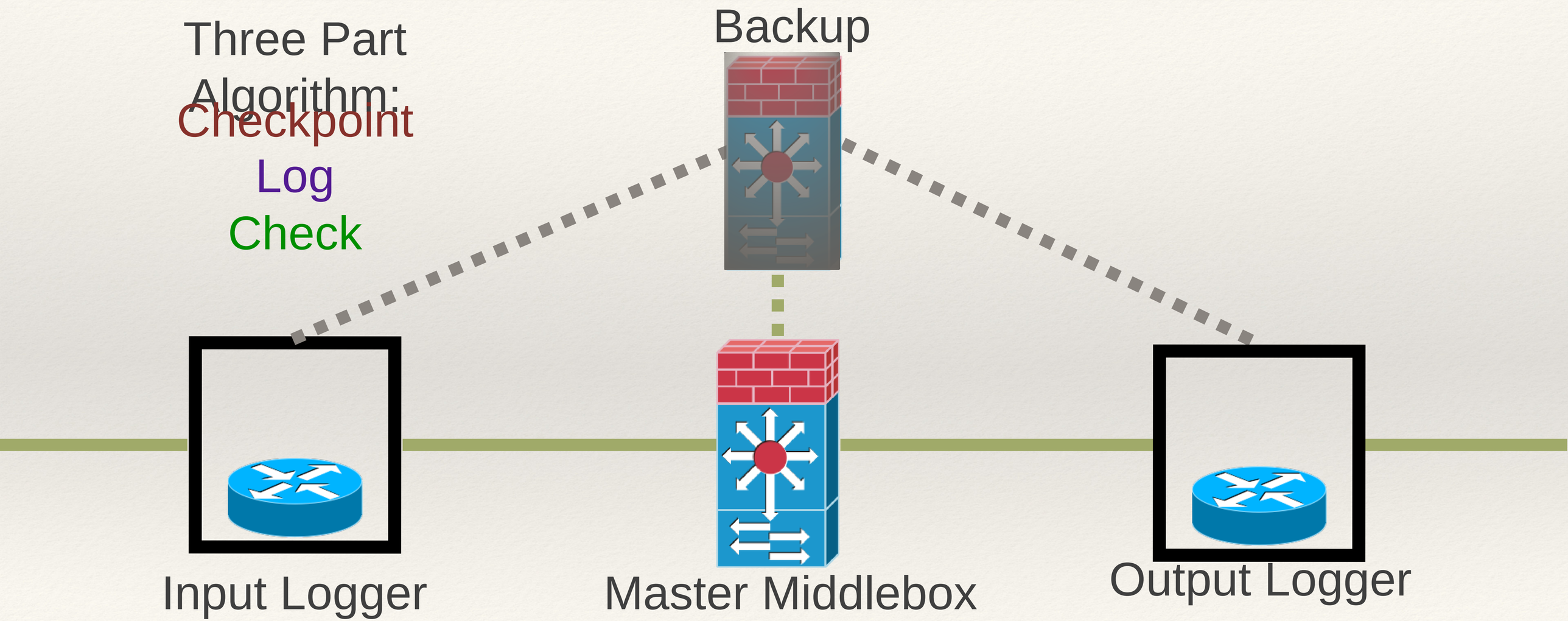
FTMB Implements Rollback Recovery.

Three Part Algorithm:

Checkpoint

Log

Check



Rollback Recovery

Three Part Algorithm:

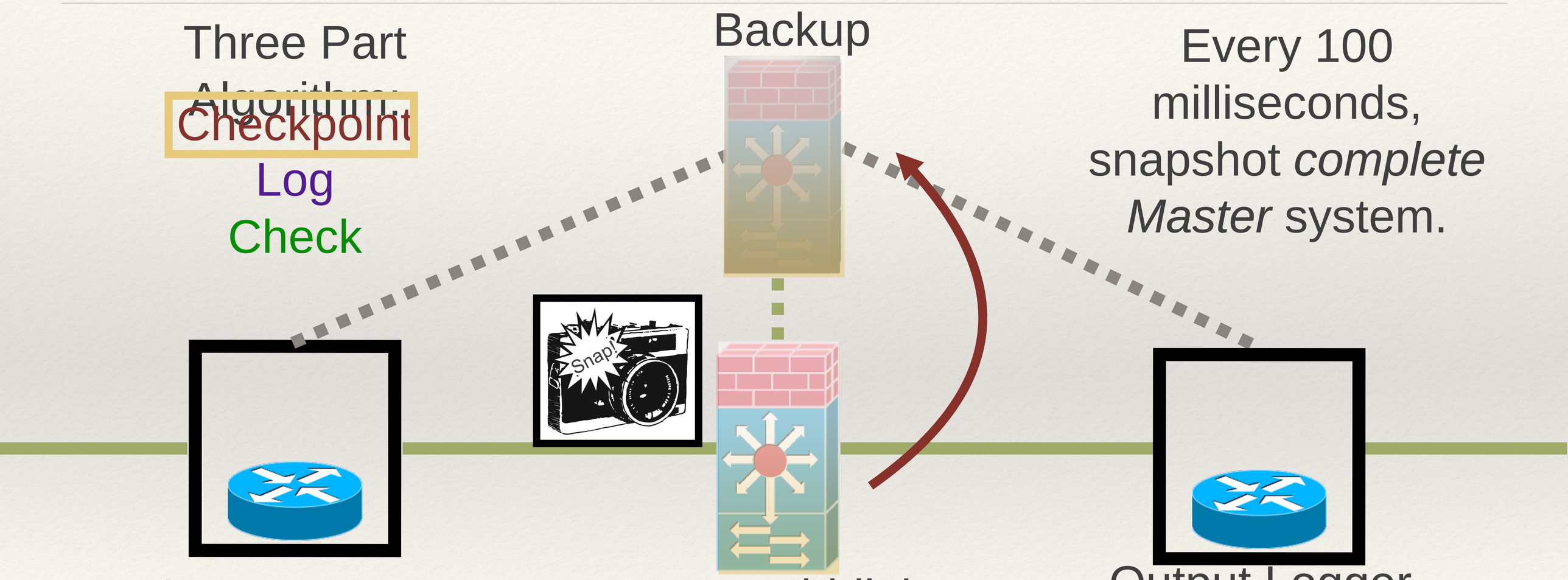
Checkpoint

Log

Check

Backup

Every 100 milliseconds, snapshot *complete Master* system.



Input Logger

Master Middlebox

Output Logger

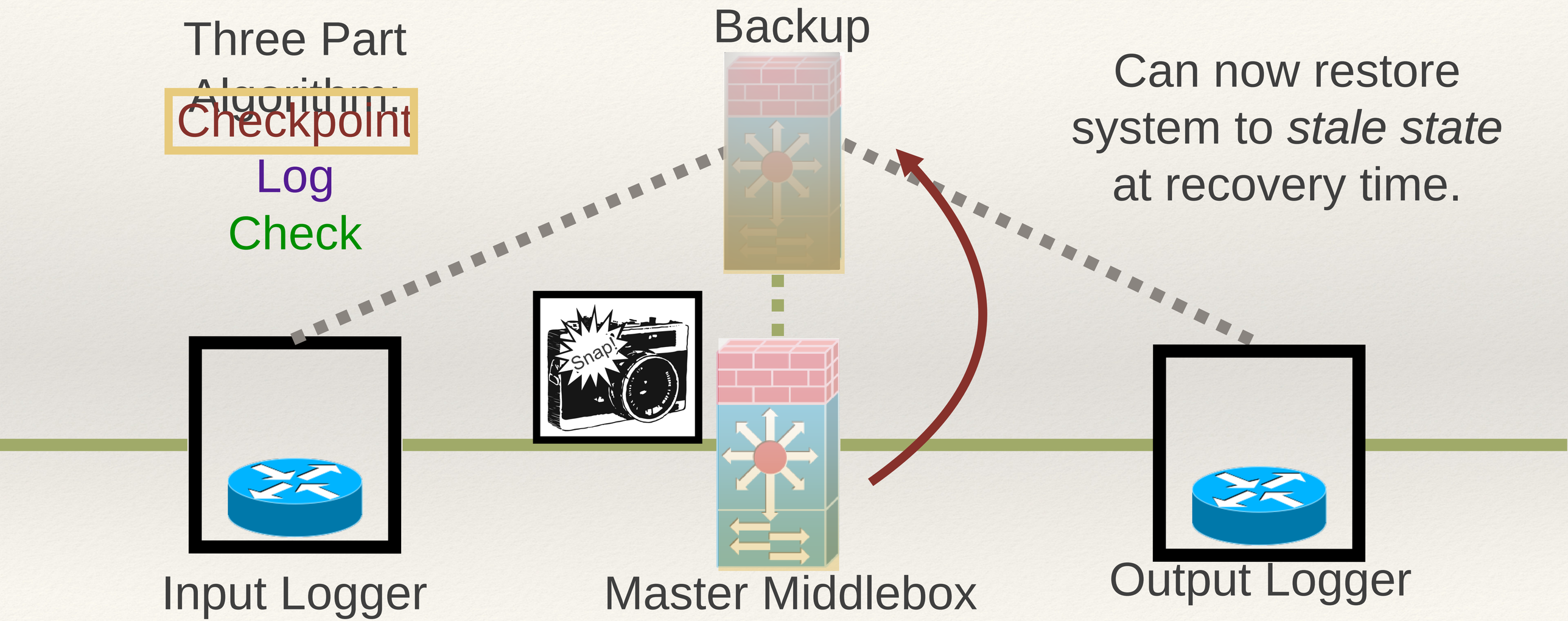
Rollback Recovery

Three Part Algorithm:

Checkpoint

Log

Check



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

Rollback Recovery

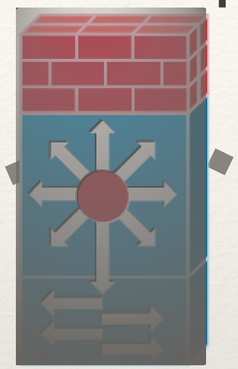
Three Part Algorithm:

Checkpoint

Log

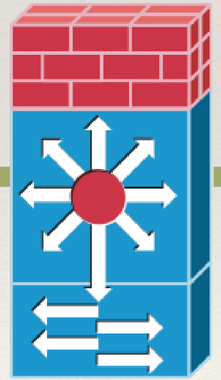
Check

Backup



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

Input Logger

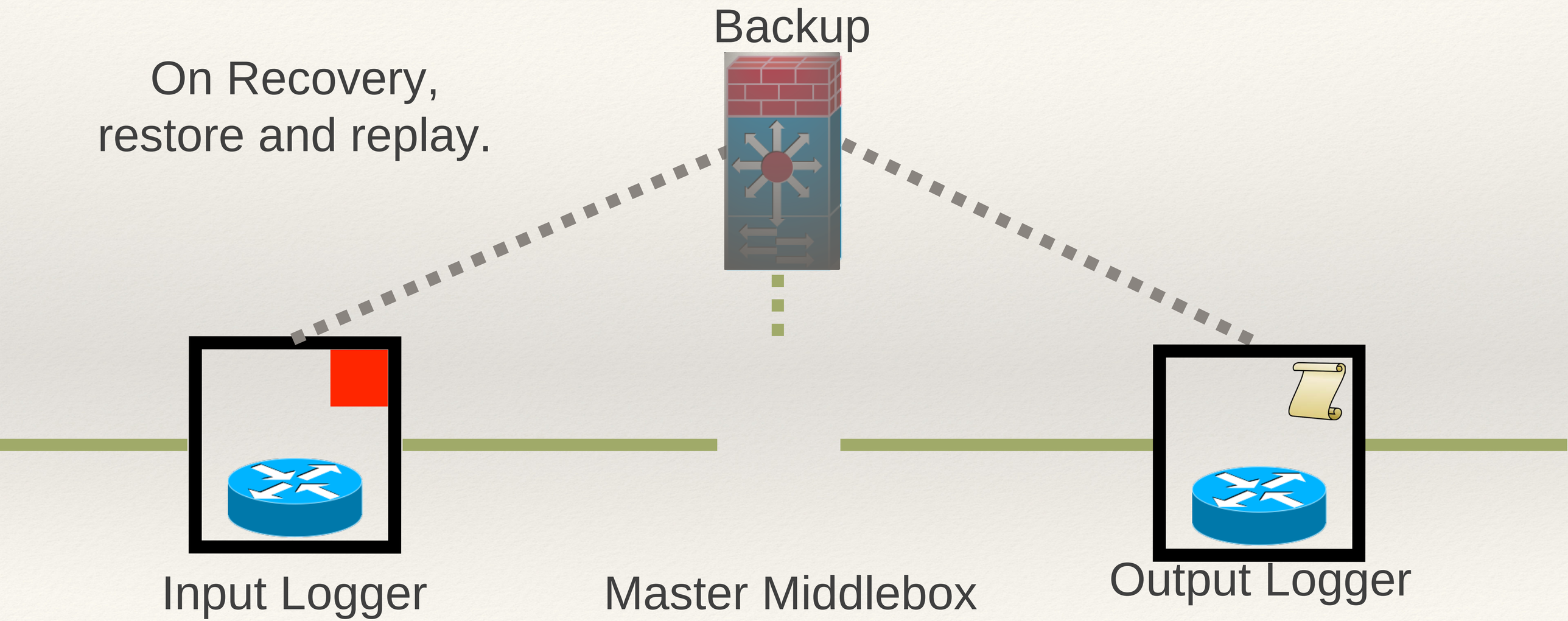


Master Middlebox

Output Logger

Rollback Recovery

On Recovery,
restore and replay.



Rollback Recovery

Three Part Algorithm:

Checkpoint

Log

Check

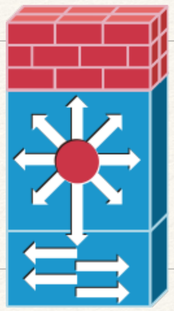
Open Questions:

(1) What do we need to log for correct replay?

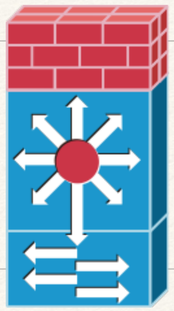
- A classically hard problem due to nondeterminism.

(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.



Middlebox Architecture



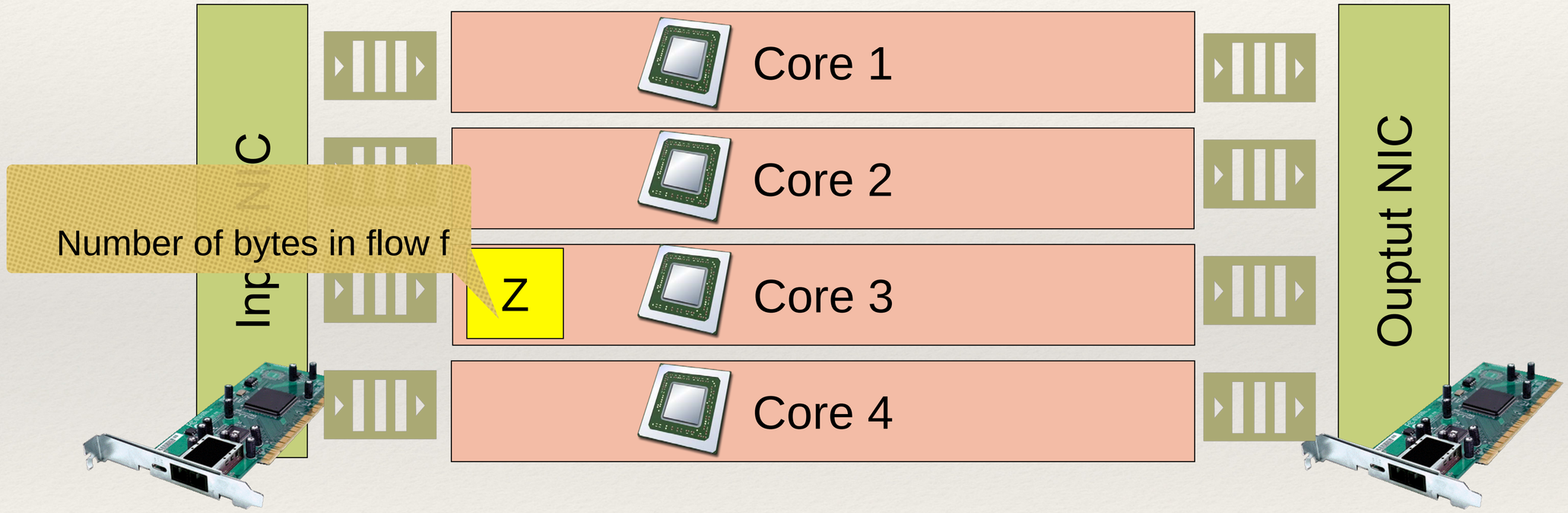
Middlebox Architecture

Input NIC “hashes” incoming packets to cores.

All packets from same flow are processed by same core.

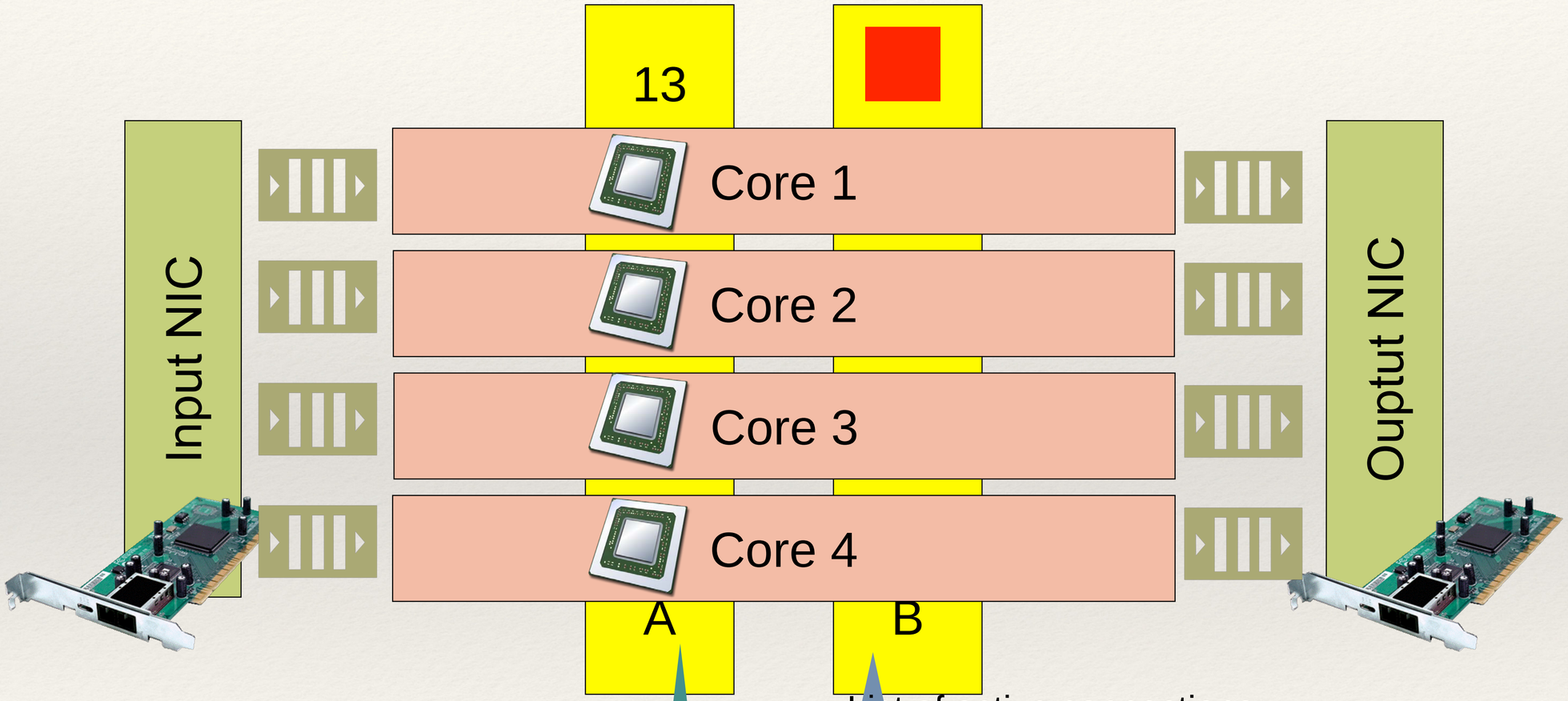
Middlebox Architecture: State

Local state: only relevant to one connection.

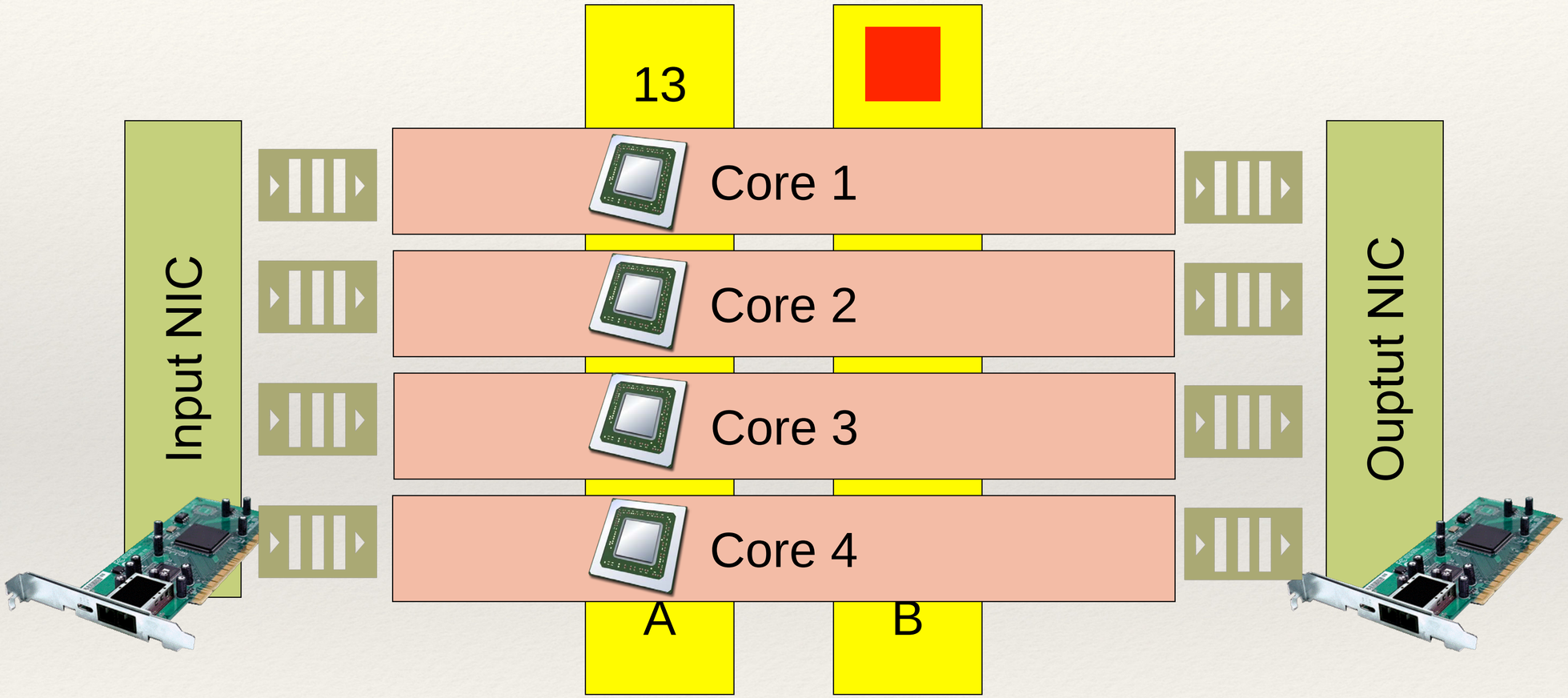


Accessing local state is *fast* because only one core “owns” the data.

Middlebox Architecture: State



Middlebox Architecture: State



Reading shared state is slower.
Writing is most expensive because it can cause *contention!*

Rollback Recovery

Three Part Algorithm:

Checkpoint

Log

ck

Open Questions:

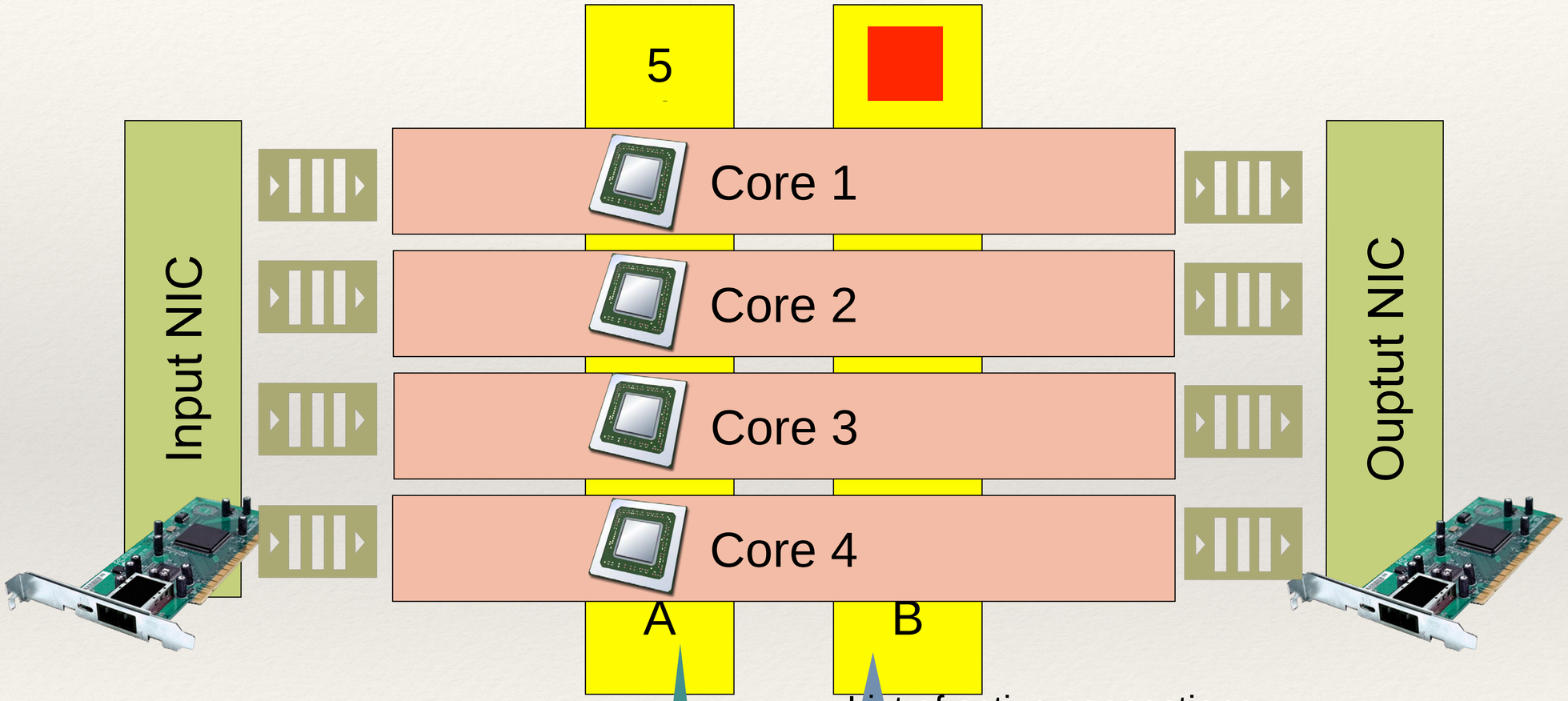
(1) What do we need to log for correct replay?

- A classically hard problem due to nondeterminism.

(2) How do we check that we have everything we need to replay a given packet?

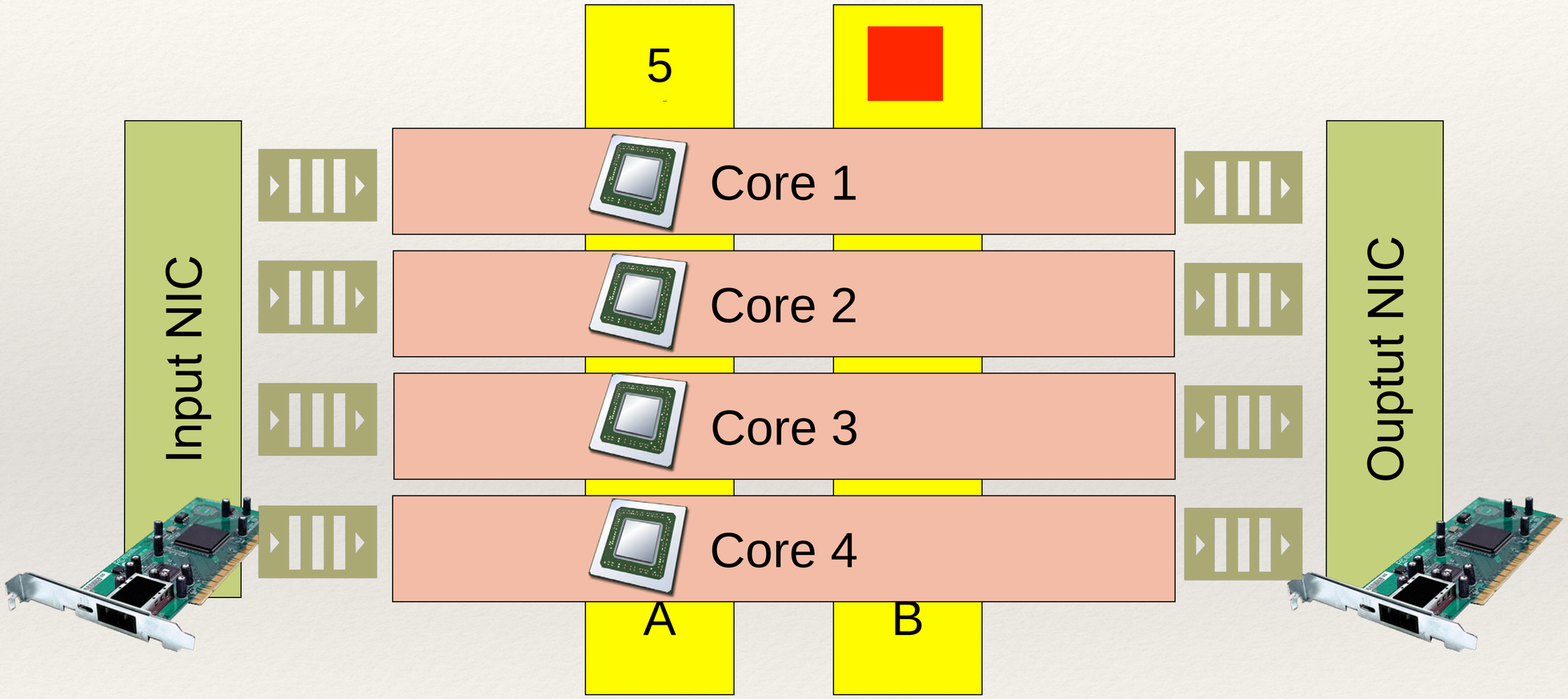
- Need to monitor system state that is updated frequently and on multiple cores.

Parallelism + Shared State



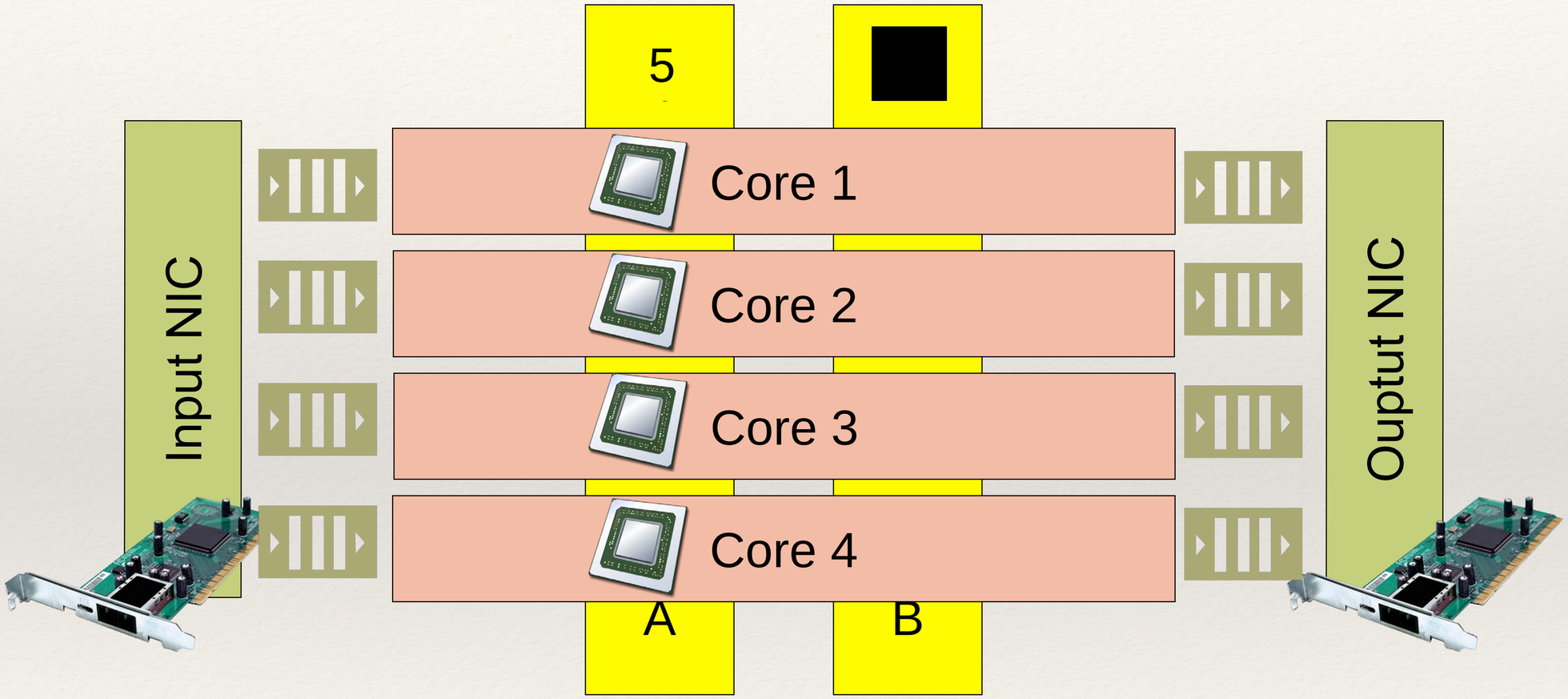
ew connections when A<5.

Parallelism + Shared State



MB Rule: allow new connections, unless A>=5.

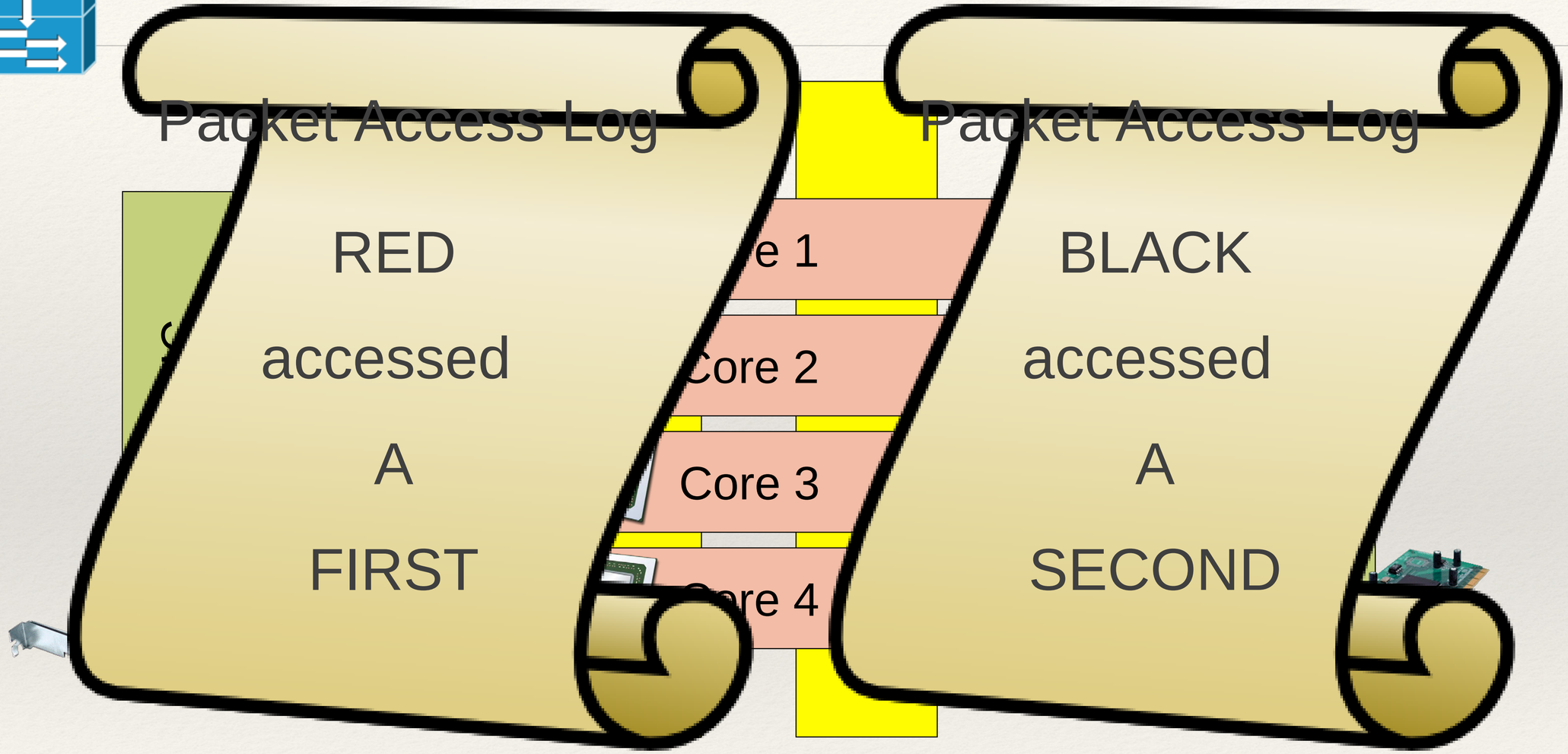
Parallelism + Shared State



MB Rule: allow new connections, unless A>=5.

FTMB logs all accesses to shared state using Packet Access Logs.

Parallelism + Shared State



Rollback Recovery

Three Part Algorithm:

Checkpoint

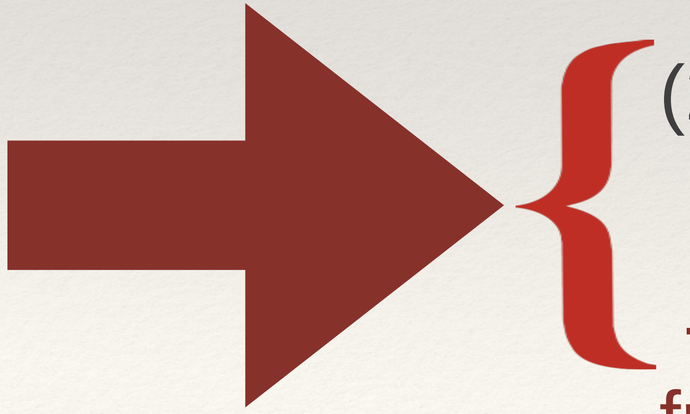
Log

Check

Open Questions:

(1) What do we need to log for correct replay?

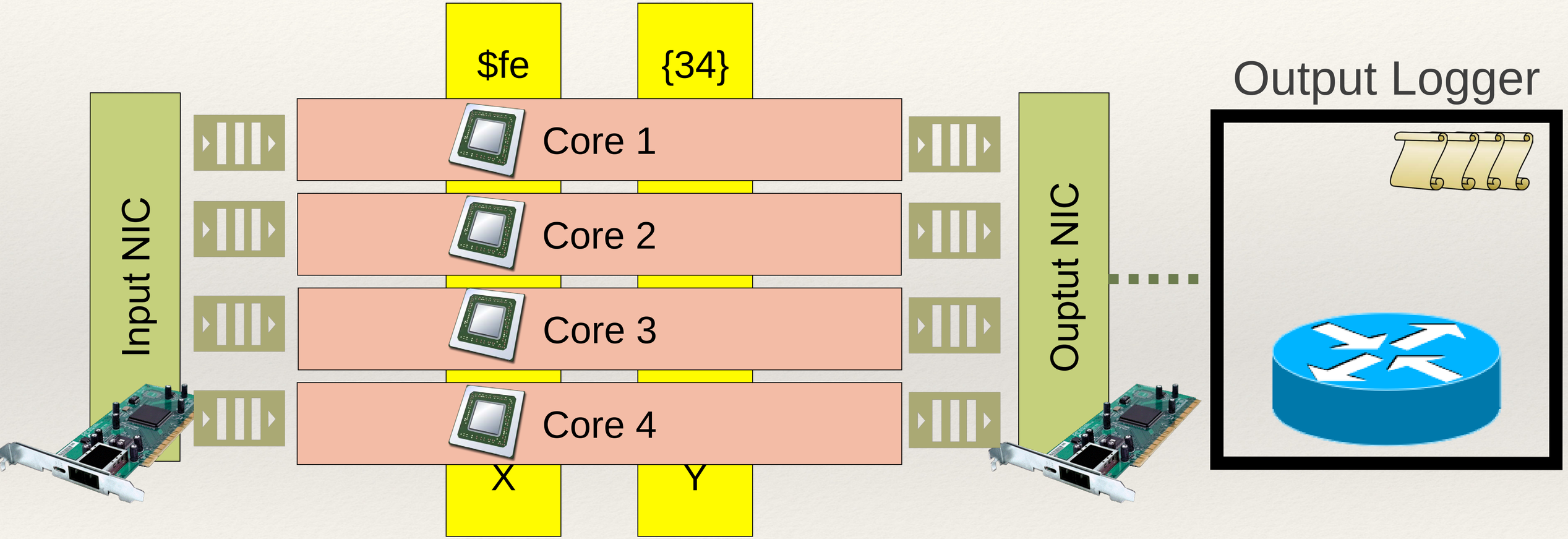
- Packet Access Logs record accesses to shared state.



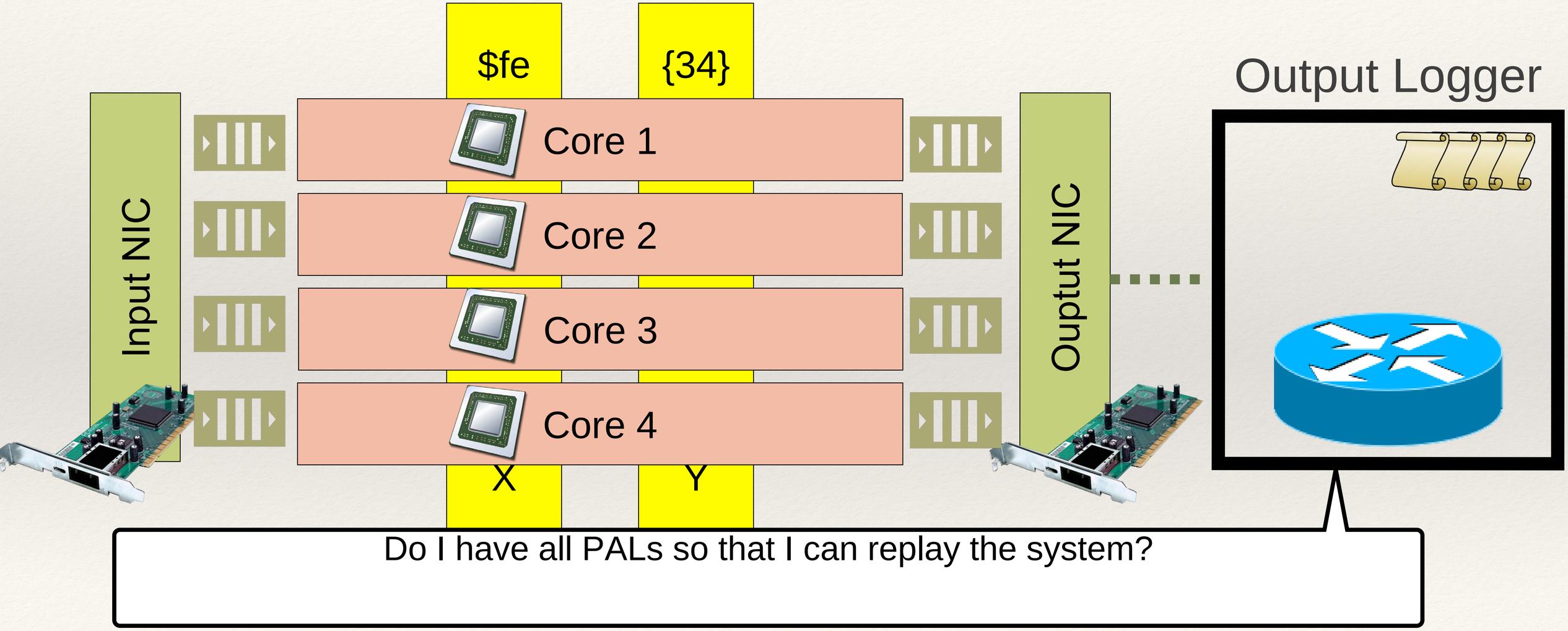
(2) How do we check that we have everything we need to replay a given packet?

- Need to monitor system state that is updated frequently and on multiple cores.

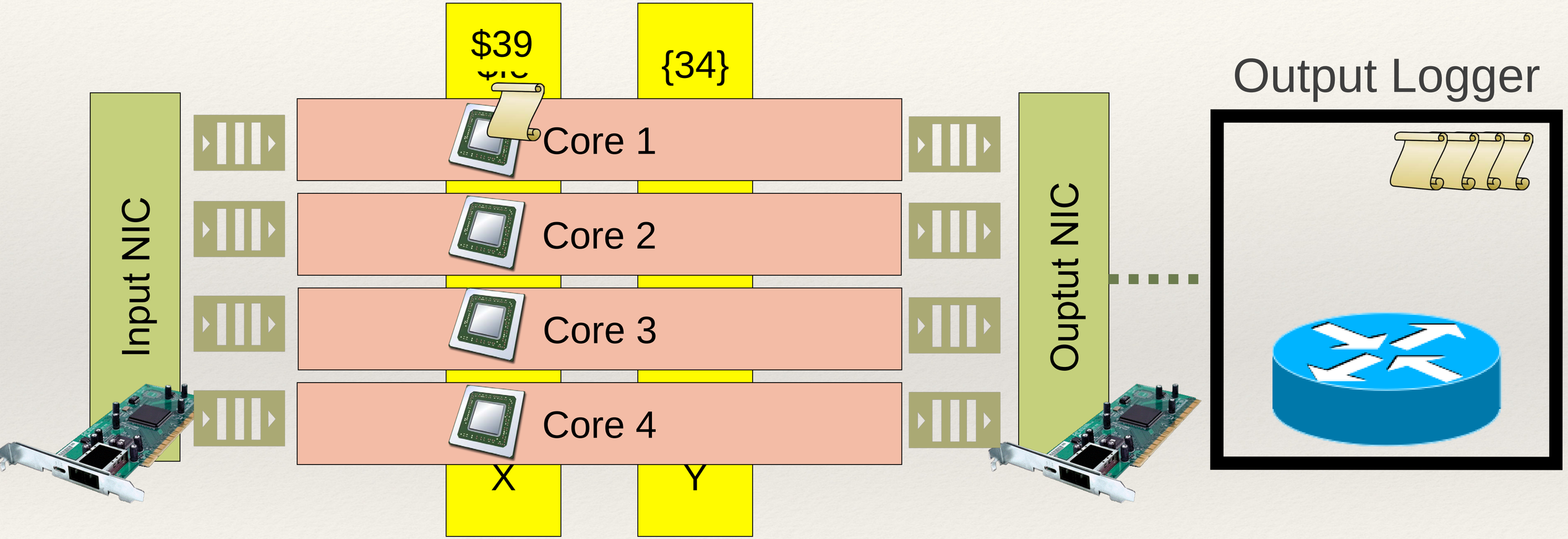
Checking for Safe Release



Checking for Safe Release



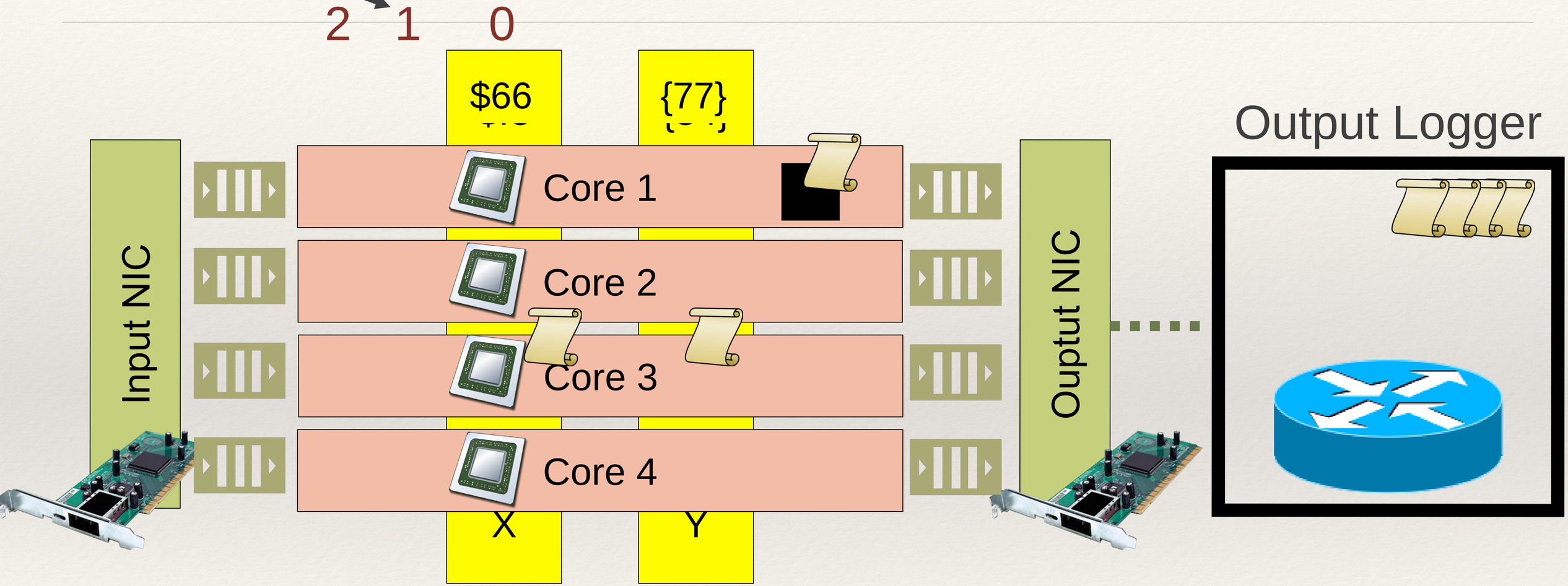
Checking for Safe Release



If black packet were released now, would only need PAL {X, Black, Firs

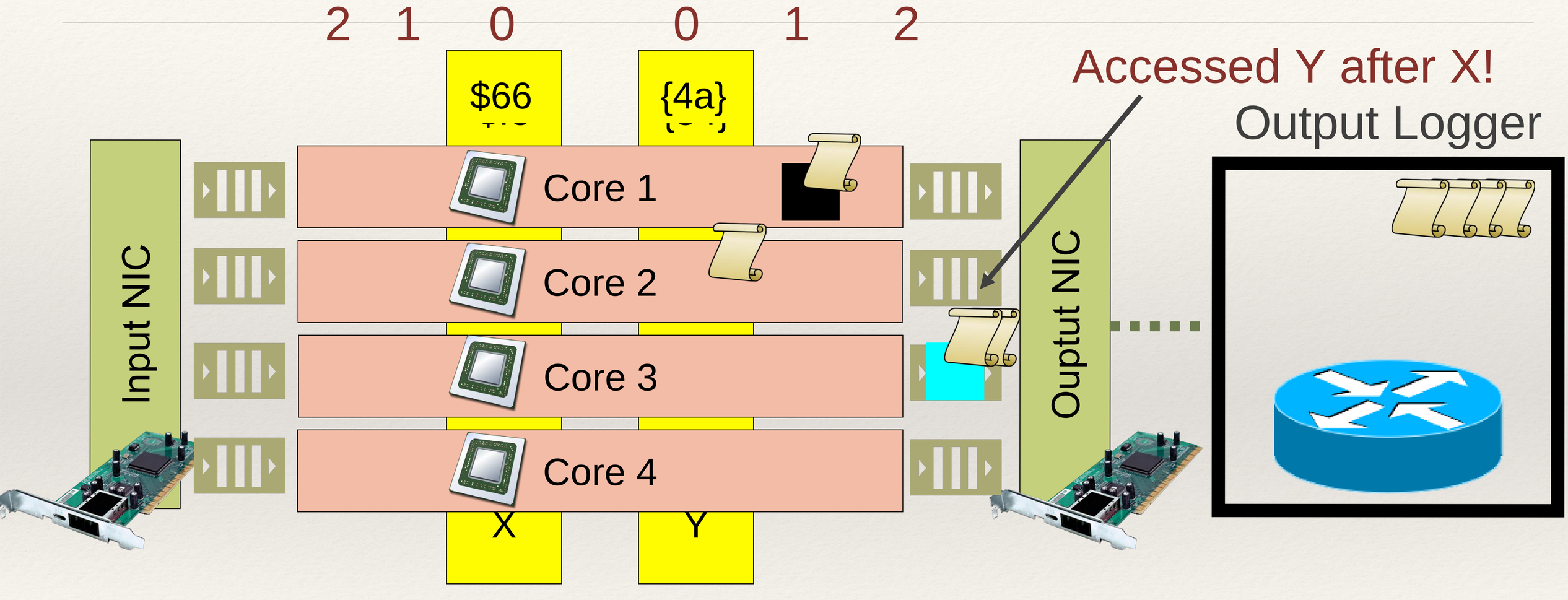
Checking for Safe Release

Need to read!



If blue packet were released now, would need its own PALs, and {X, Black

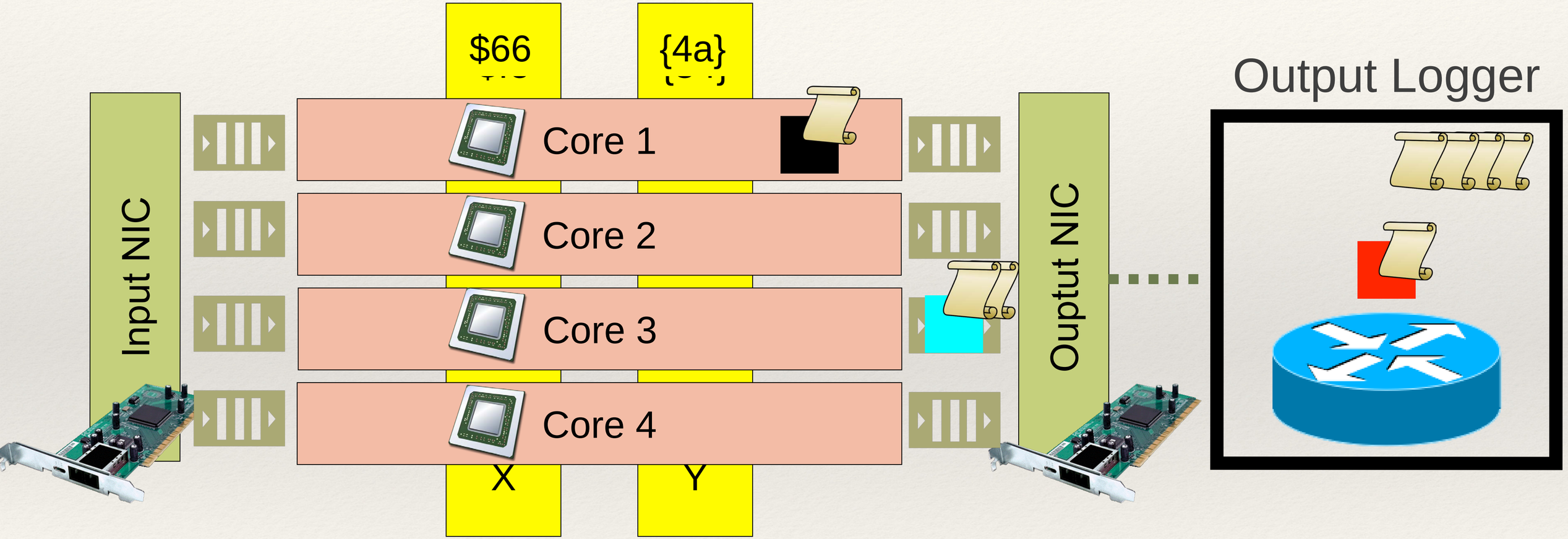
Checking for Safe Release



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

…and {Blue, X, 2nd} and {Black, X, Fi

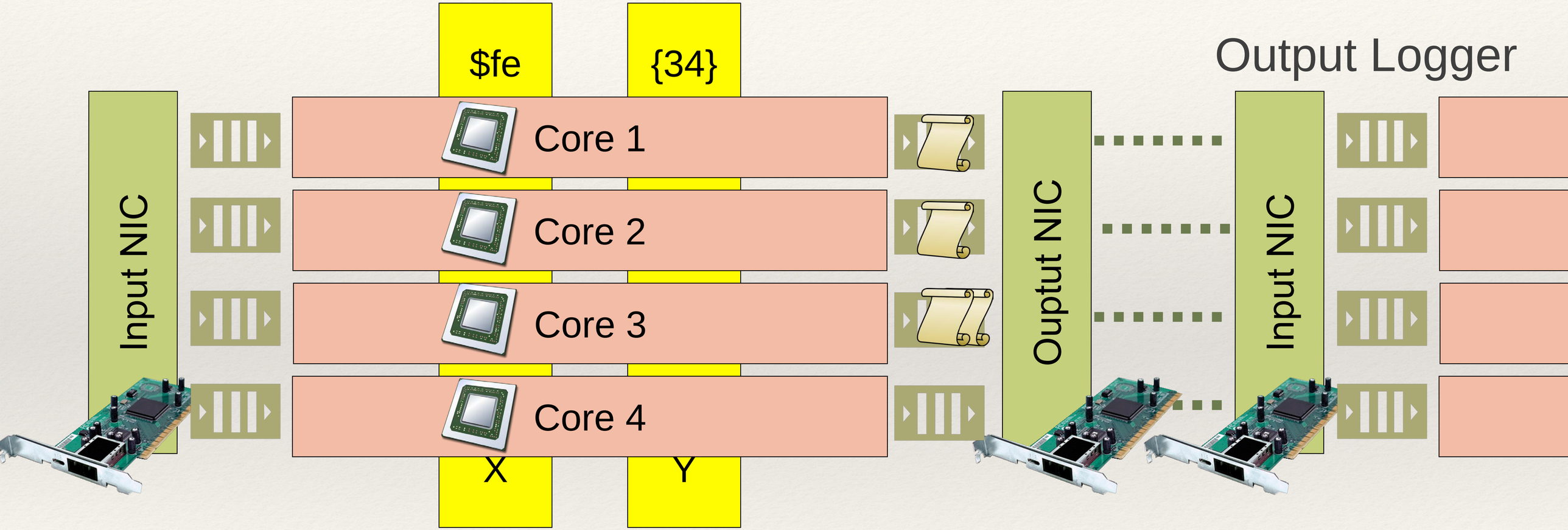
Checking for Safe Release



Can depend on PALs from different cores & variables, causing a lot of ex

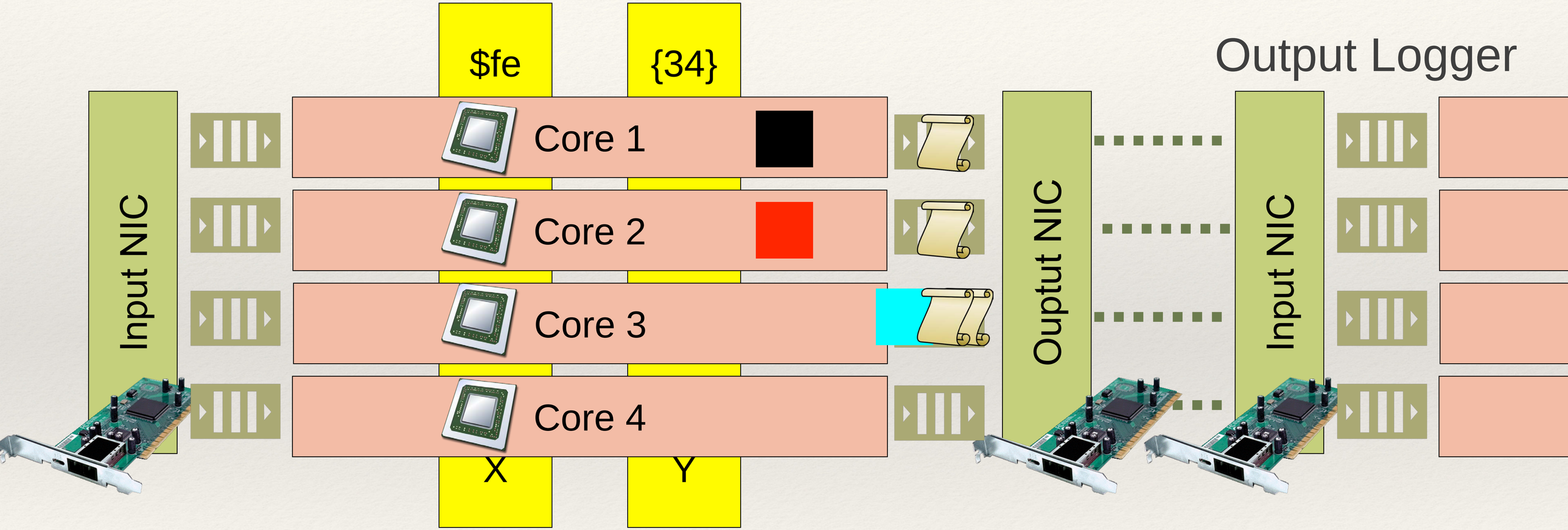
Ordered Logging and Parallel Release

Ordered Logging



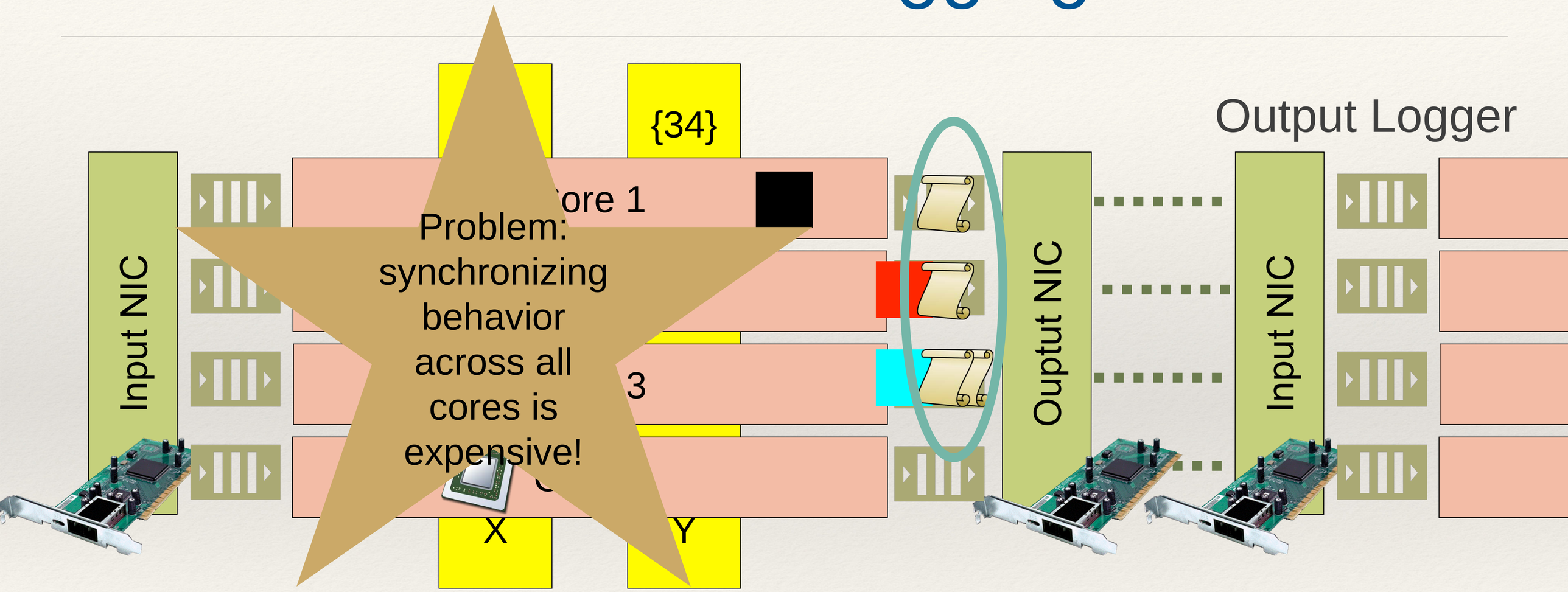
PALs are written to output queues immediately when created.

Ordered Logging



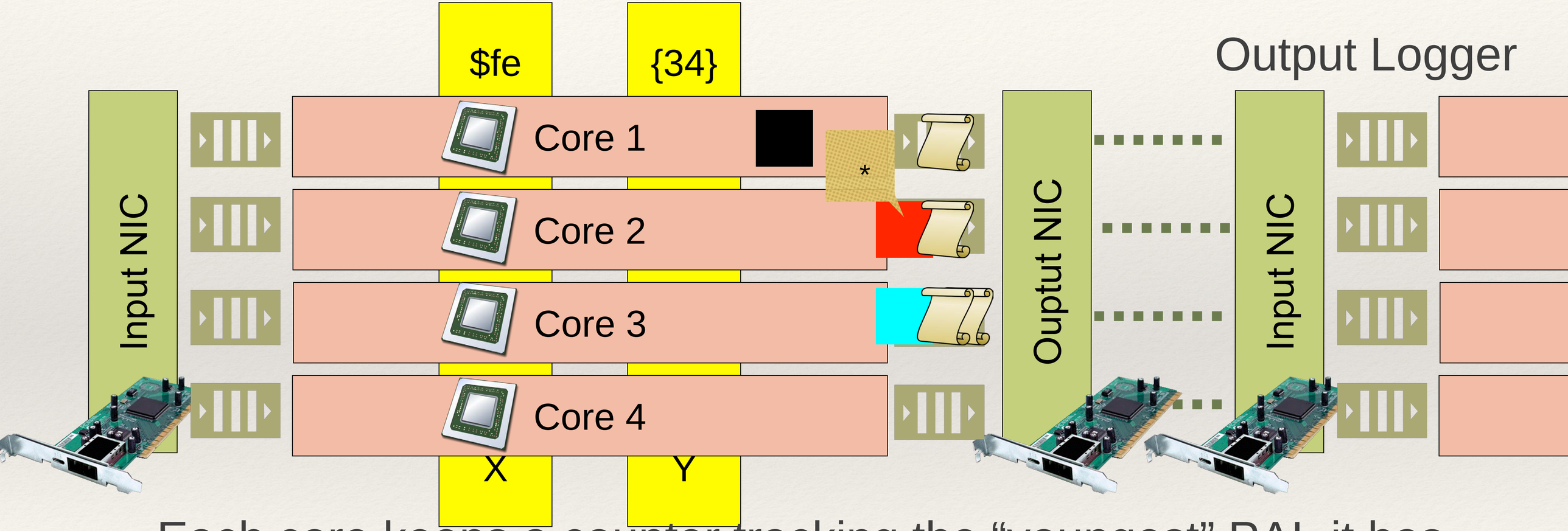
at output queue, all PALs it depends on are already enqueued; or are alrea

Ordered Logging



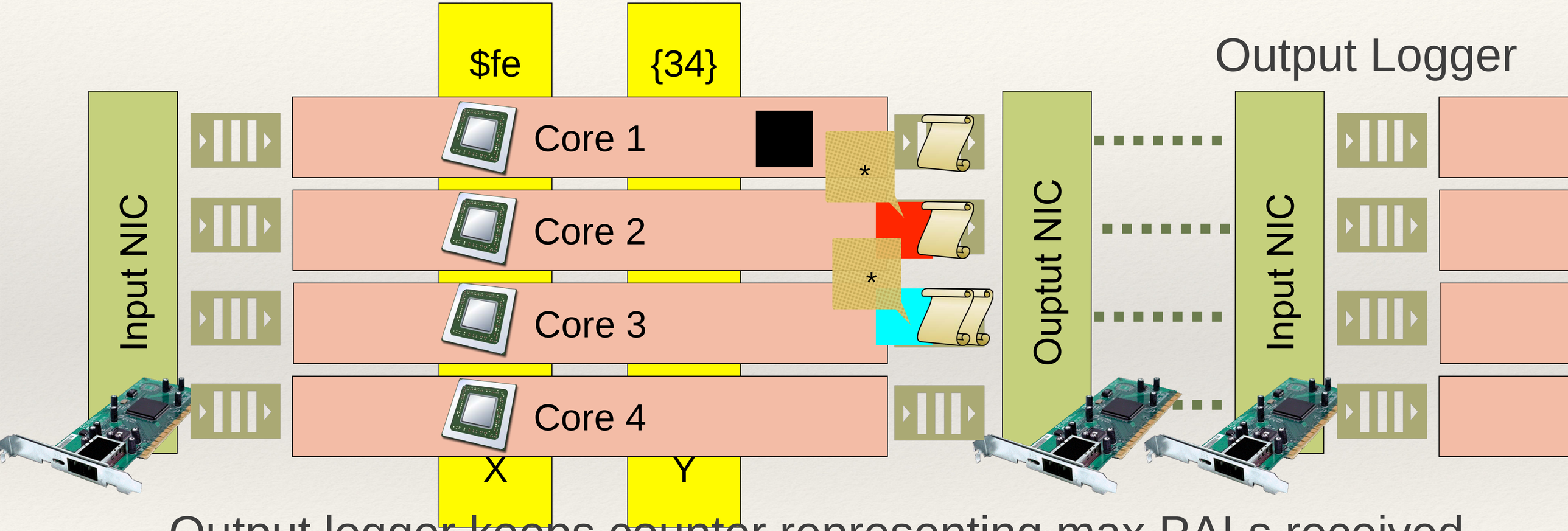
What we want: “flush” all PALs to Output Logger. Then we’re done!

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)

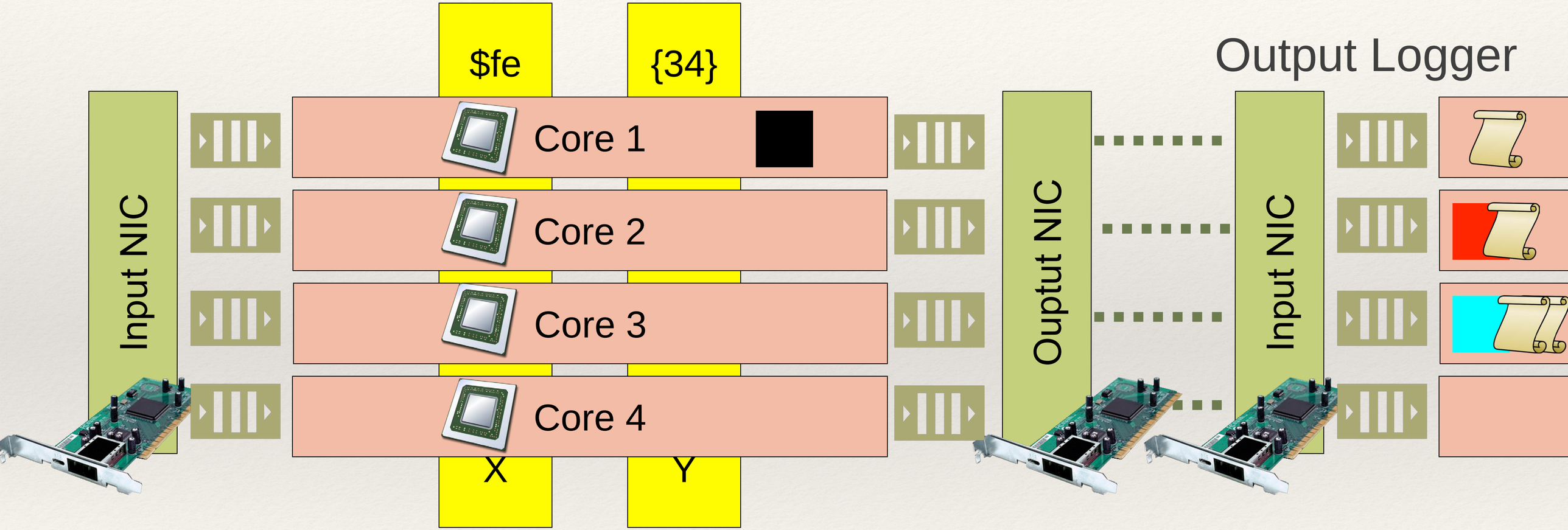
Parallel Release



Output logger keeps counter representing max PALs received.

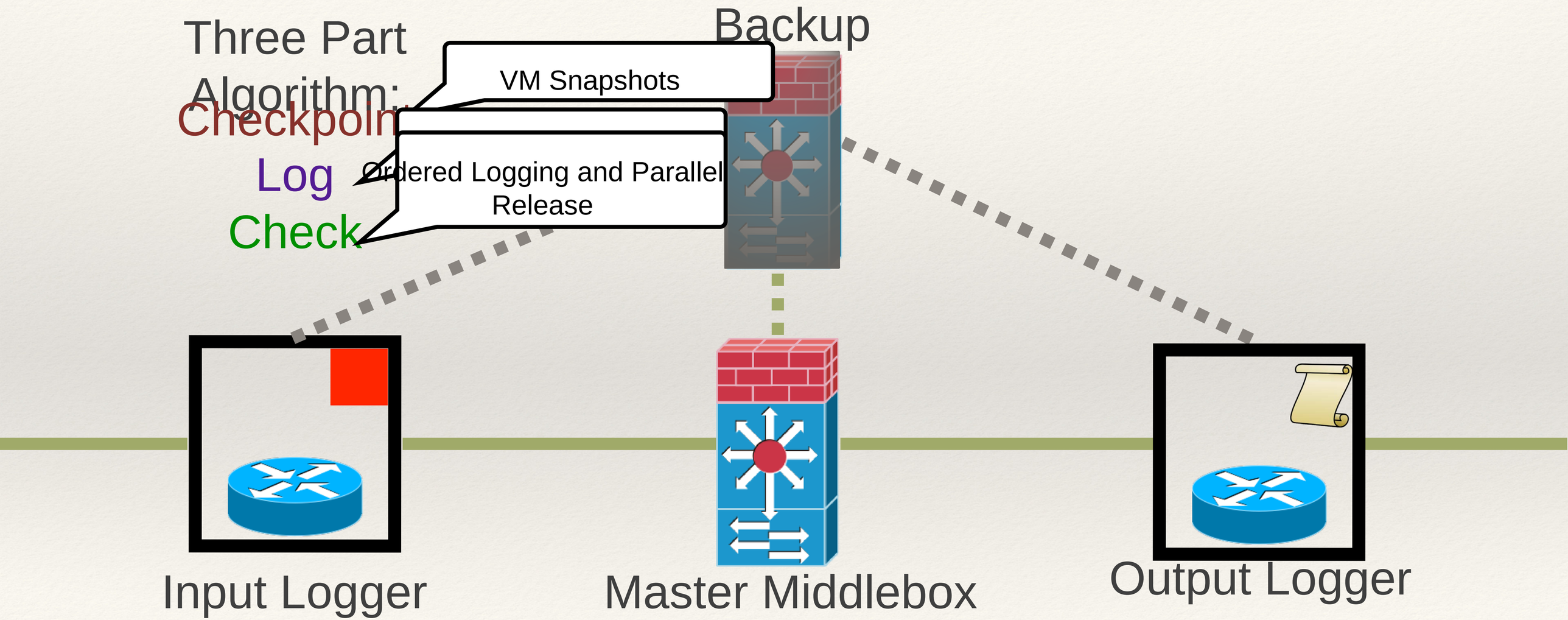
Receive packet: reads all counters to compare against the attached counters.

Parallel Release



If attached counters <= all counters, release packet!

Recap


Three Part Algorithm:
Checkpoint
Log
Check
VM Snapshots
Ordered Logging and Parallel Release
Backup
Input Logger
Master Middlebox
Output Logger

Thanks

Backupland

Latency	Throughput	Recovery Time
Remus [NSDI 2008]: 50,000us overhead	None higher than 200kpps	100s of ms
Pico [SOCC 2013]: 8000us overhead		FTMB: increases recovery time by 50-300ms.
FTMB: 30us overhead	FTMB: 1.4-4Mpps	Still fast enough not to trigger TCP timeouts or errors!

What I’m not talking about today

- a prototype based on Xen, Click, and DPDK.
- a tool to instrument shared state with PALs automatically.
- How replay works in detail.
- Alternative output commit approaches.

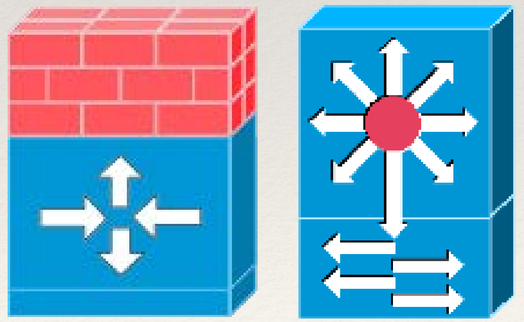
Ordered Logging and Parallel Release

- 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.

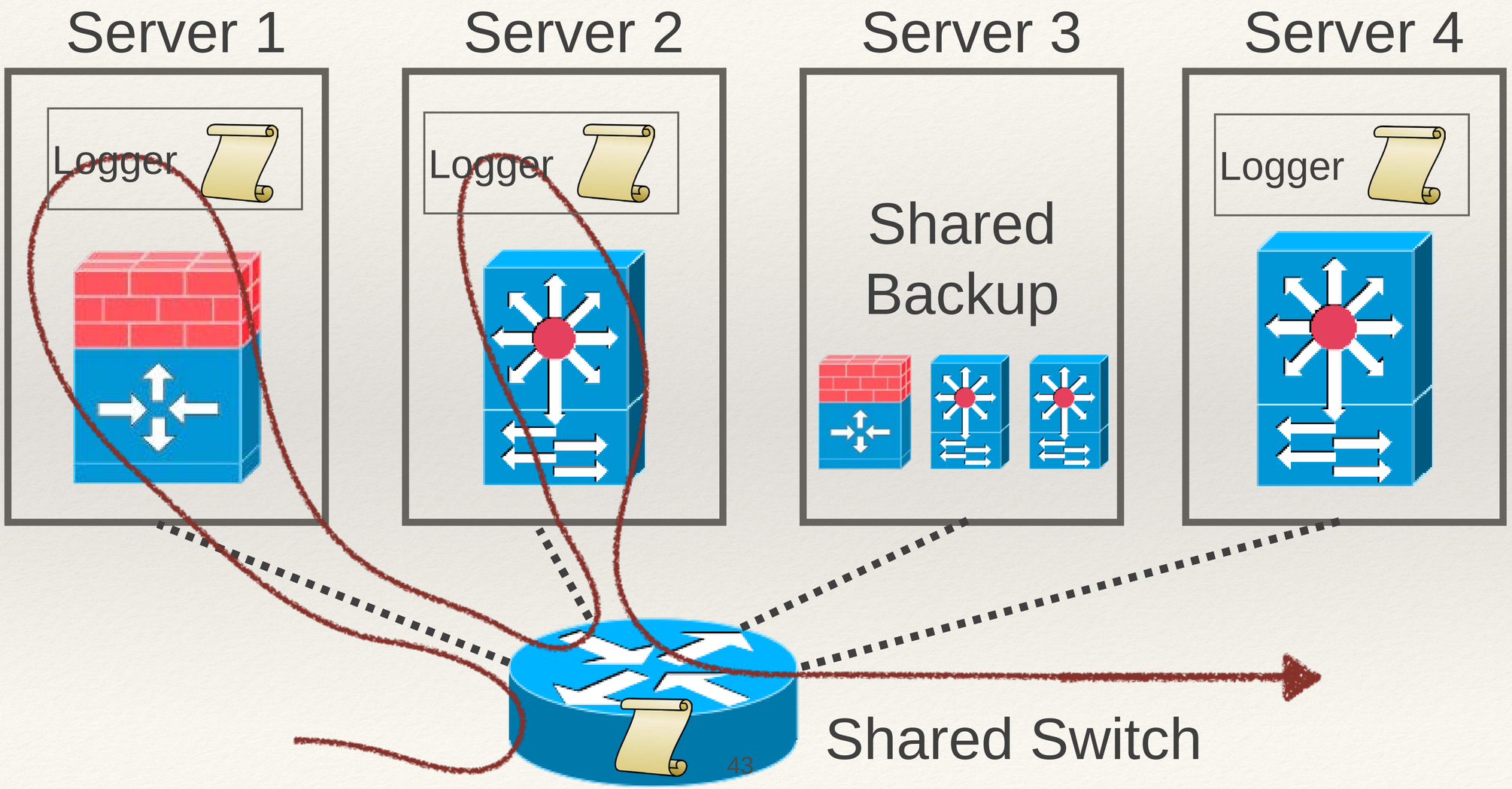
FTMB Outside of Academia

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

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

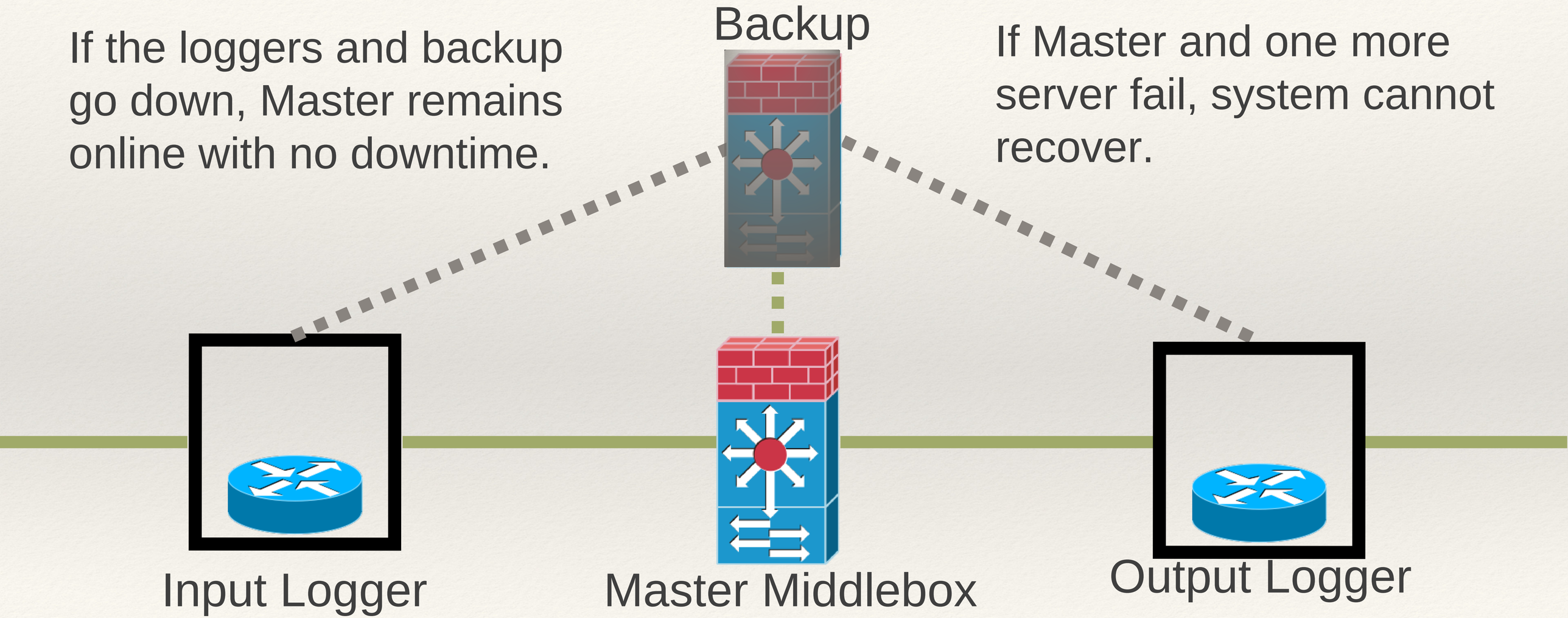


NFV Frameworks

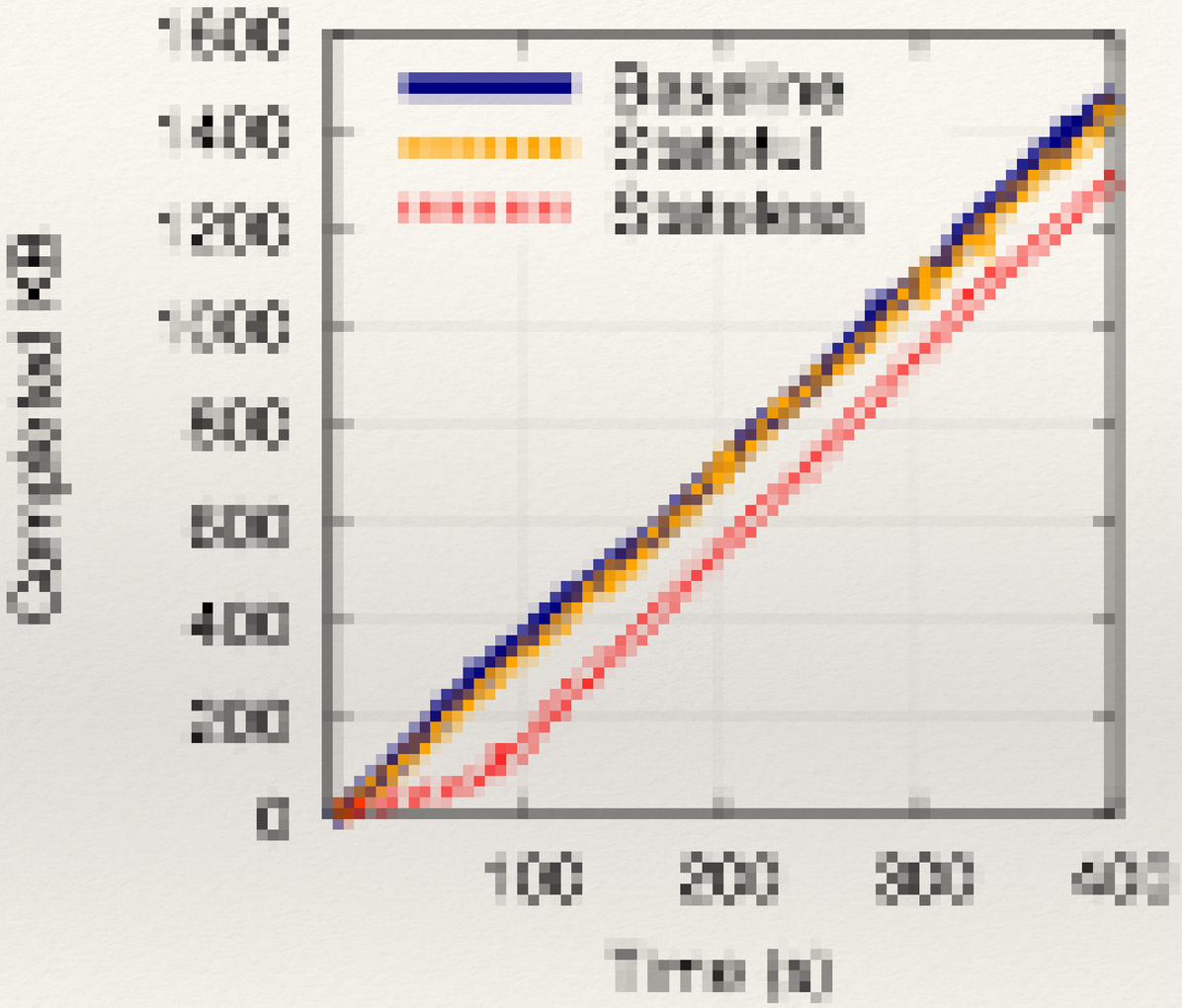


Recovery Guarantees

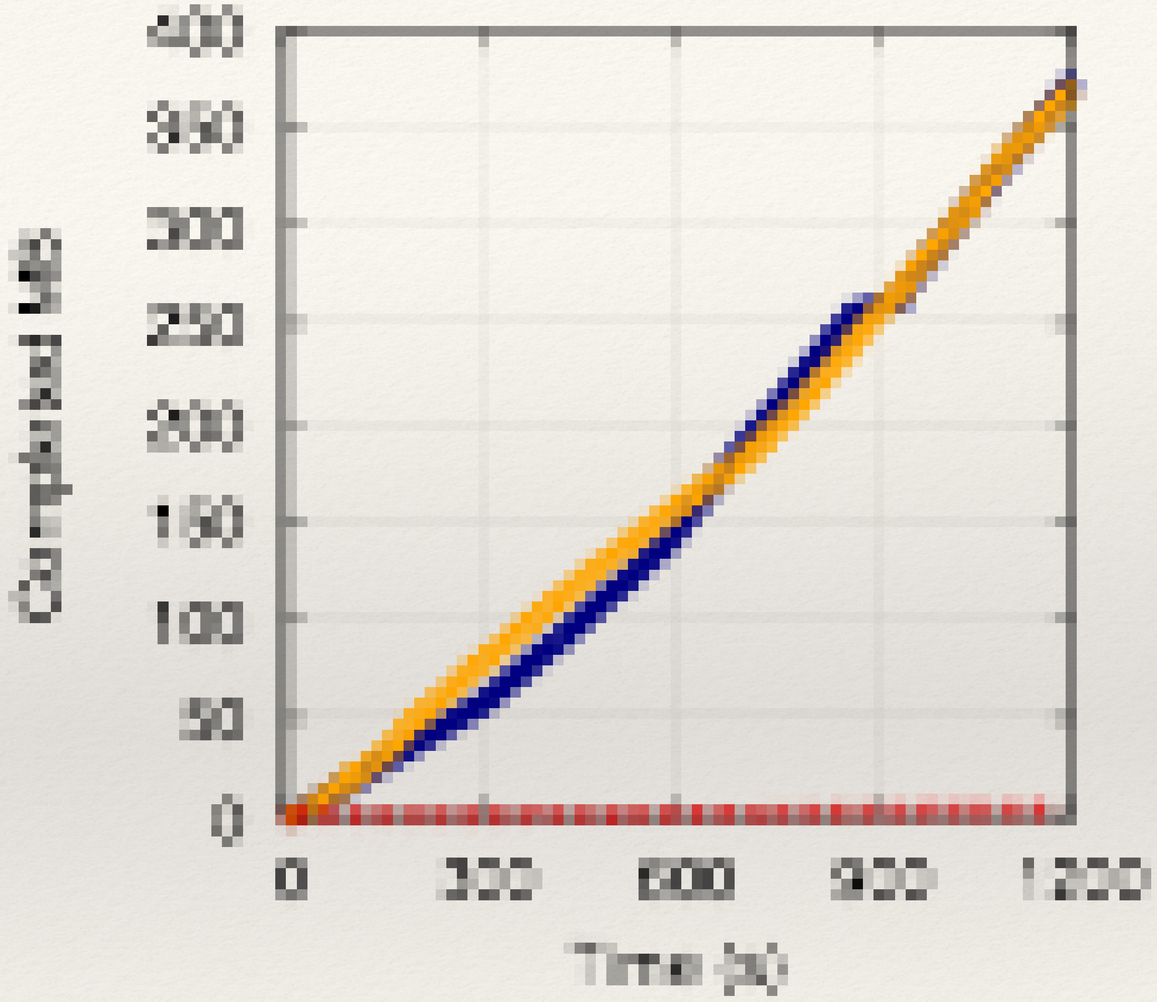
If the loggers and backup go down, Master remains online with no downtime.



If Master and one more server fail, system cannot recover.



FTP Client



BitTorrent Client