RRC: Responsive Replicated Containers

Diyu Zhou*
UCLA and EPFL

Yuval Tamir
UCLA

***Looking for a faculty job**

Server Applications Need Responsive Fault Tolerance

Server Applications:

- Low latency
- High throughput

- High reliability

Server Applications Need Responsive Fault Tolerance

Server Applications:

- Low latency
- High throughput

→ Multithreading

High reliability

→ Fault Tolerance

Server Applications Need Responsive Fault Tolerance

Server Applications:

- Low latency
- High throughput
  → Multithreading

High reliability
  → Fault Tolerance

Fault Tolerance Mechanism Requirements

- Low latency overhead
- Maintain high throughput
  - Low throughput overhead
  - Support multithreading

Server Applications Need Responsive Fault Tolerance

Server Applications:

- Low latency
- High throughput

  → Multithreading

High reliability

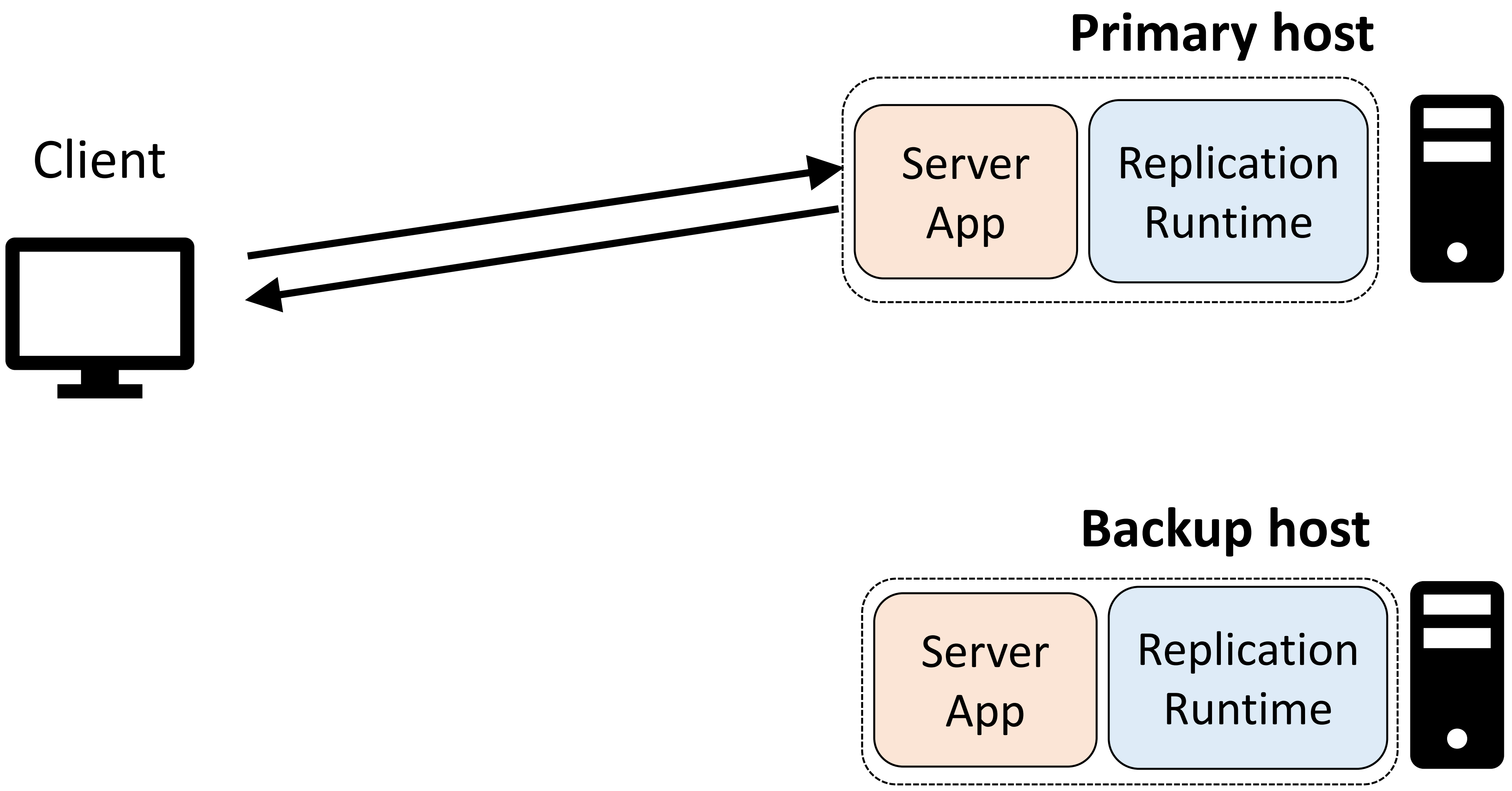
  → Fault Tolerance

Fault Tolerance Mechanism Requirements

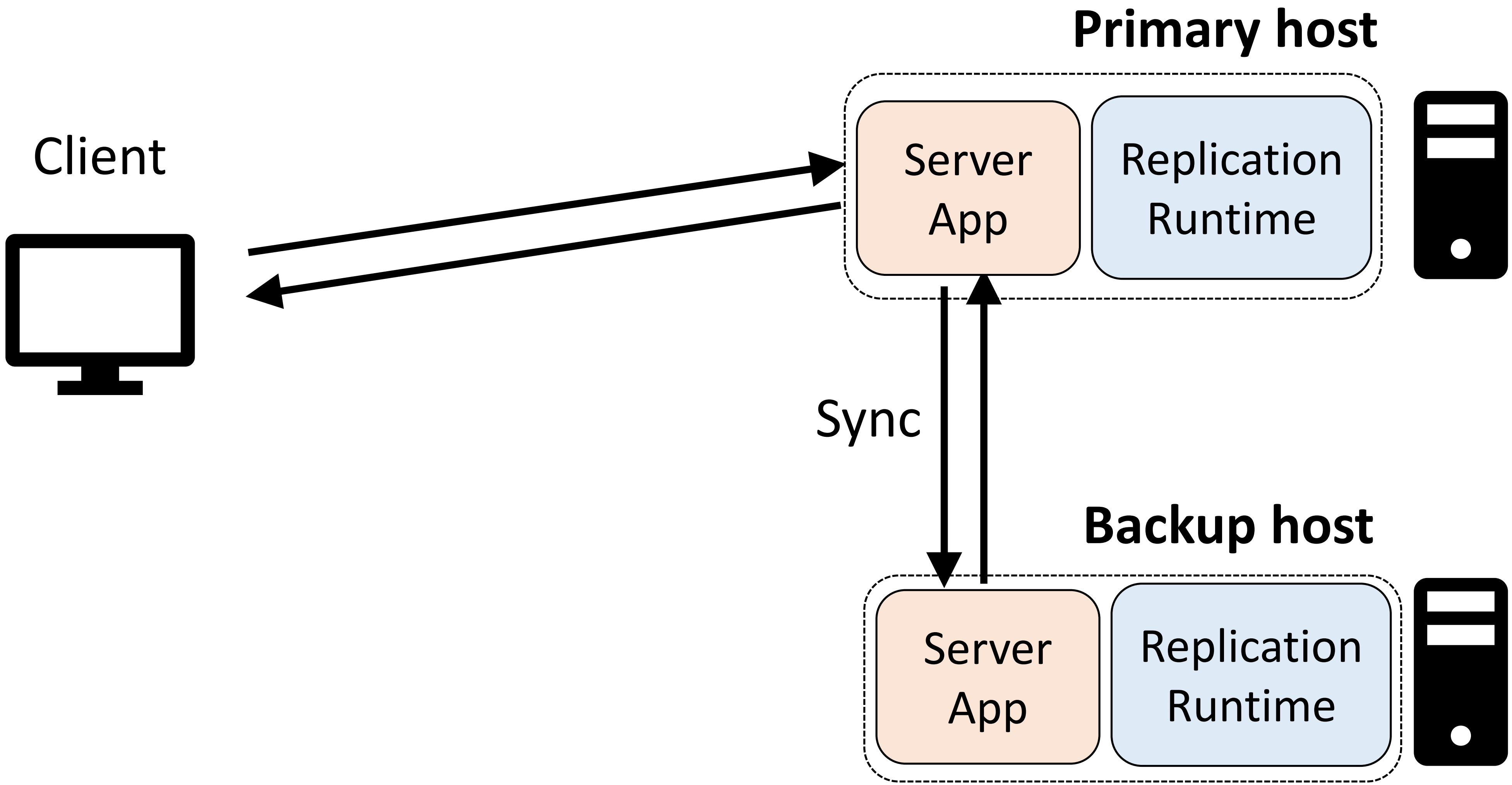
- Low latency overhead
- Maintain high throughput
  - Low throughput overhead
  - Support multithreading
- Minimize development cost
  - No code modification
  - Compatibility with existing clients

  → Application Transparency

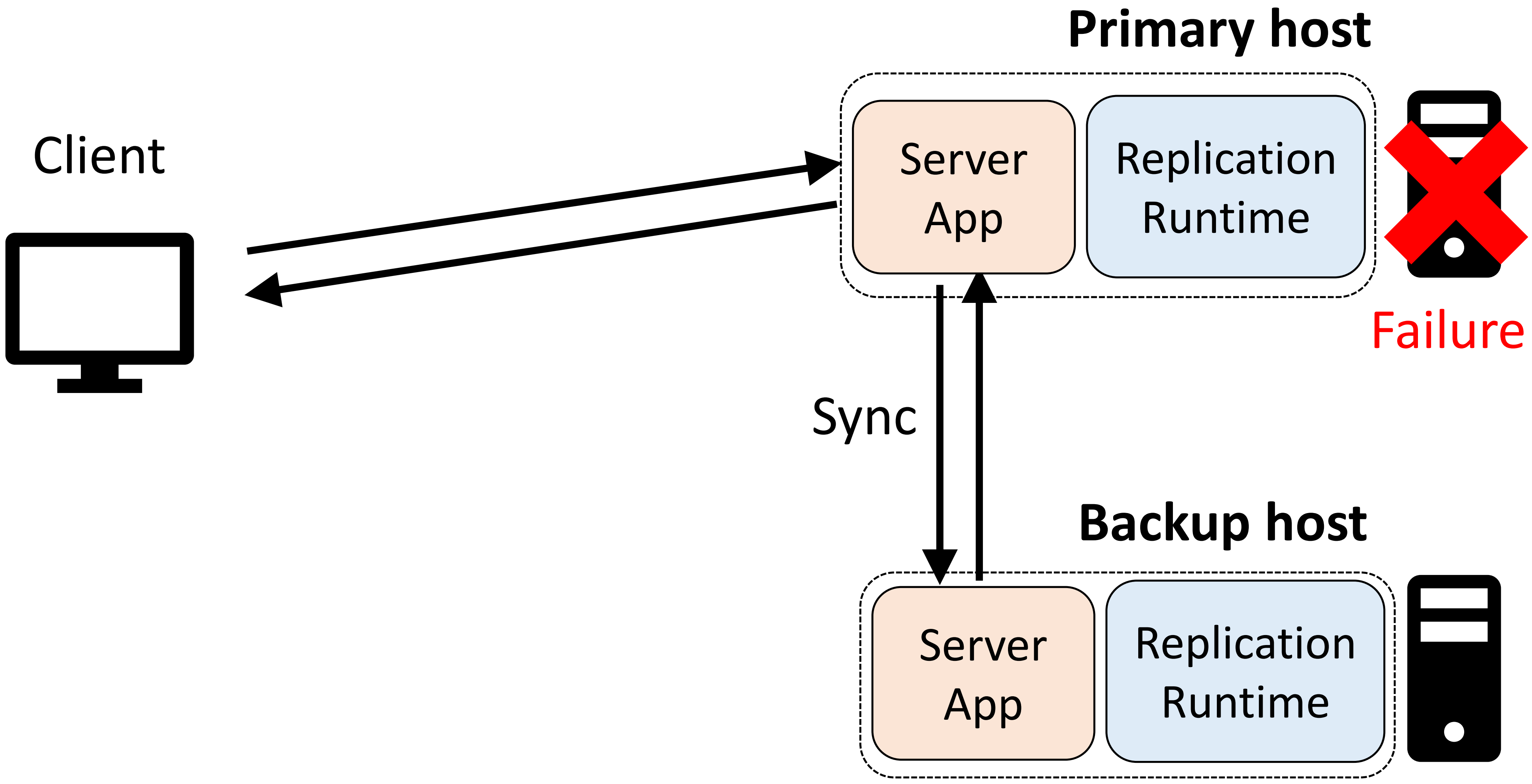
Replication → Application-Transparent Fault Tolerance



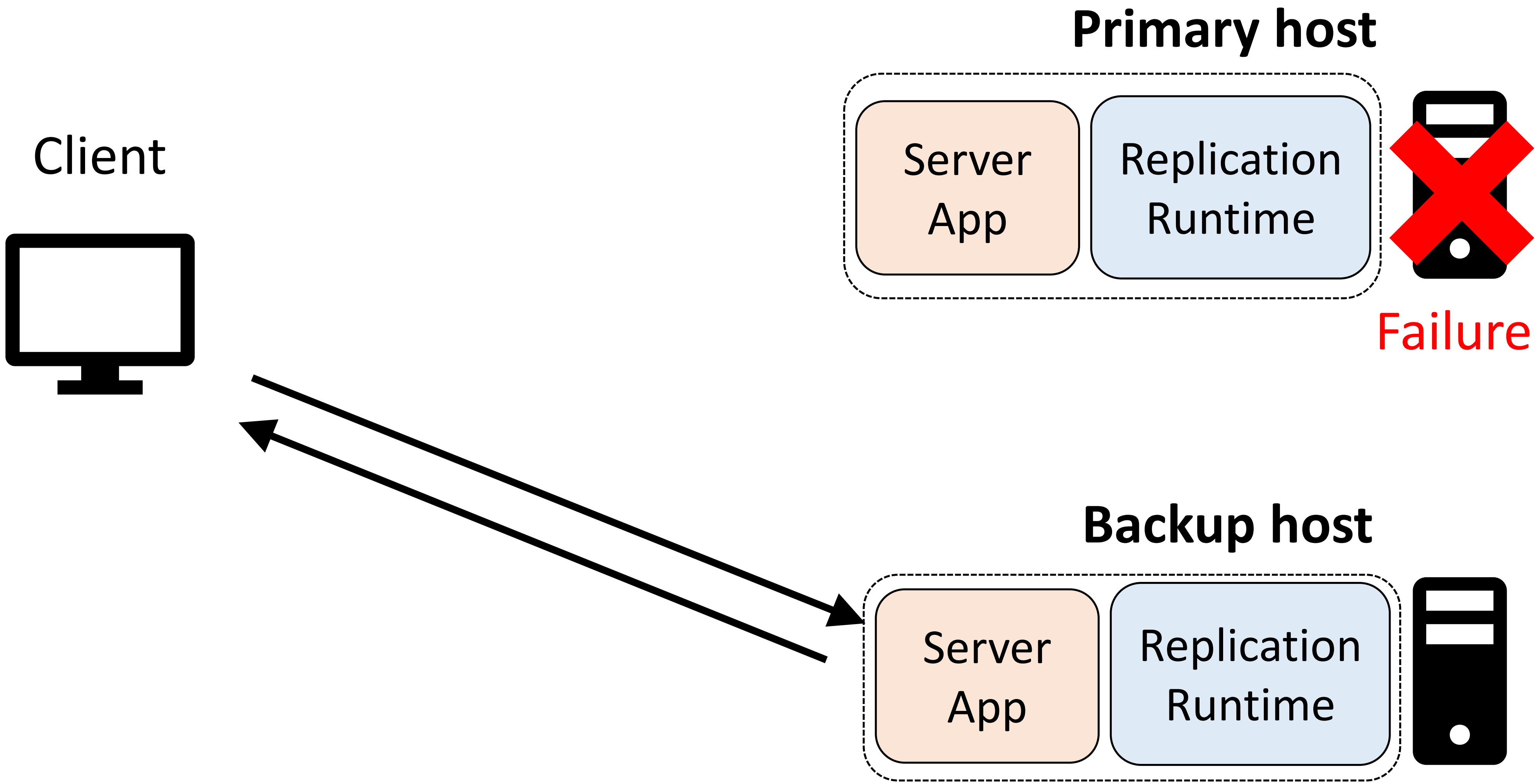
Replication → Application-Transparent Fault Tolerance



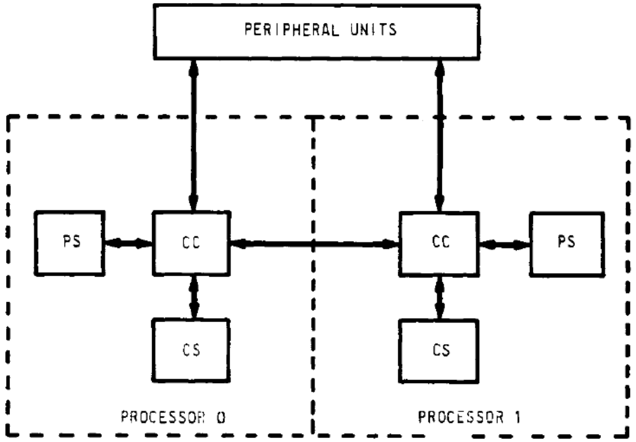
Replication → Application-Transparent Fault Tolerance



Replication → Application-Transparent Fault Tolerance

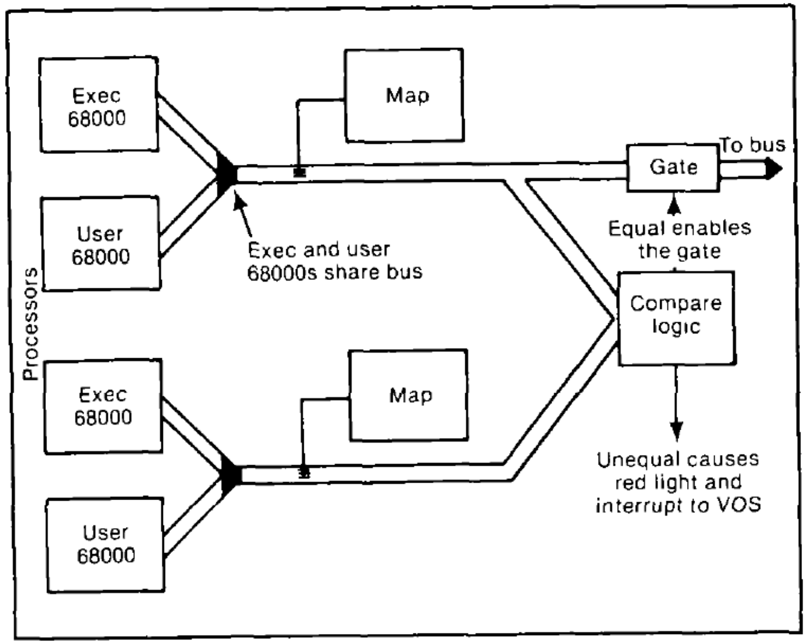


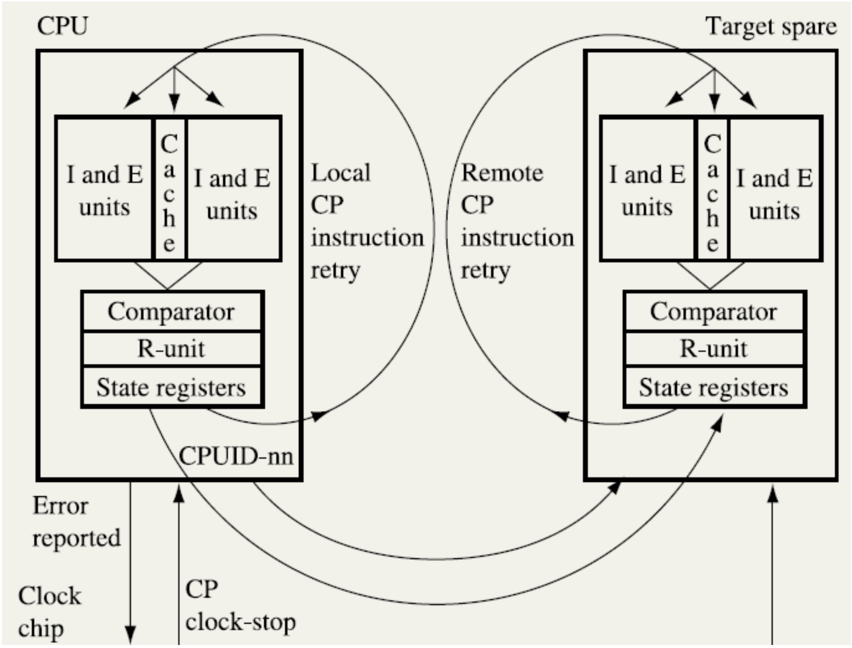
Replication is Old News



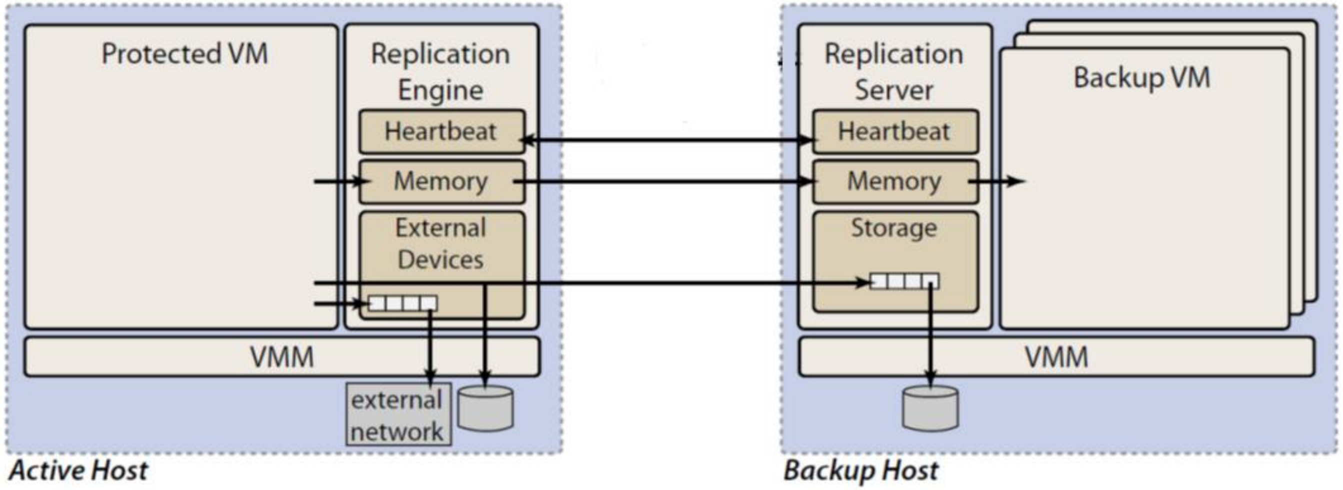
Bell Systems No. 1 ESS (1964)

Stratus/32 multiprocessor node (1983)





IBM G5/G6 Processing Unit (1999)



Remus: Virtual Machine Replication (2008)

What is Missing in Existing Replication Schemes?

- Many older schemes:
  - Require customized hardware
  - No support for multithreaded applications

What is Missing in Existing Replication Schemes?

- Many older schemes:
  - Require customized hardware
  - No support for multithreaded applications
- Schemes based on checkpointing to a **passive** backup
  - Unacceptable high latency overhead

What is Missing in Existing Replication Schemes?

- Many older schemes:
  - Require customized hardware
  - No support for multithreaded applications
- Schemes based on checkpointing to a **passive** backup
  - Unacceptable high latency overhead
- Schemes based on **active** replication
  - Untracked nondeterministic events (e.g., data races)
    - Unpredictable slowdown during normal operation (with some schemes)
    - Recovery failure (with some schemes)
  - Performance limited by tight coupling among replicas.

What is Missing in Existing Replication Schemes?

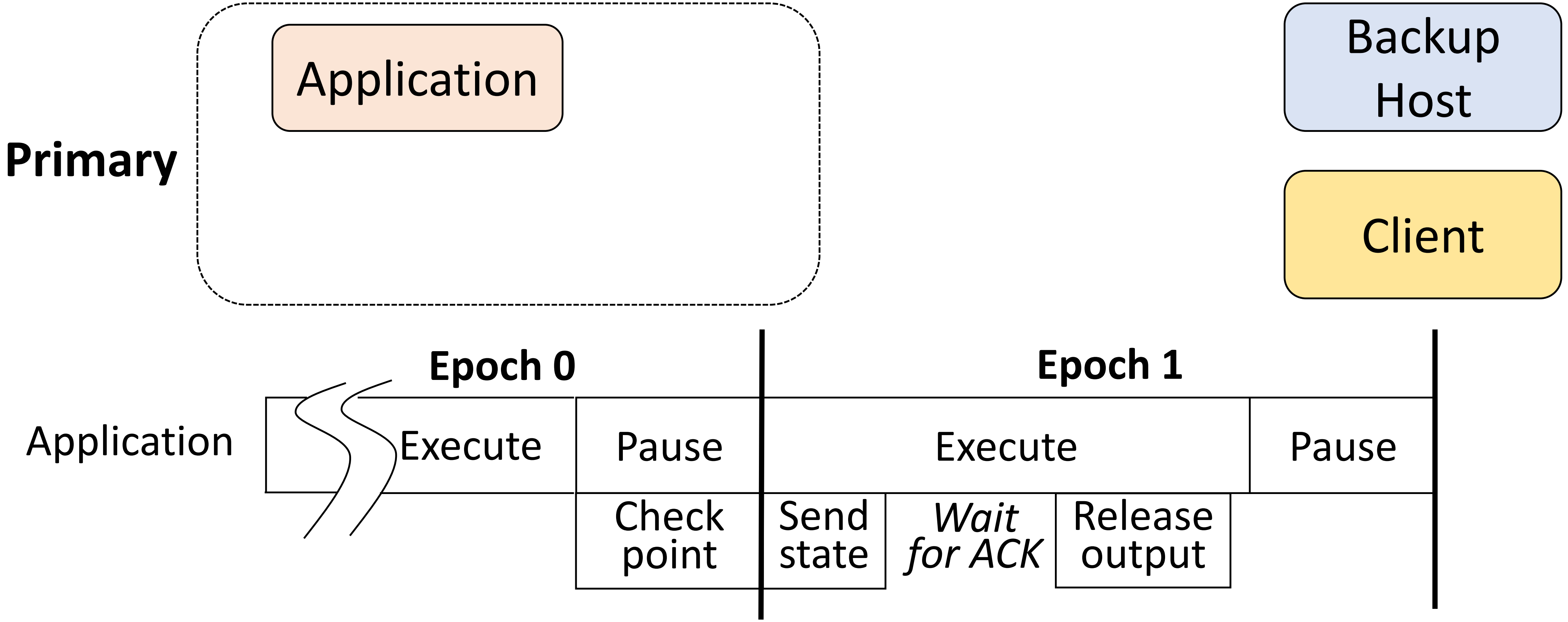
- Many older schemes:
  - Require customized hardware
  - No support for multithreaded applications
- Schemes based on checkpointing to a **passive** backup
  - Unacceptable high latency overhead
- Schemes based on **active** replication
  - Untracked nondeterministic events (e.g., data races)
    - Unpredictable slowdown during normal operation (with some schemes)
    - Recovery failure (with some schemes)
  - Performance limited by tight coupling among replicas.

RRC overcomes limitations by **decoupling** replication-related operations from normal operations

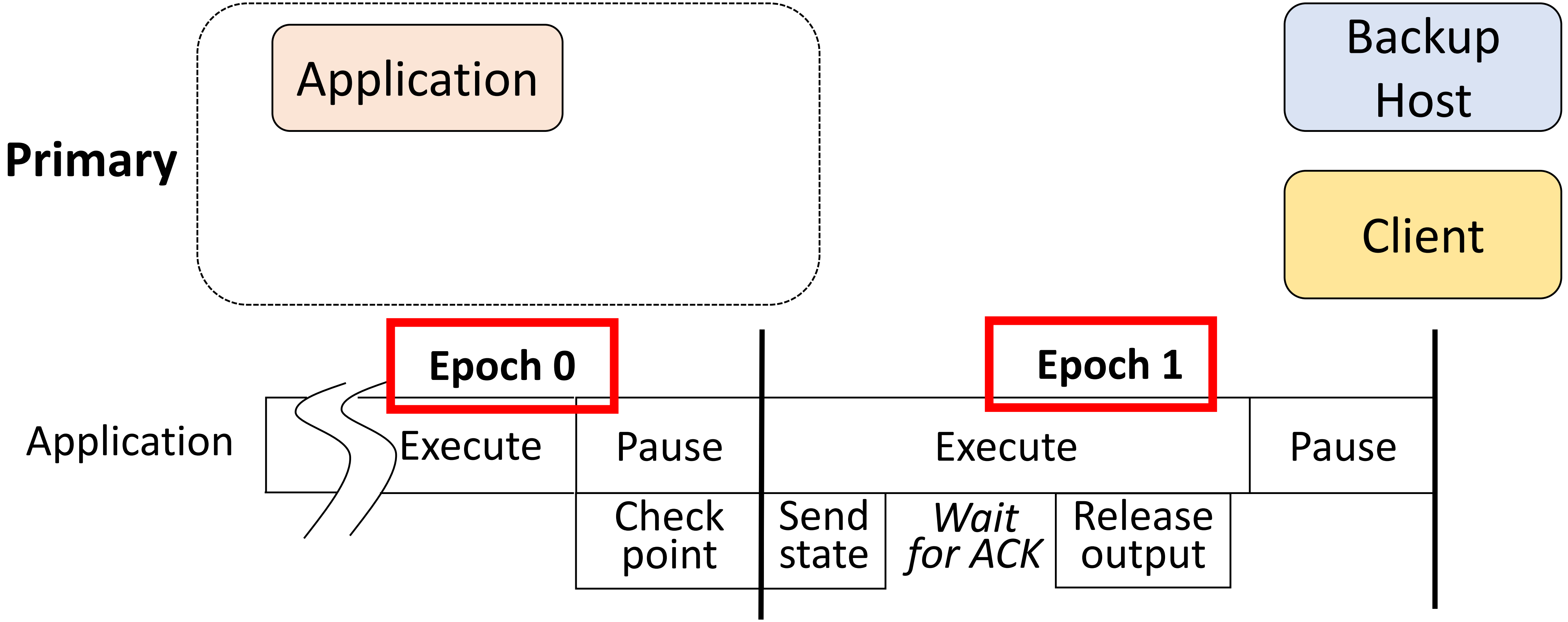
Talk Outline

- Preface
- **Motivation**
- RRC overview
- Overcoming design and implementation challenges
- Evaluation

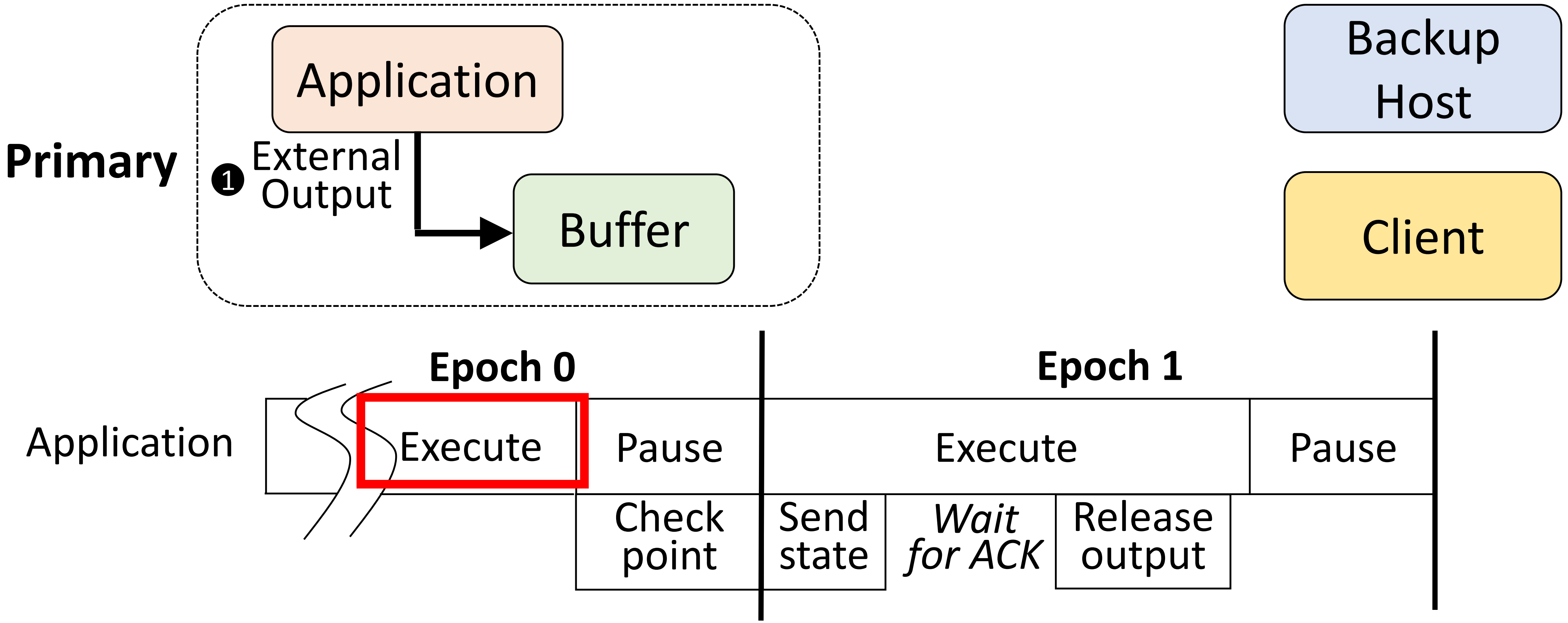
Passive Backup: Checkpointing-Based Mechanisms



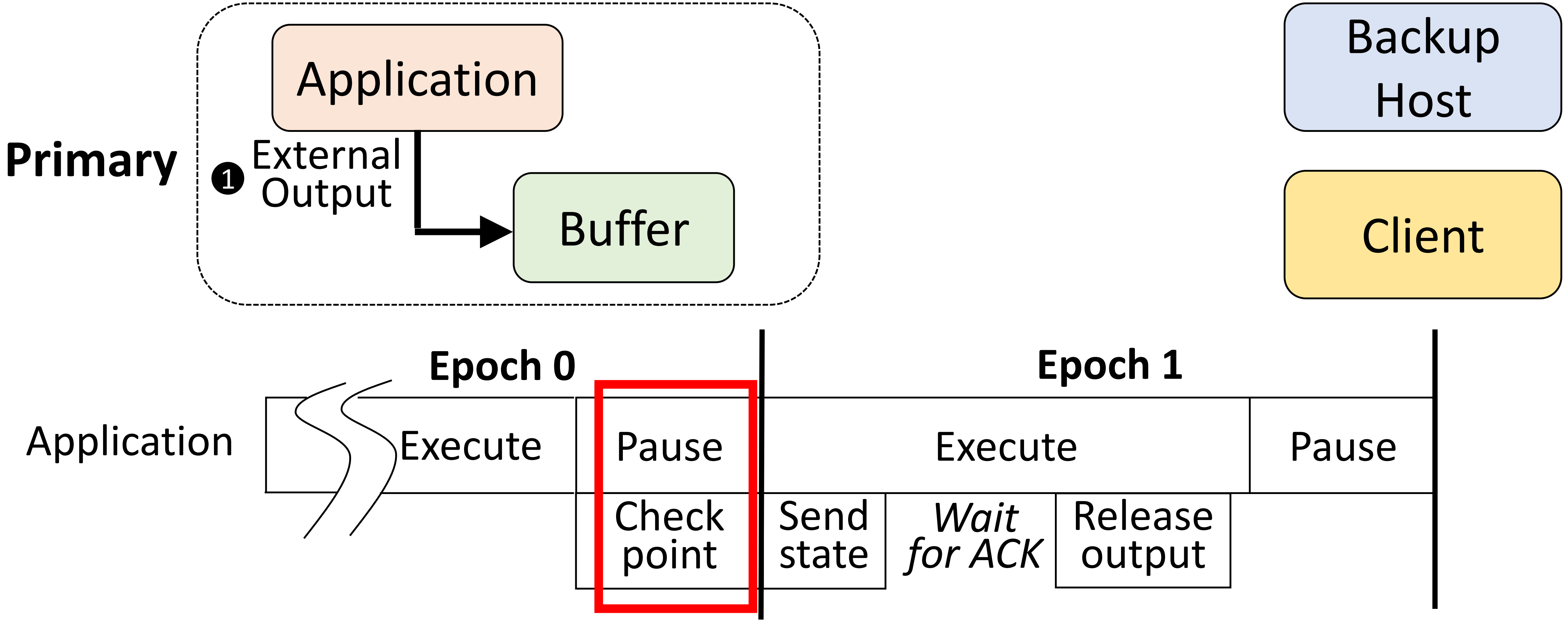
Passive Backup: Checkpointing-Based Mechanisms



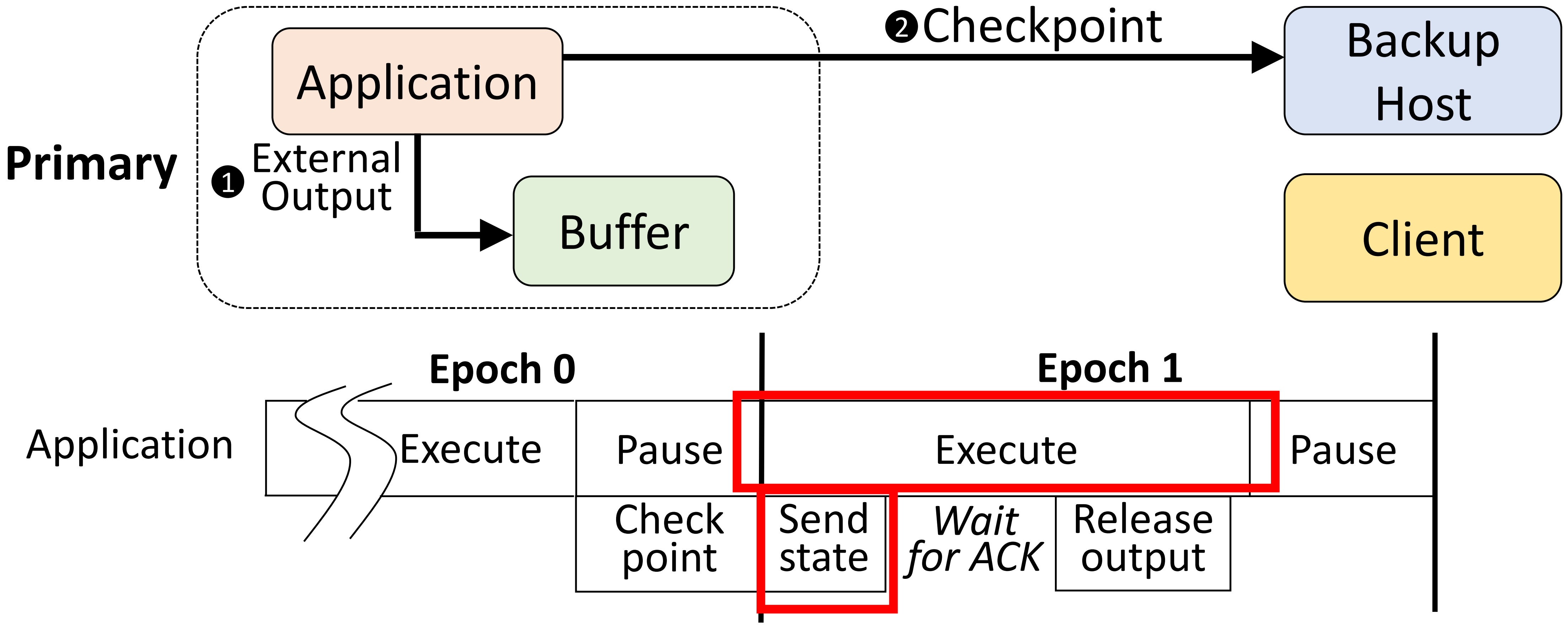
Passive Backup: Checkpointing-Based Mechanisms



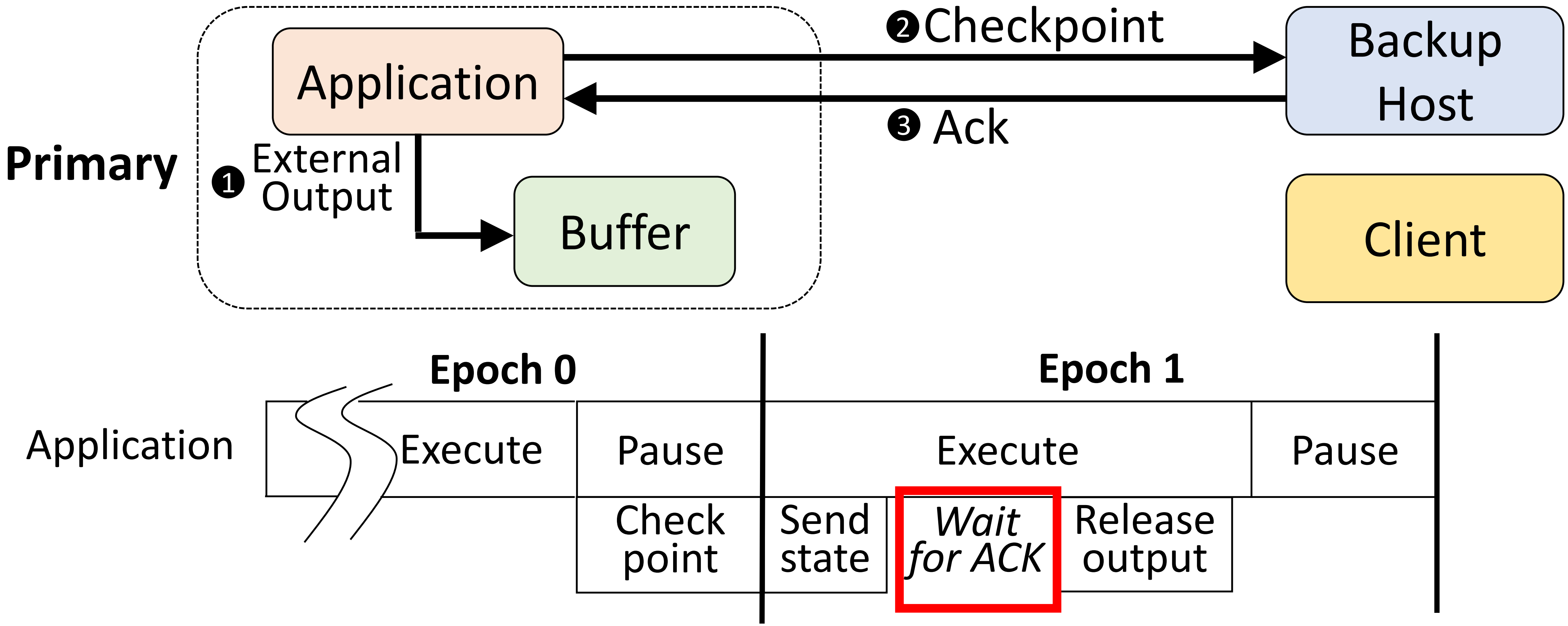
Passive Backup: Checkpointing-Based Mechanisms



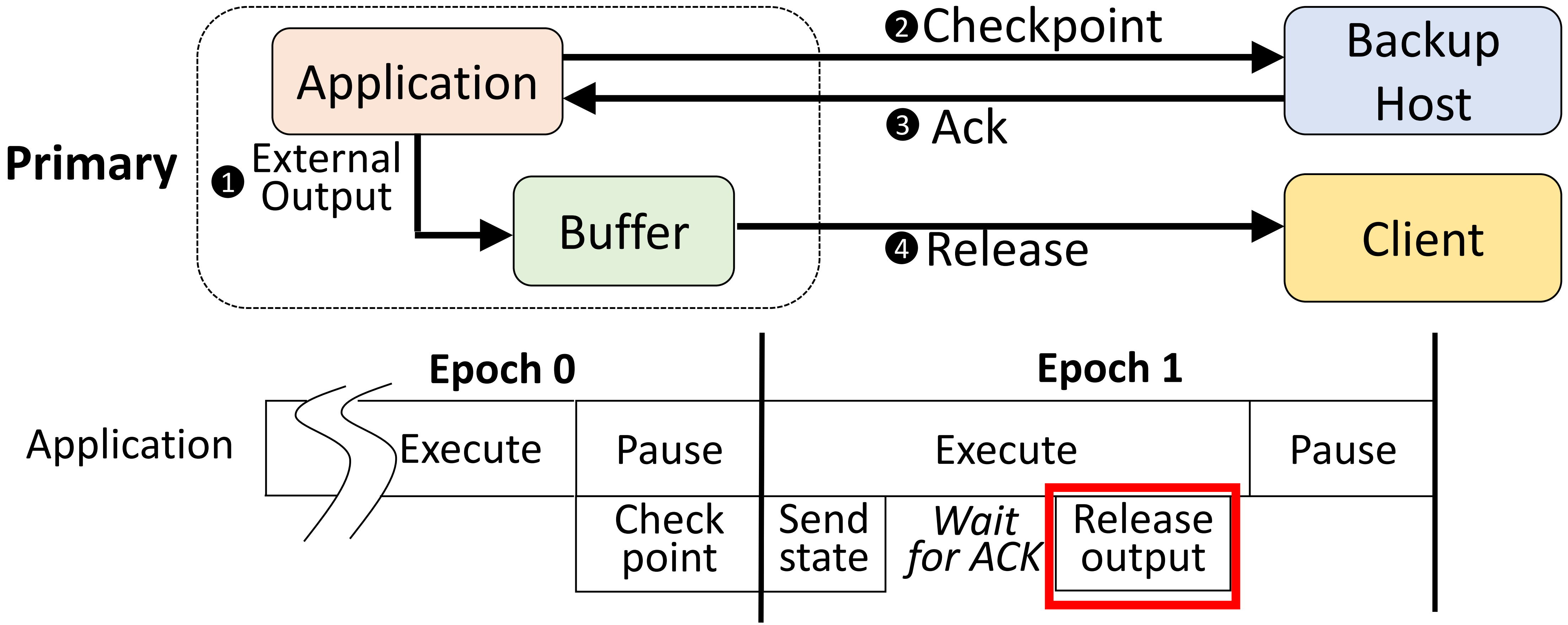
Passive Backup: Checkpointing-Based Mechanisms



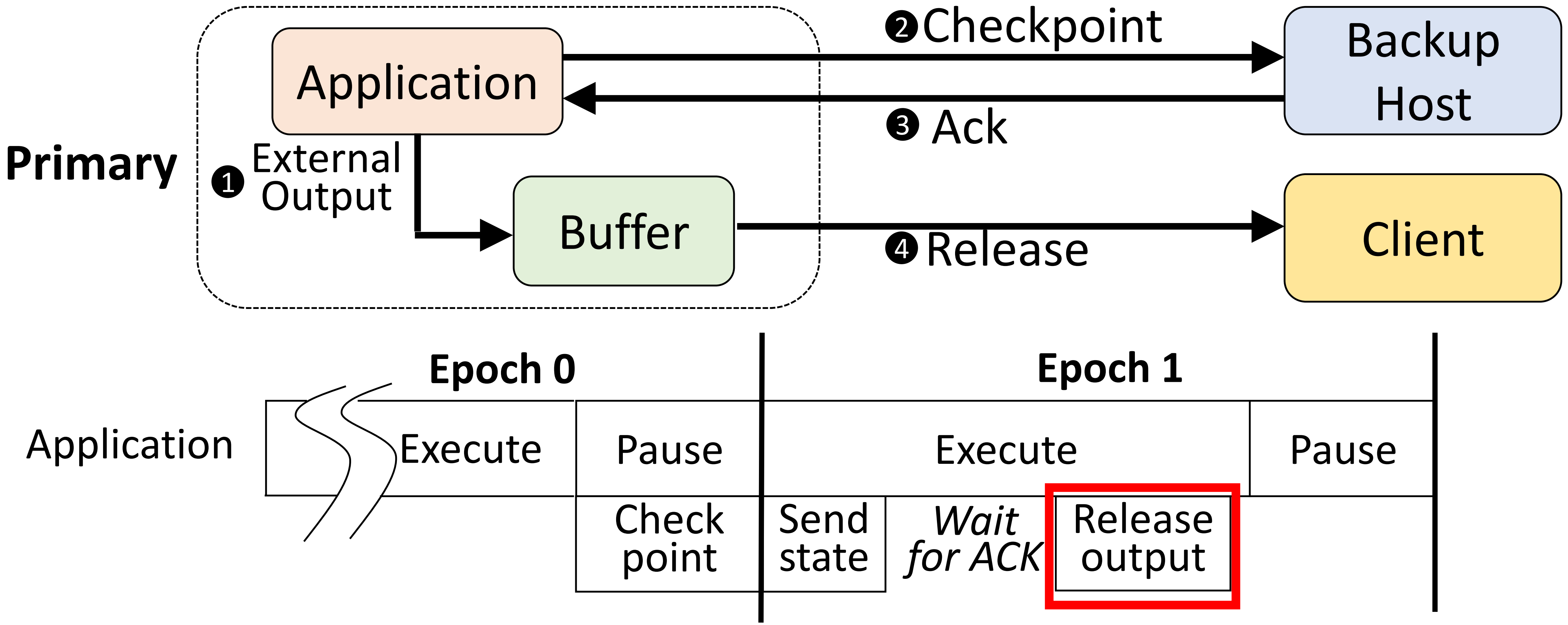
Passive Backup: Checkpointing-Based Mechanisms



Passive Backup: Checkpointing-Based Mechanisms



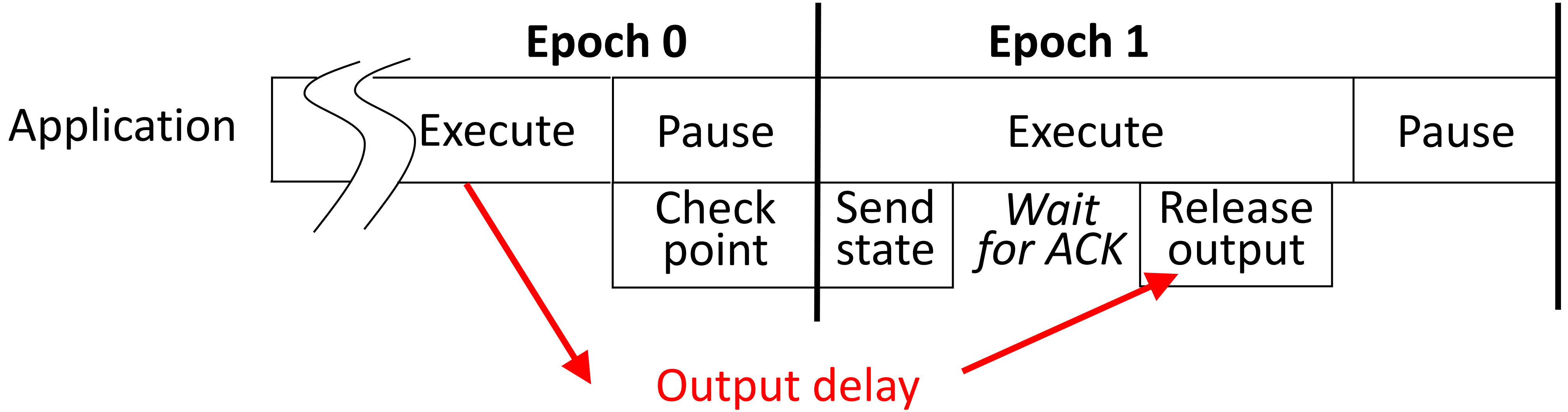
Passive Backup: Checkpointing-Based Mechanisms



Why delayed output:

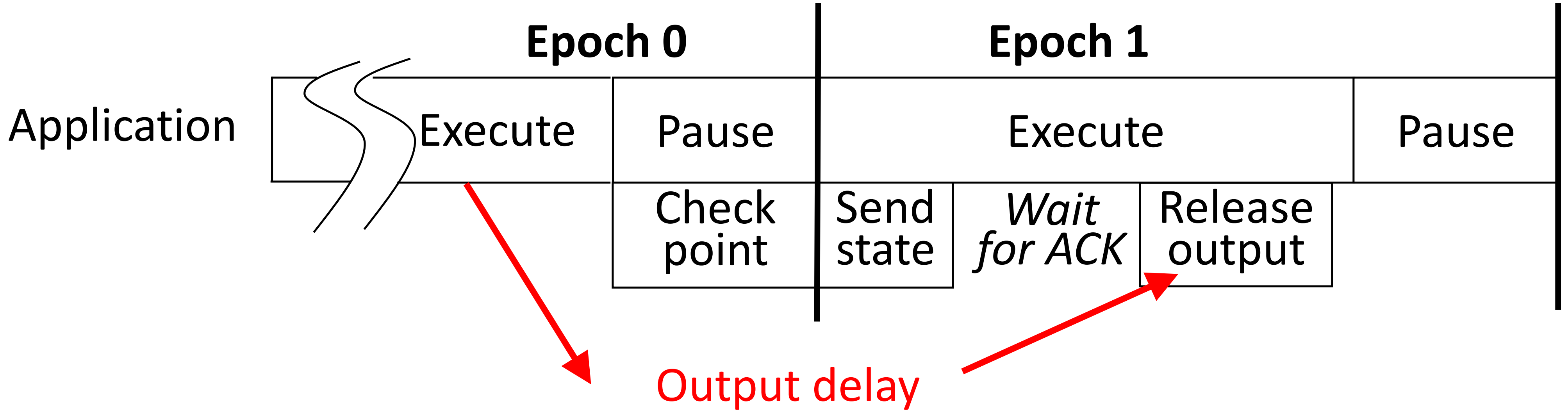
Backup needs to restore state consistent with clients

Checkpointing-Based Mechanisms → High latency Overhead



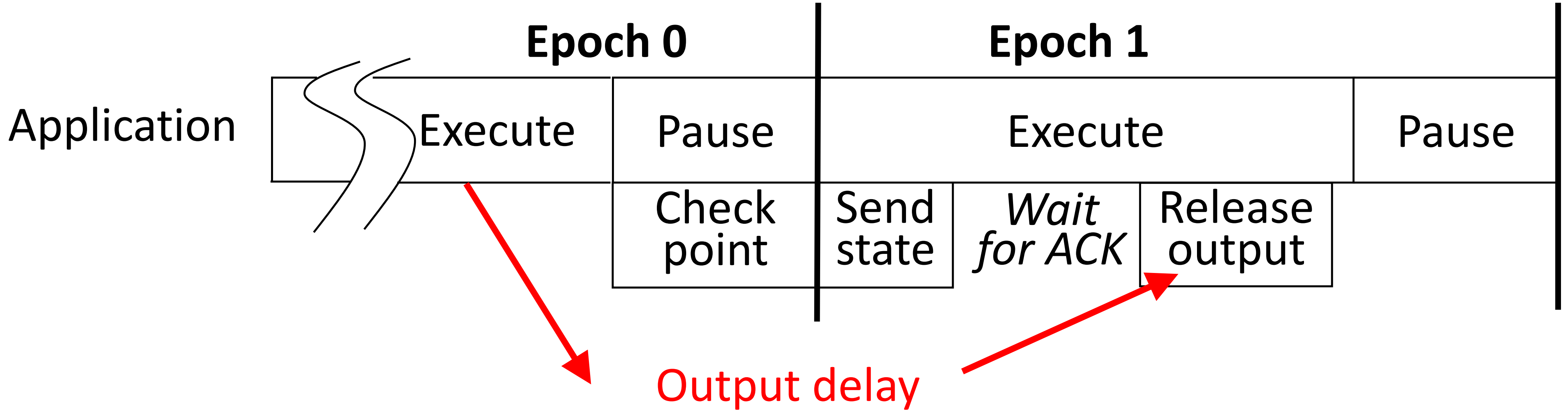
- Output delay = remaining execute time in Epoch 0 + time up to receipt of ACK in Epoch 1

Checkpointing-Based Mechanisms → High latency Overhead



- Output delay = remaining execute time in Epoch 0 + time up to receipt of ACK in Epoch 1
- Checkpointing is expensive → Critical checkpointing (epoch) interval tradeoff
  - *Short* interval → *High* throughput overhead, *low* latency overhead
  - *Long* interval → *Low* throughput overhead, *high* latency overhead

Checkpointing-Based Mechanisms → High latency Overhead



- Output delay = remaining execute time in Epoch 0 + time up to receipt of ACK in Epoch 1
- Checkpointing is expensive → Critical checkpointing (epoch) interval tradeoff
  - *Short* interval → *High* throughput overhead, *low* latency overhead
  - *Long* interval → *Low* throughput overhead, *high* latency overhead

In practice: 10s of milliseconds interval → 10s of milliseconds latency

→ Unacceptably high latency overhead

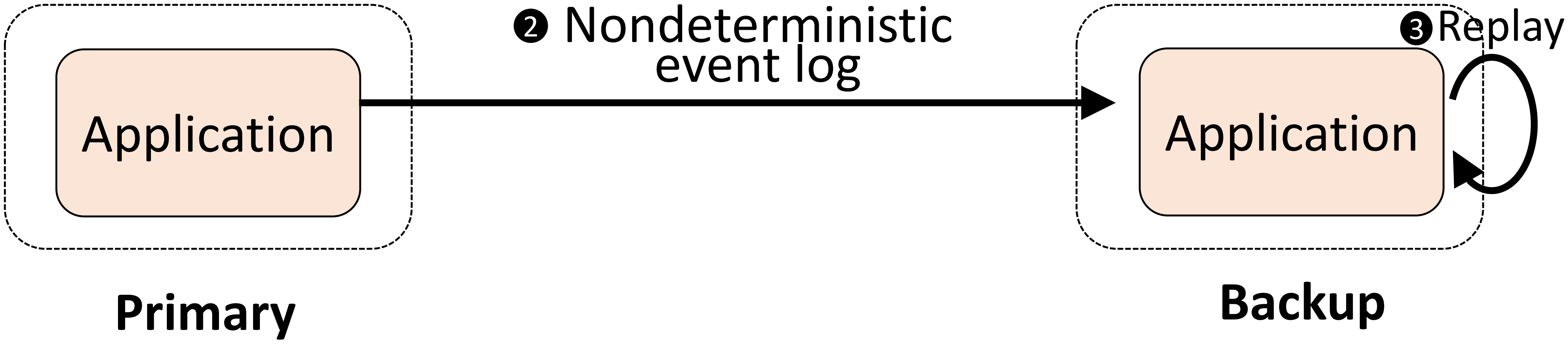
Active Backup: Mechanisms based on Active Replication

- Primary and backup execute application code

Active Backup: Mechanisms based on Active Replication

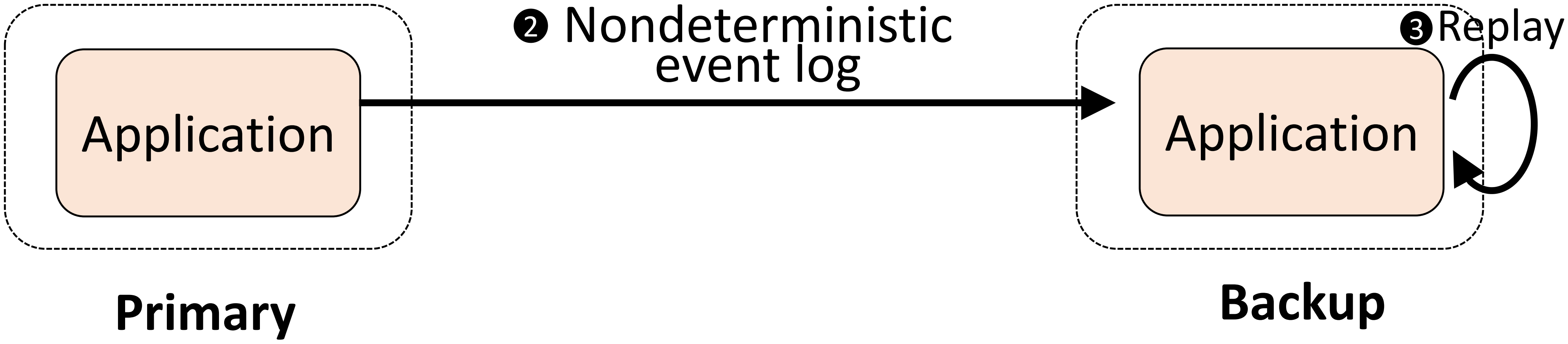
- Primary and backup execute application code
- Primary sends outcomes of nondeterministic events to backup

Active Backup: Mechanisms based on Active Replication



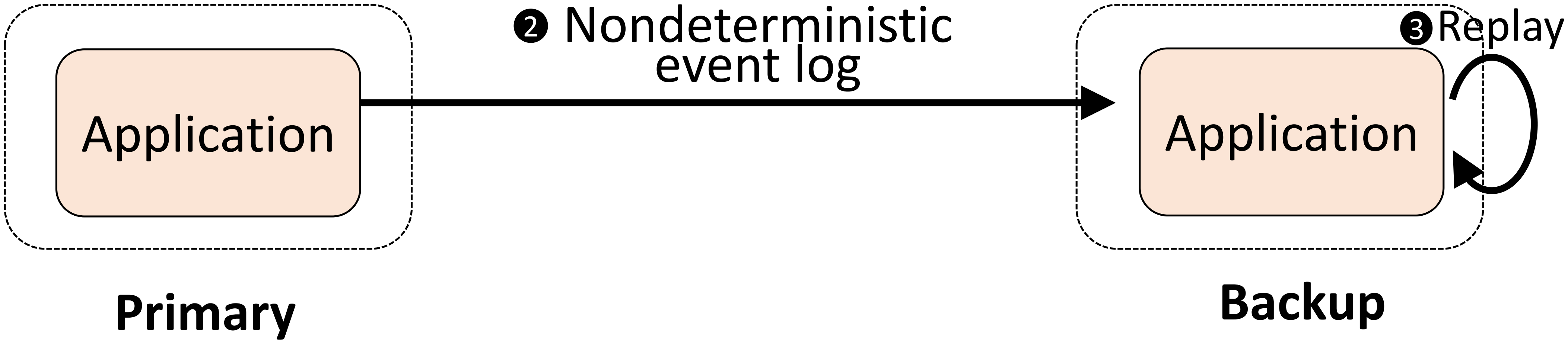
- Primary and backup execute application code
- Primary sends outcomes of nondeterministic events to backup
- Backup enforces outcome of nondeterministic events to match execution

Disadvantages of Active Backup Mechanisms



Backup execution must be consistent with primary:

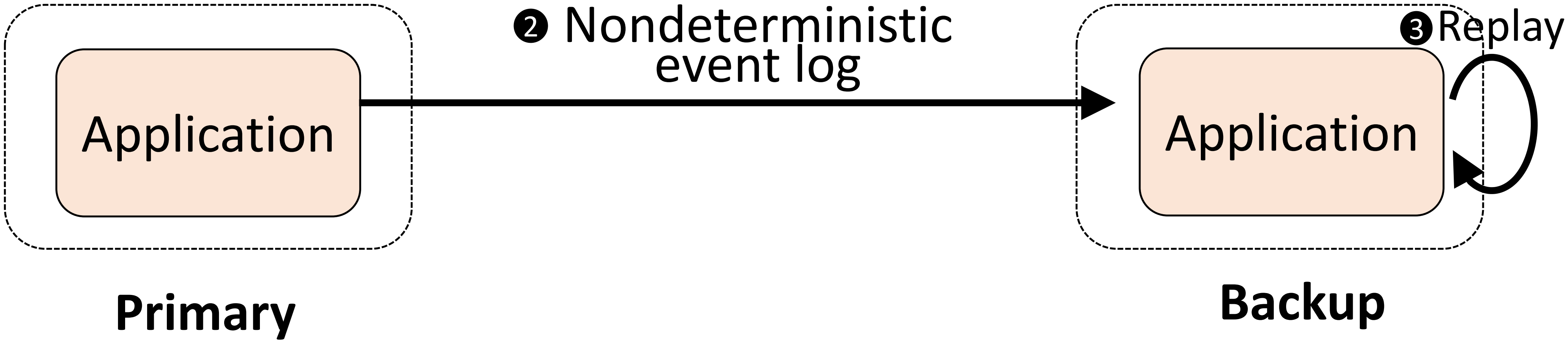
Disadvantages of Active Backup Mechanisms



Backup execution must be consistent with primary:

→ Consequences of untracked nondeterministic events (e.g., data races):
- Unpredictable slowdowns during normal operation (for some mechanisms)
- Recovery failure (for some mechanisms)

Disadvantages of Active Backup Mechanisms



Backup execution must be consistent with primary:

→ Consequences of untracked nondeterministic events (e.g., data races):

- Unpredictable slowdowns during normal operation (for some mechanisms)
- Recovery failure (for some mechanisms)

- Performance limited by tight coupling between replicas
- Resource overhead **lower bound** = 100%

Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

- Passive backup mechanisms:
  - Checkpoint interval $\leftrightarrow$ delay in releasing outputs
  - Time to take a checkpoint $\leftrightarrow$ service interruption

Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

- Passive backup mechanisms:
  - Checkpoint interval $\leftrightarrow$ delay in releasing outputs
  - Time to take a checkpoint $\leftrightarrow$ service interruption

- Active backup mechanisms:
  - Untracked nondeterminism $\leftrightarrow$ service interruption
  - Performance on the primary $\leftrightarrow$ performance on the backup

Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

- Passive backup mechanisms:
  - Checkpoint interval ↔ delay in releasing outputs
  - Time to take a checkpoint ↔ service interruption

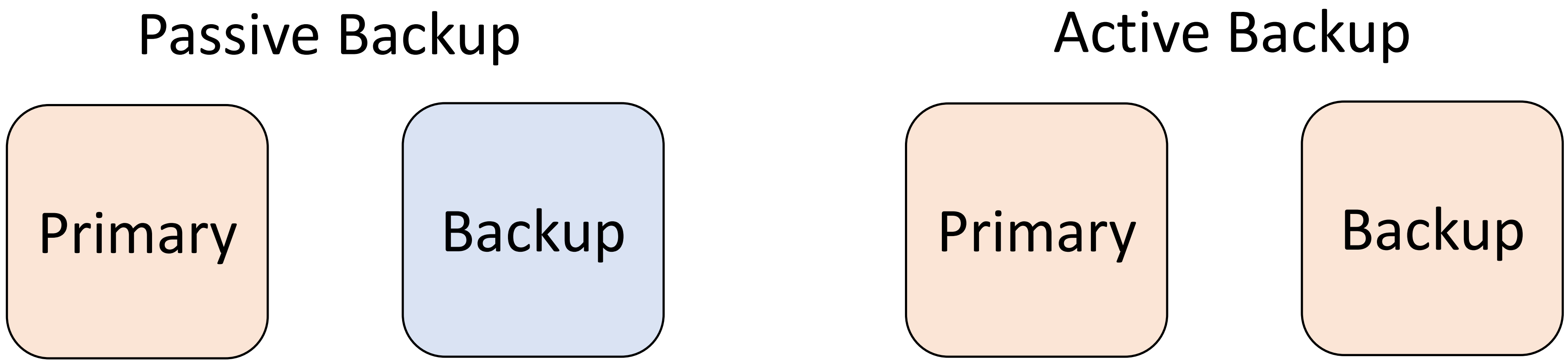
- Active backup mechanisms:
  - Untracked nondeterminism ↔ service interruption
  - Performance on the primary ↔ performance on the backup

RRC breaks these couplings

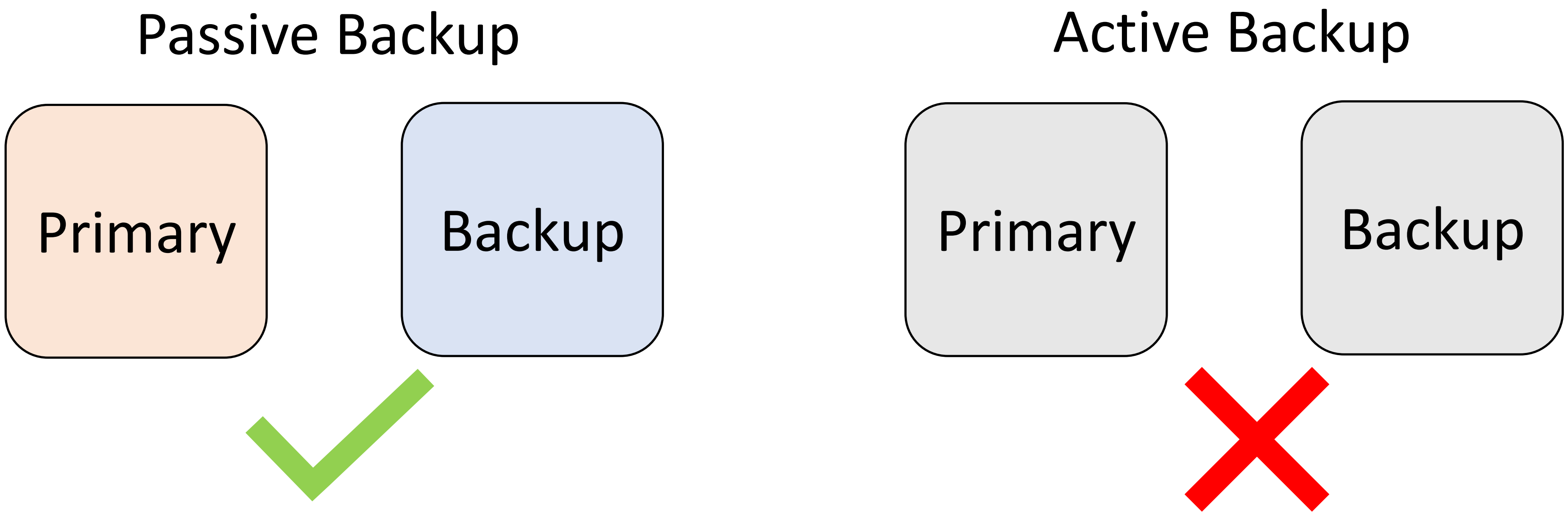
Talk Outline

- Preface
- Motivation
- **RRC overview**
- Overcoming design and implementation challenges
- Evaluation

Passive Backup as the Starting Point



Passive Backup as the Starting Point



- Avoid vulnerability to nondeterminism
- Avoid coupling performance of primary with backup
- Reduce resource overhead

Decoupling Latency Overhead from Checkpoint Interval Using hybrid replication

Passive backup mechanisms: High latency overhead (10s of milliseconds)

Root cause: Coupling of latency overhead and checkpointing interval

Decoupling Latency Overhead from Checkpoint Interval Using hybrid replication

Passive backup mechanisms: High latency overhead (10s of milliseconds)

Root cause: Coupling of latency overhead and checkpointing interval

Solution: Hybrid replication – combine checkpointing with execution replay

- Outputs release decoupled from checkpoint commitment

Decoupling Latency Overhead from Checkpoint Interval Using hybrid replication

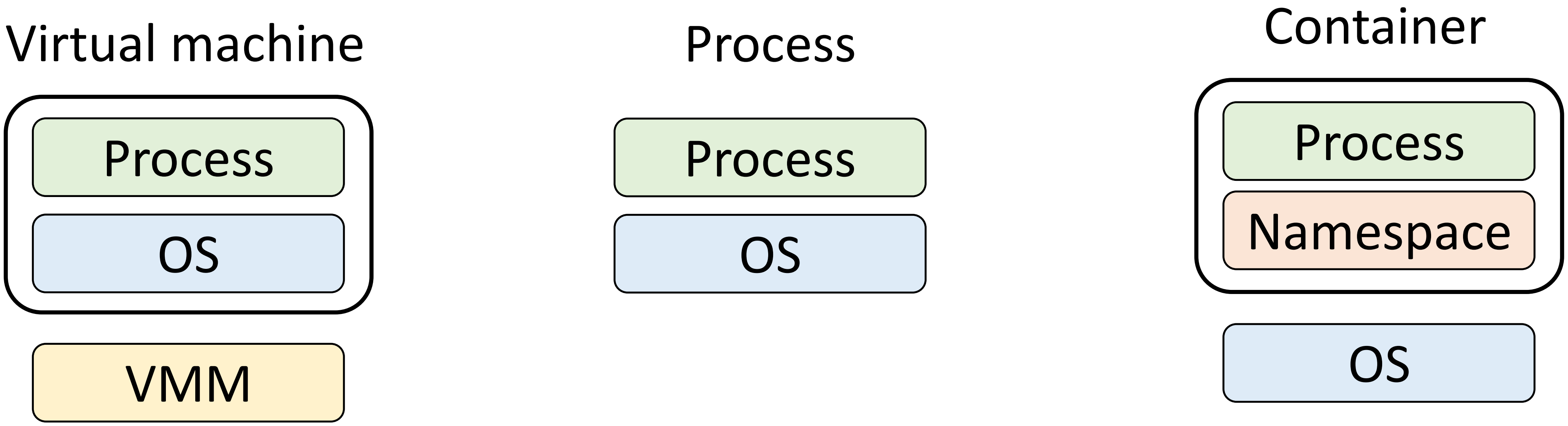
Passive backup mechanisms: High latency overhead (10s of milliseconds)

Root cause: Coupling of latency overhead and checkpointing interval

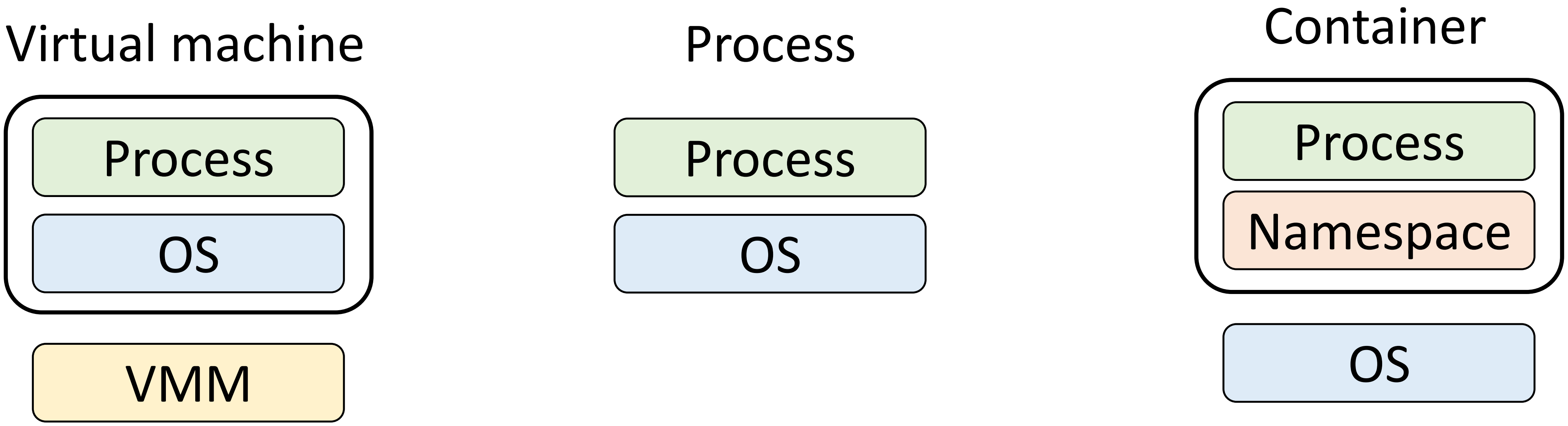
Solution: Hybrid replication – combine checkpointing with execution replay

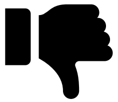
- Outputs release decoupled from checkpoint commitment
- On primary failure
  - Restore the last checkpoint on backup
  - Backup replays primary execution up to the last released outputs

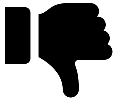
Choice of Granularity of Replication



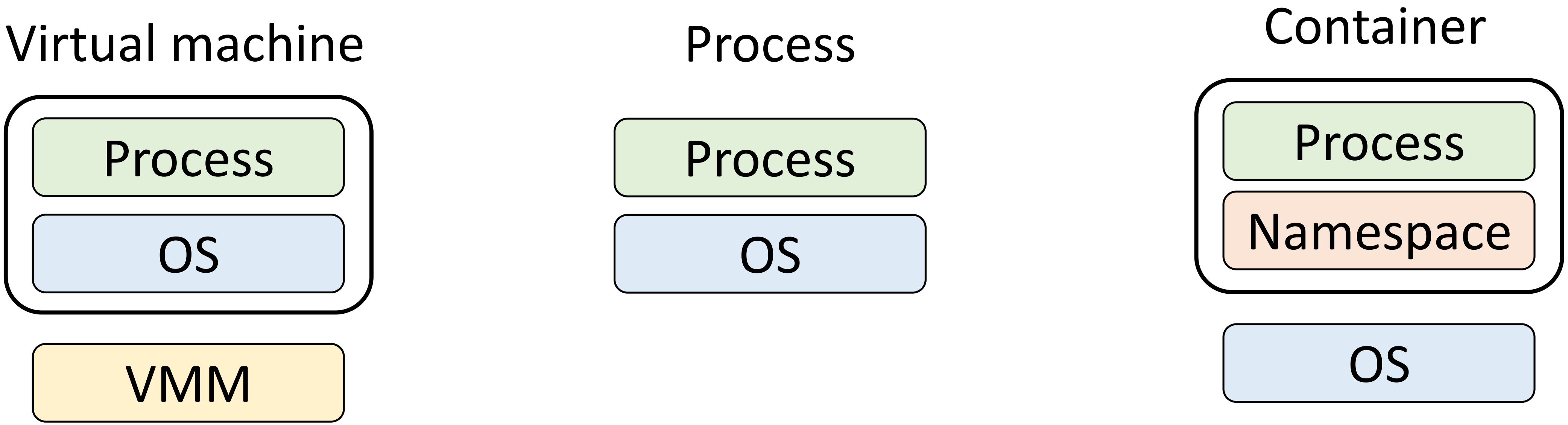
Choice of Granularity of Replication




High runtime overheads

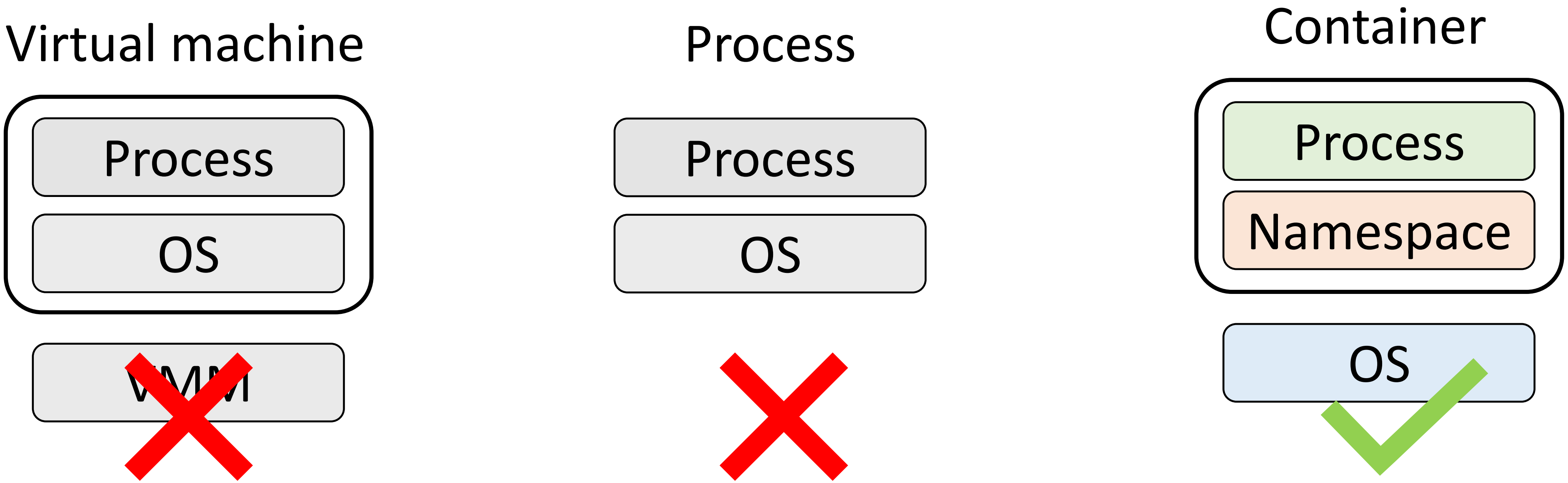

Tracking OS nondeterministic events

Choice of Granularity of Replication



- High runtime overheads
- Tracking OS nondeterministic events
- Naming conflicts e.g., process ID

Choice of Granularity of Replication

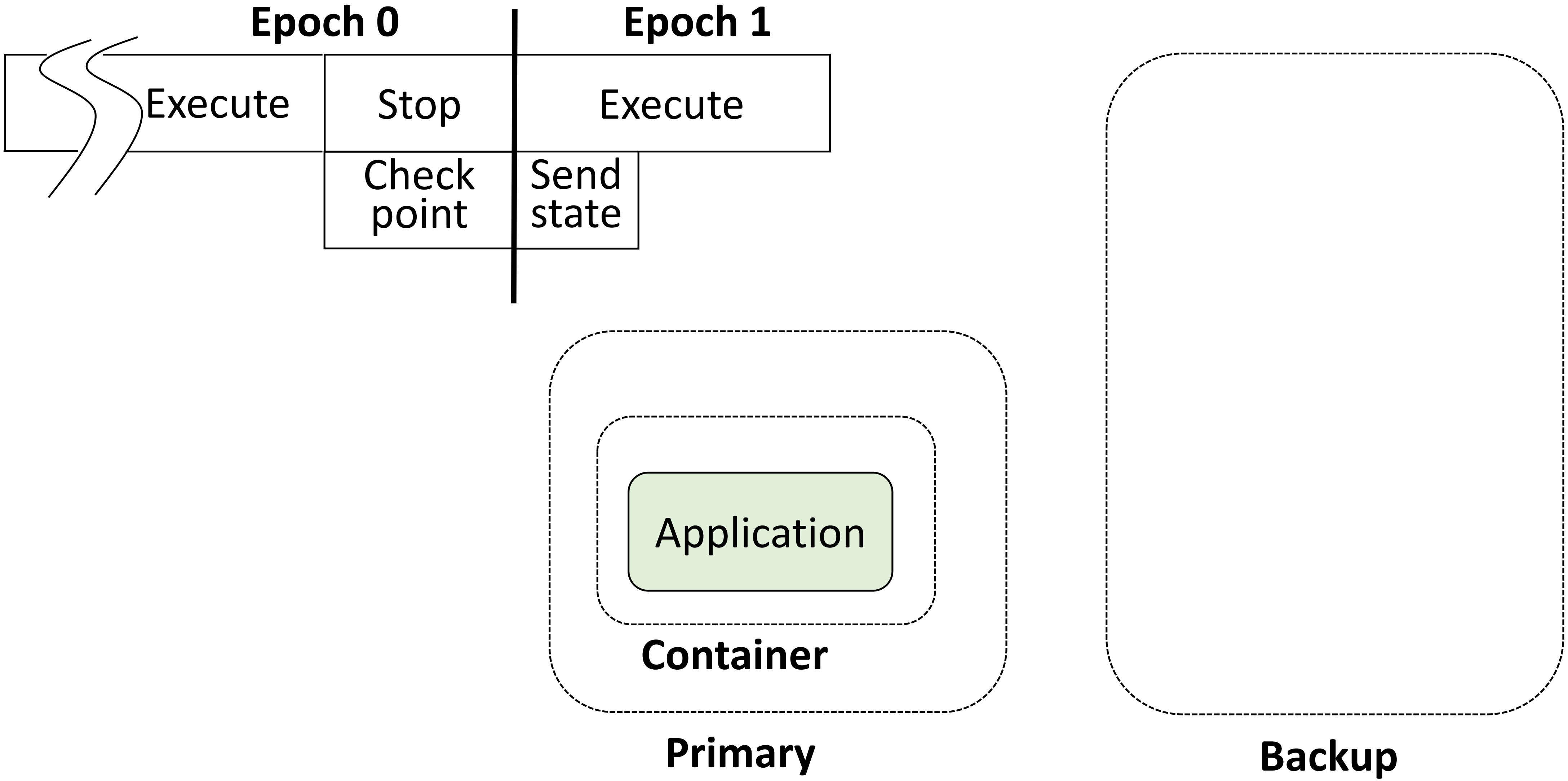


- High runtime overheads
- Tracking OS nondeterministic events

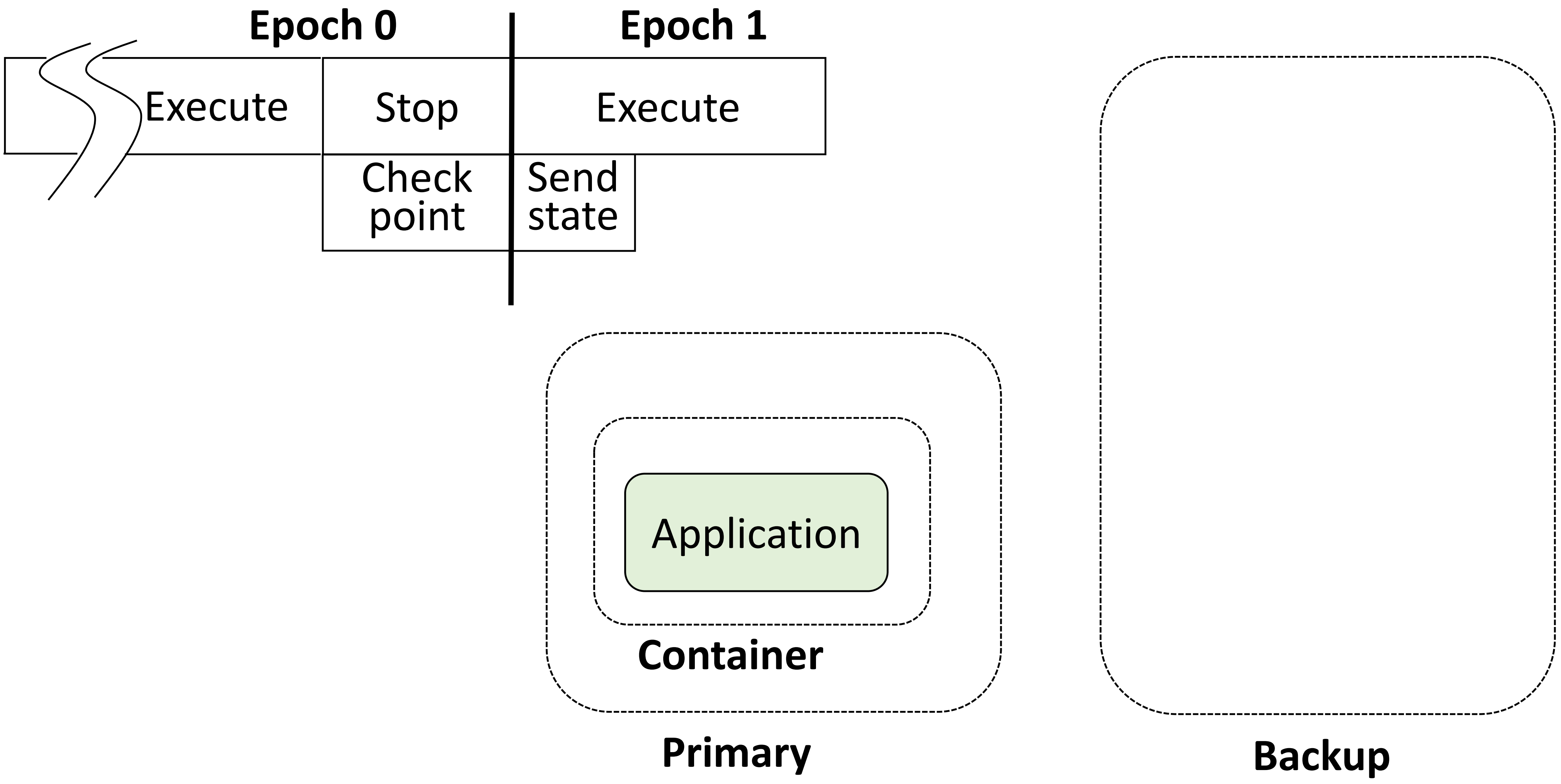
- Naming conflicts e.g., process ID

- Resolves limitations of processes/ VMs

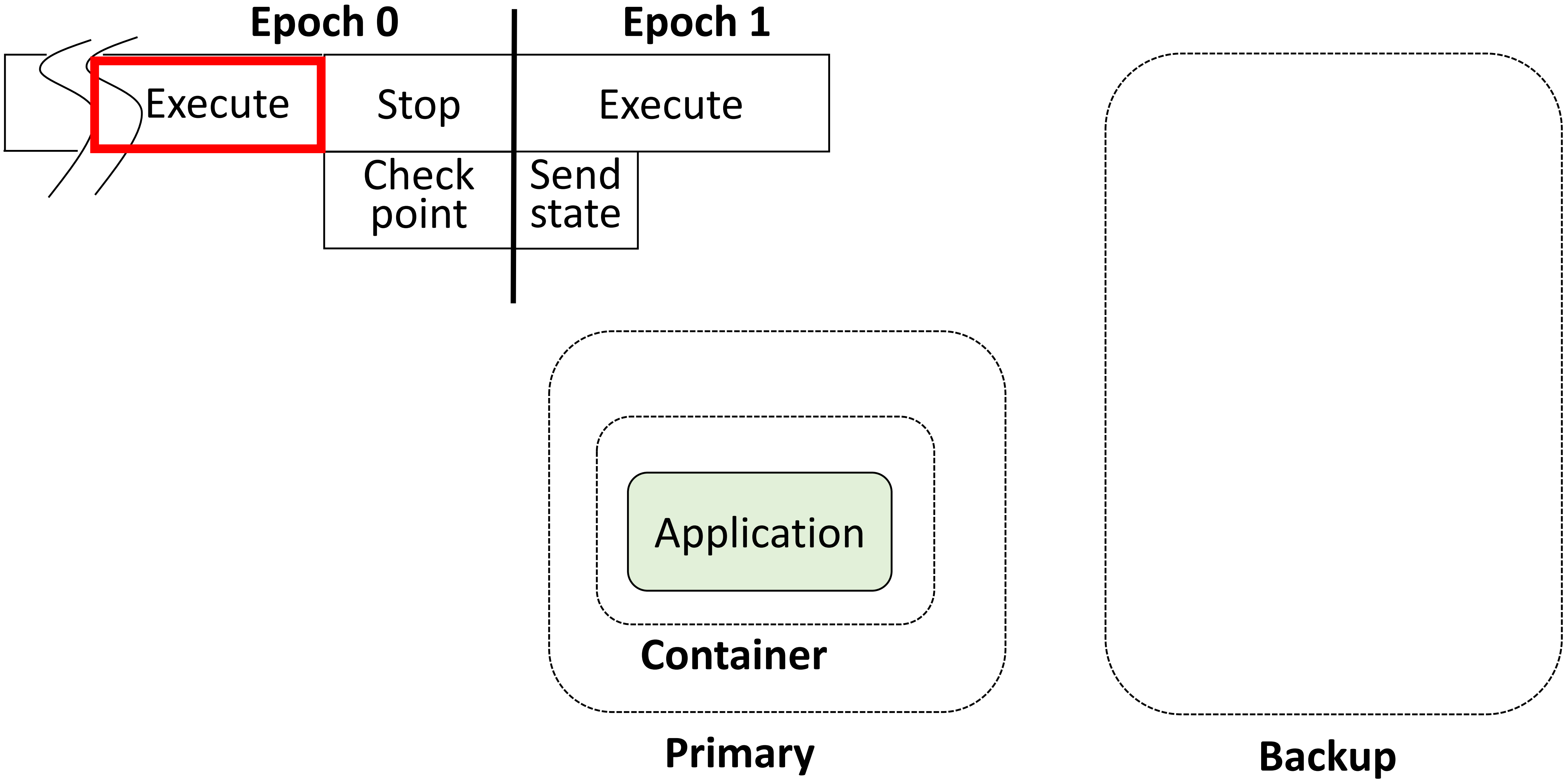
Normal operation



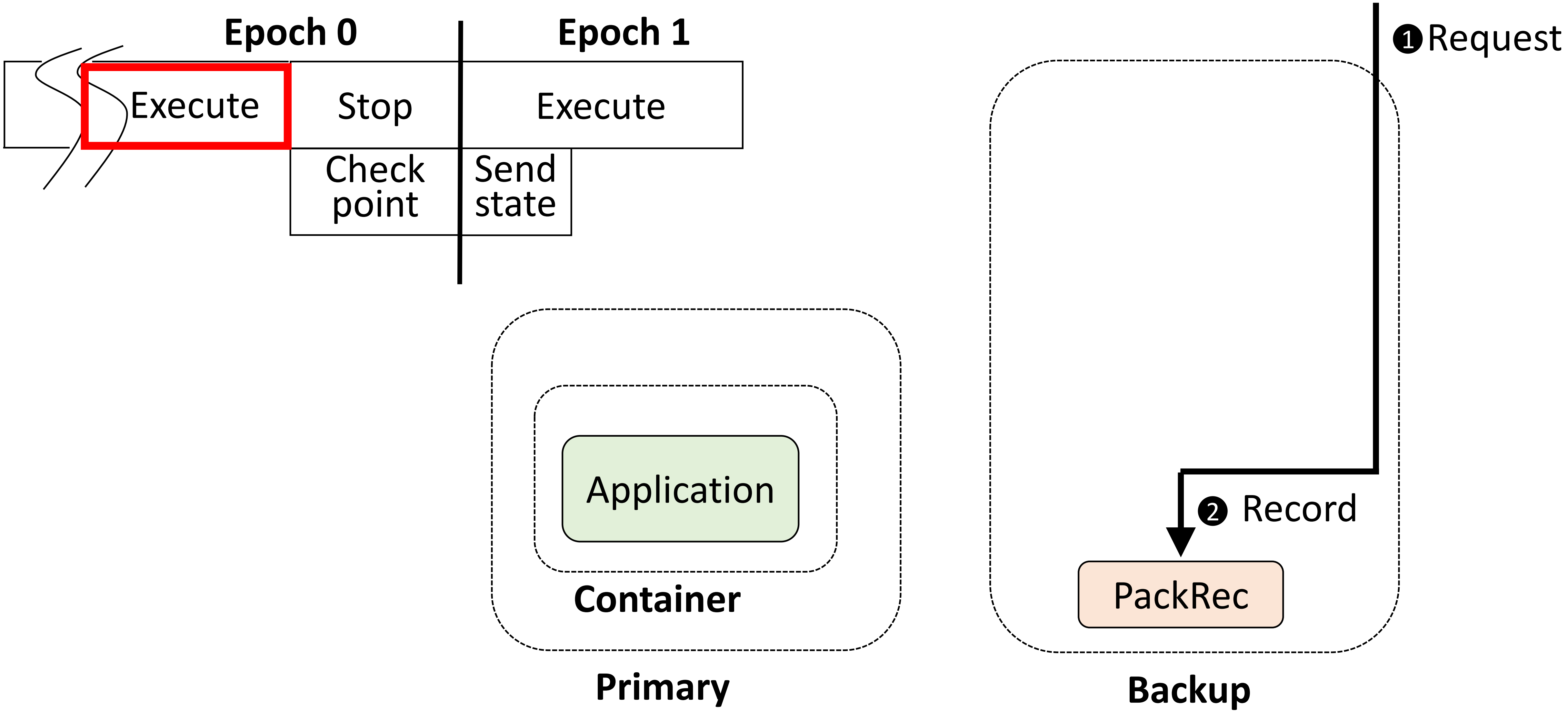
Normal operation



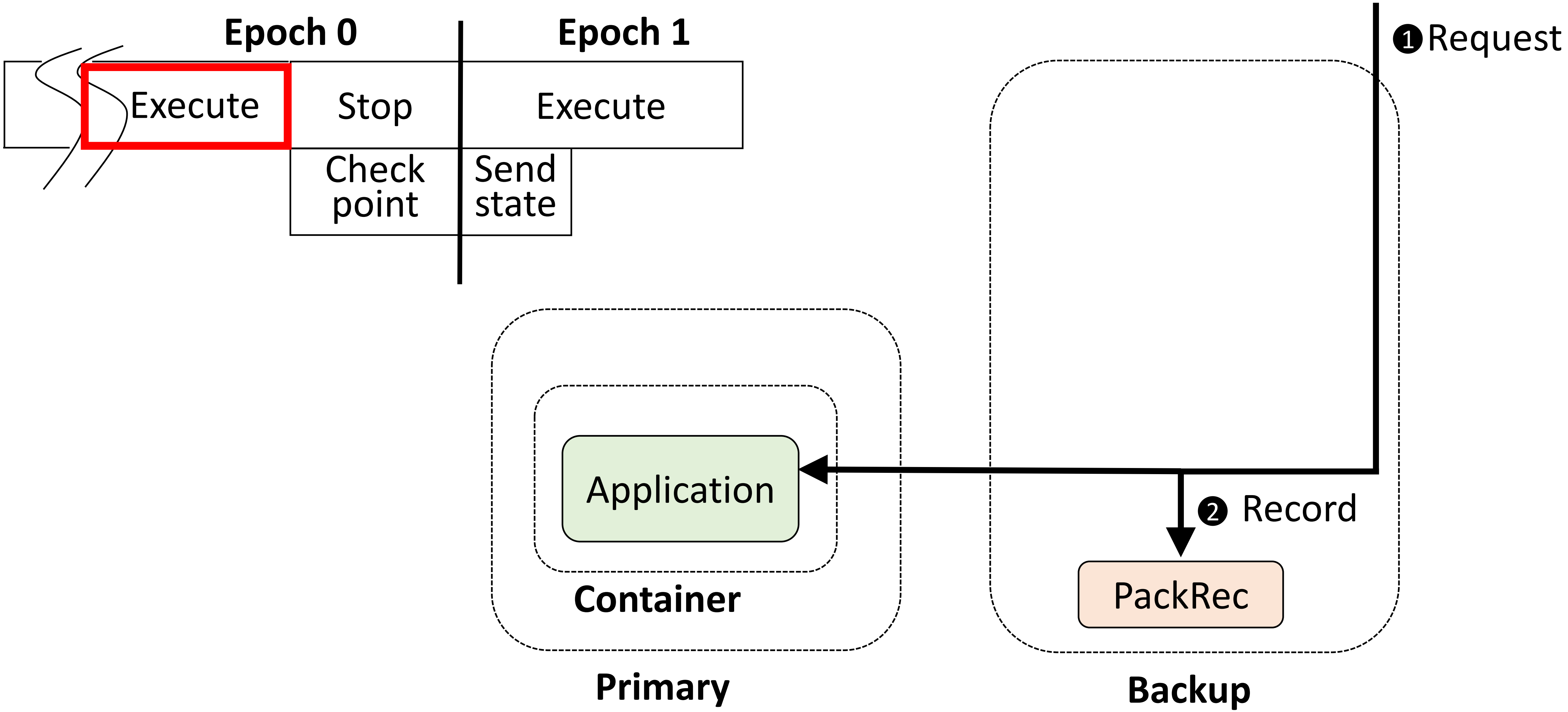
Normal operation



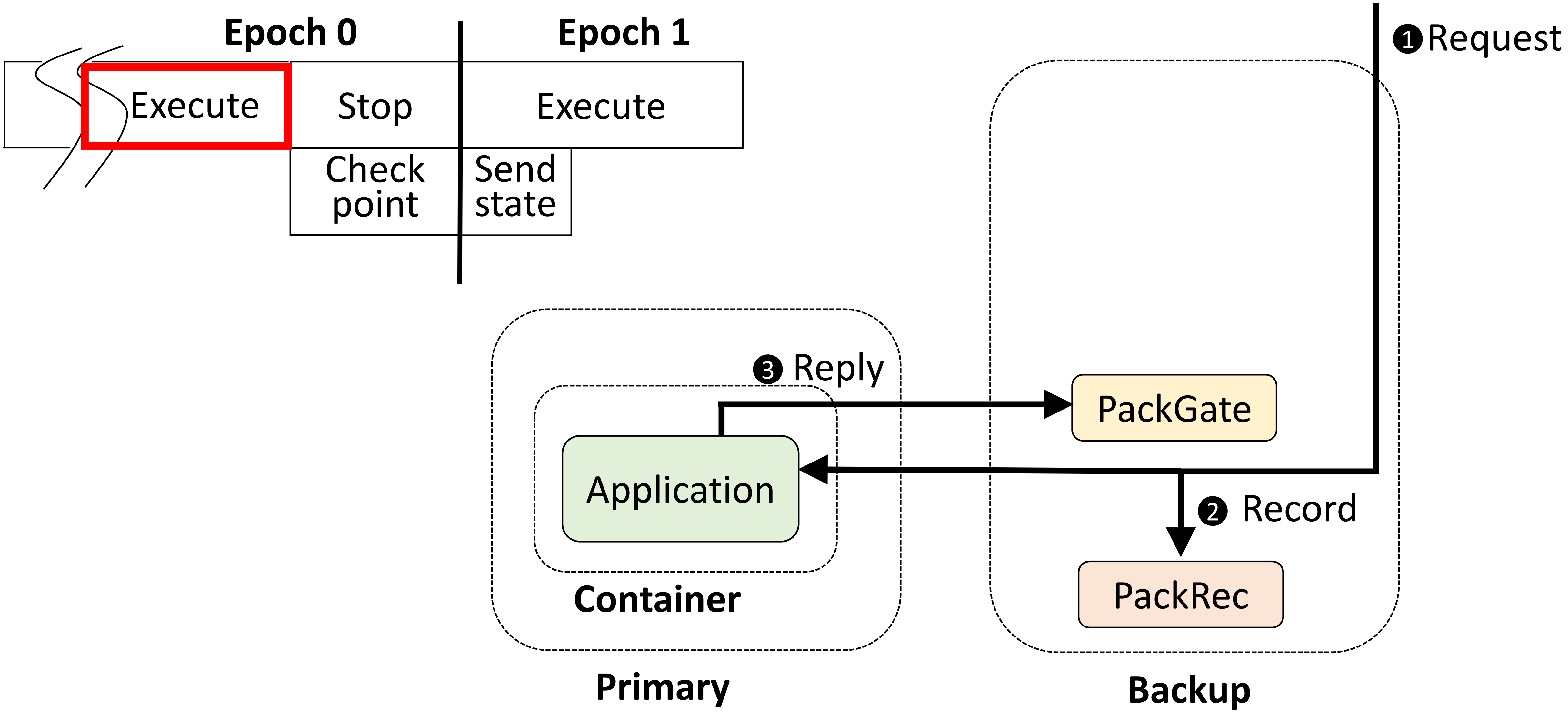
Normal operation



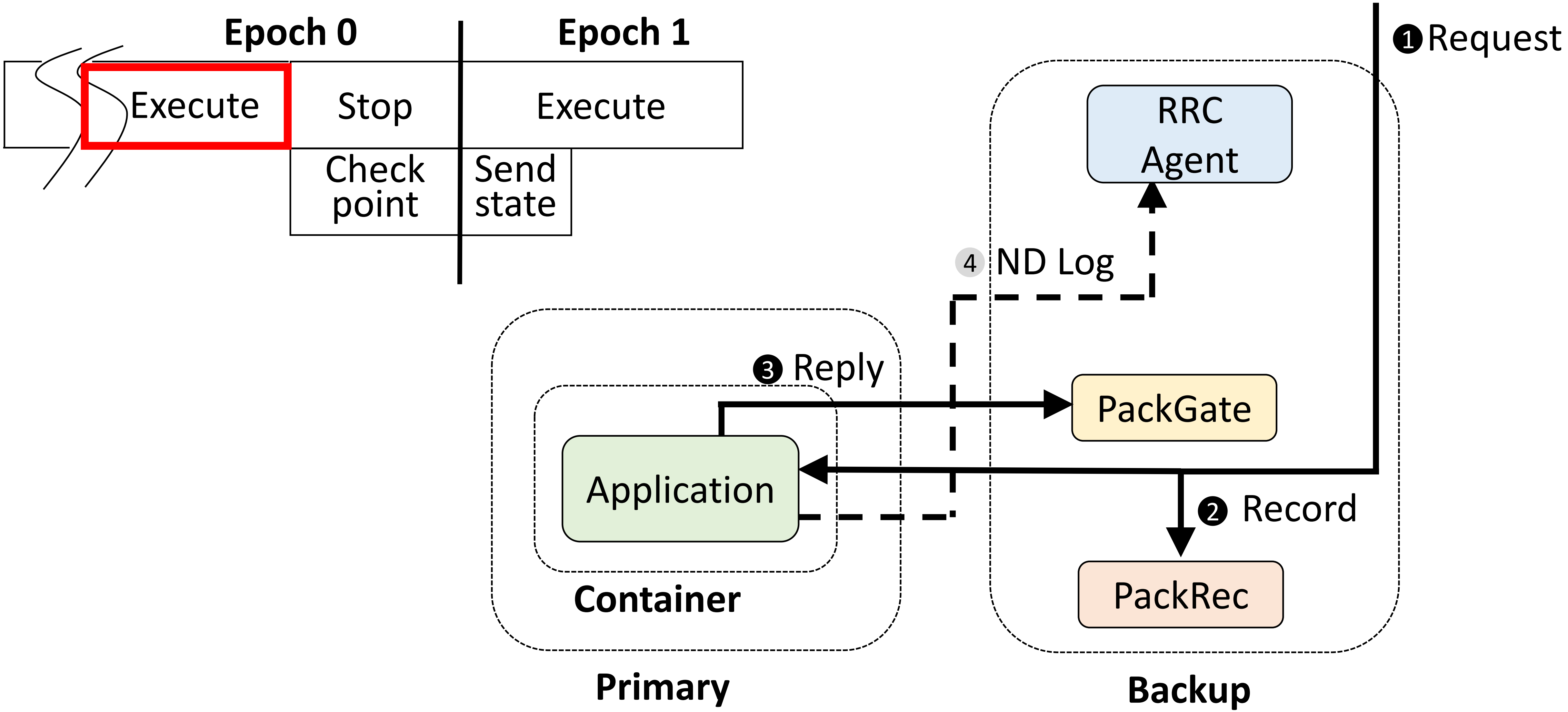
Normal operation



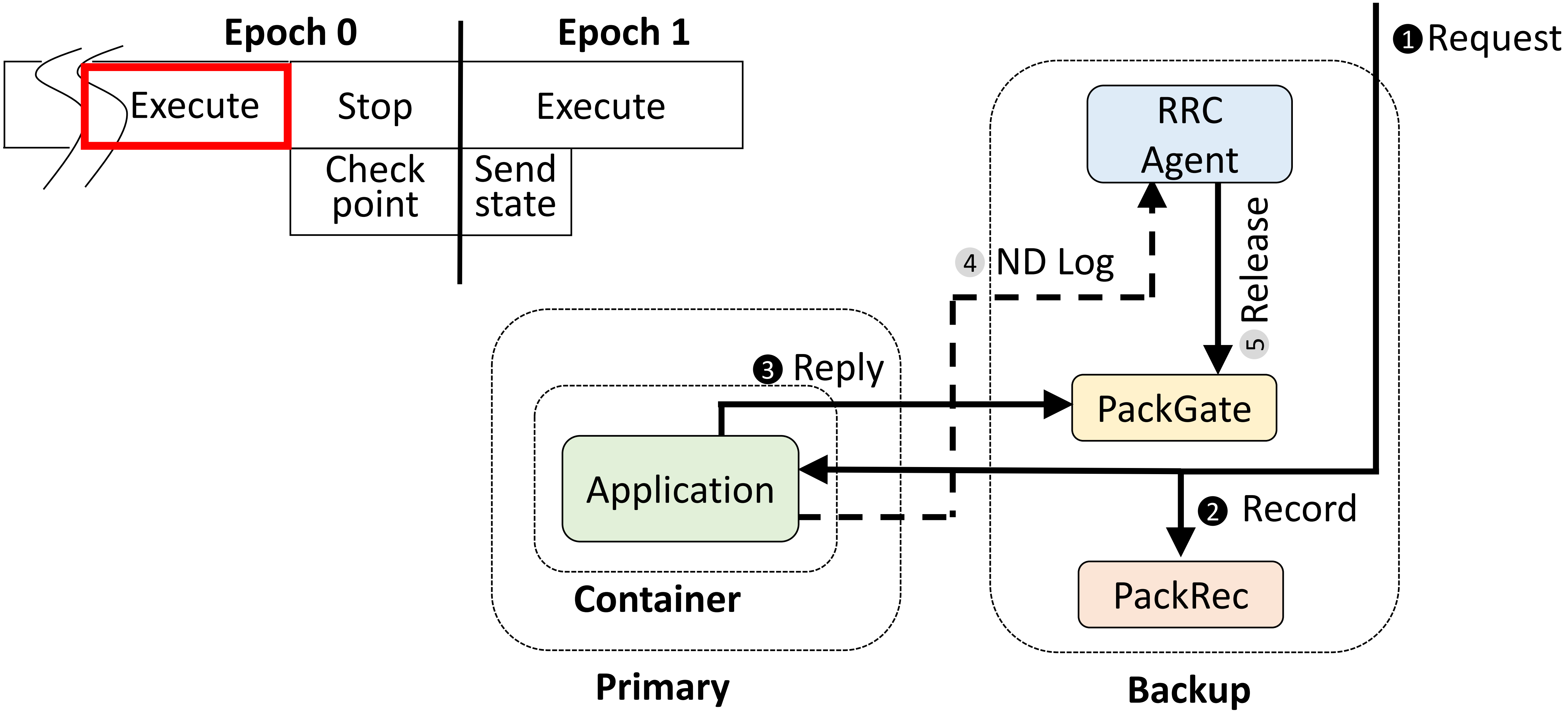
Normal operation



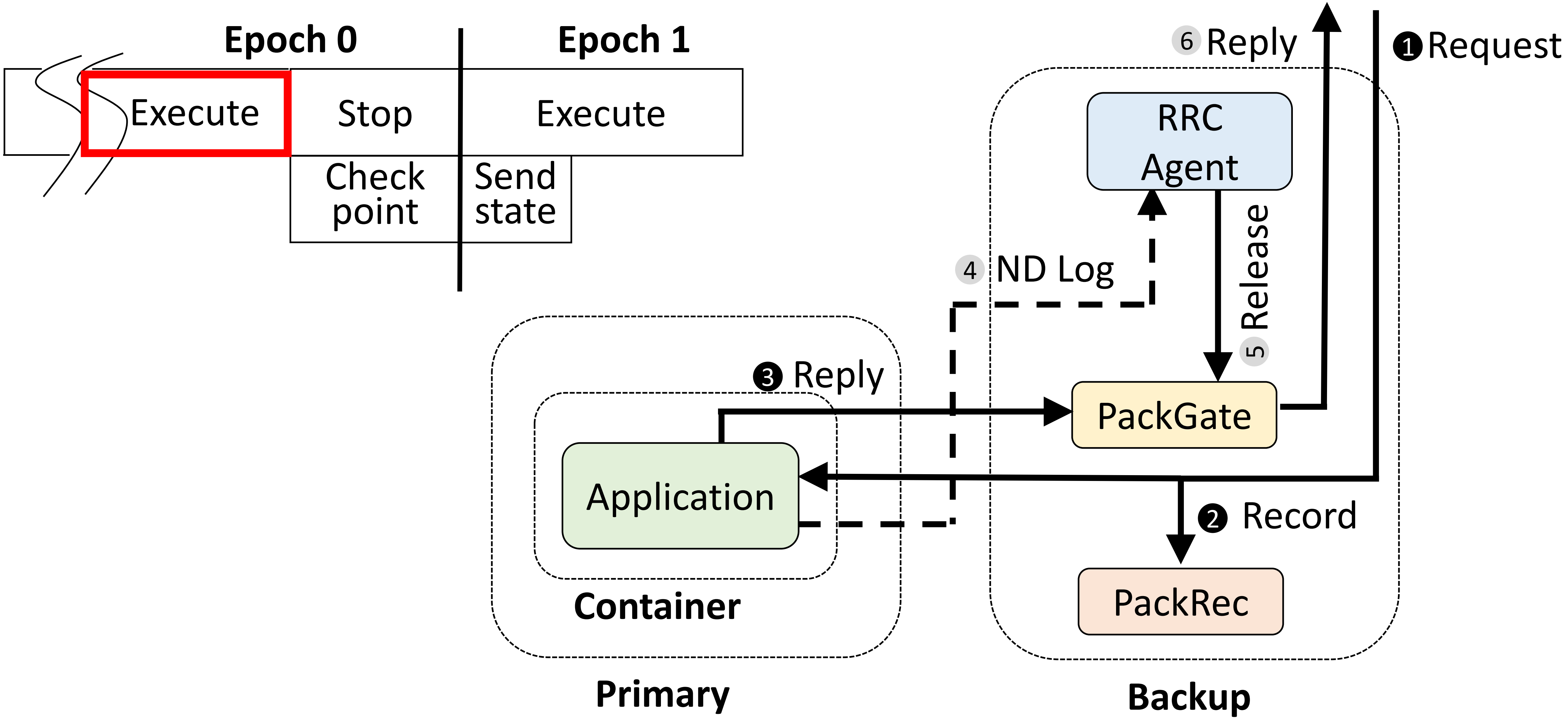
Normal operation



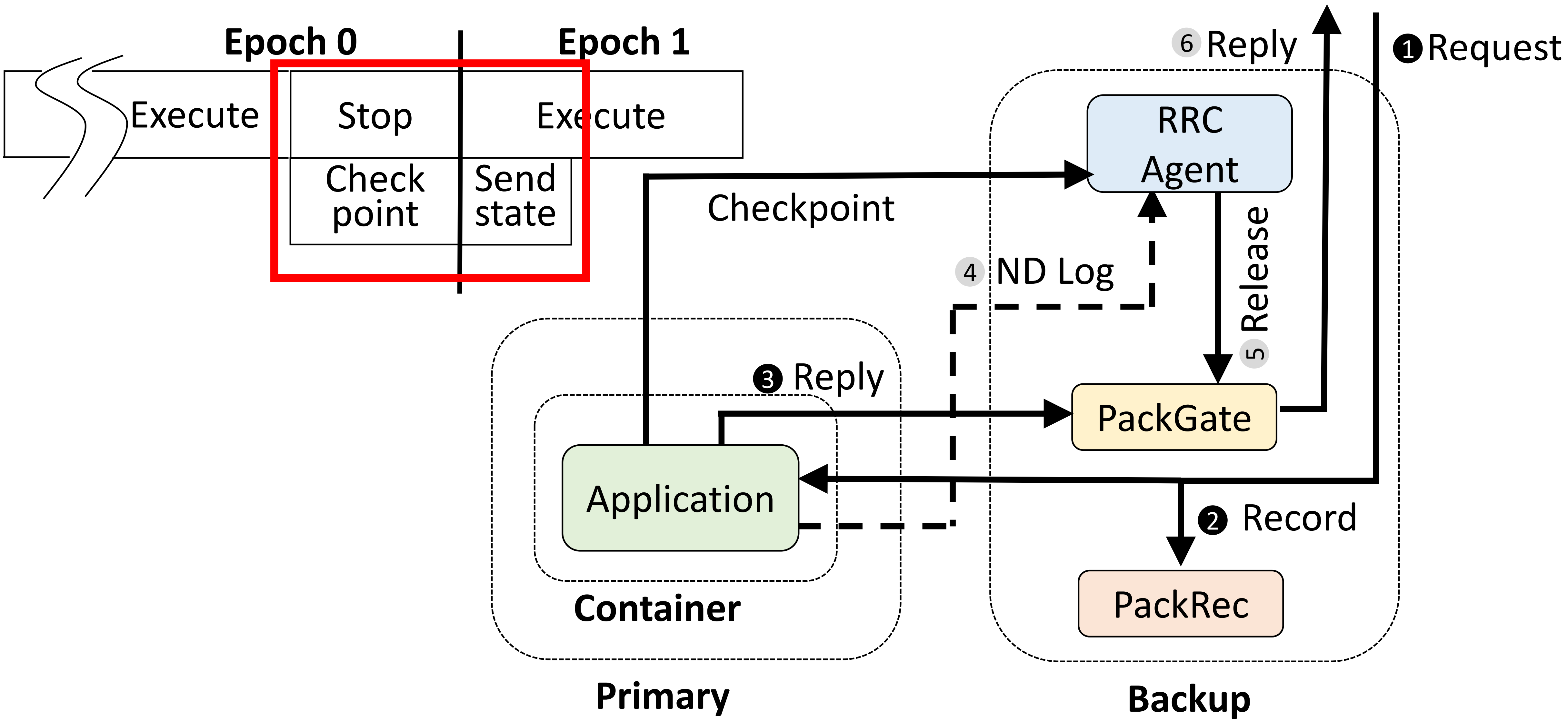
Normal operation



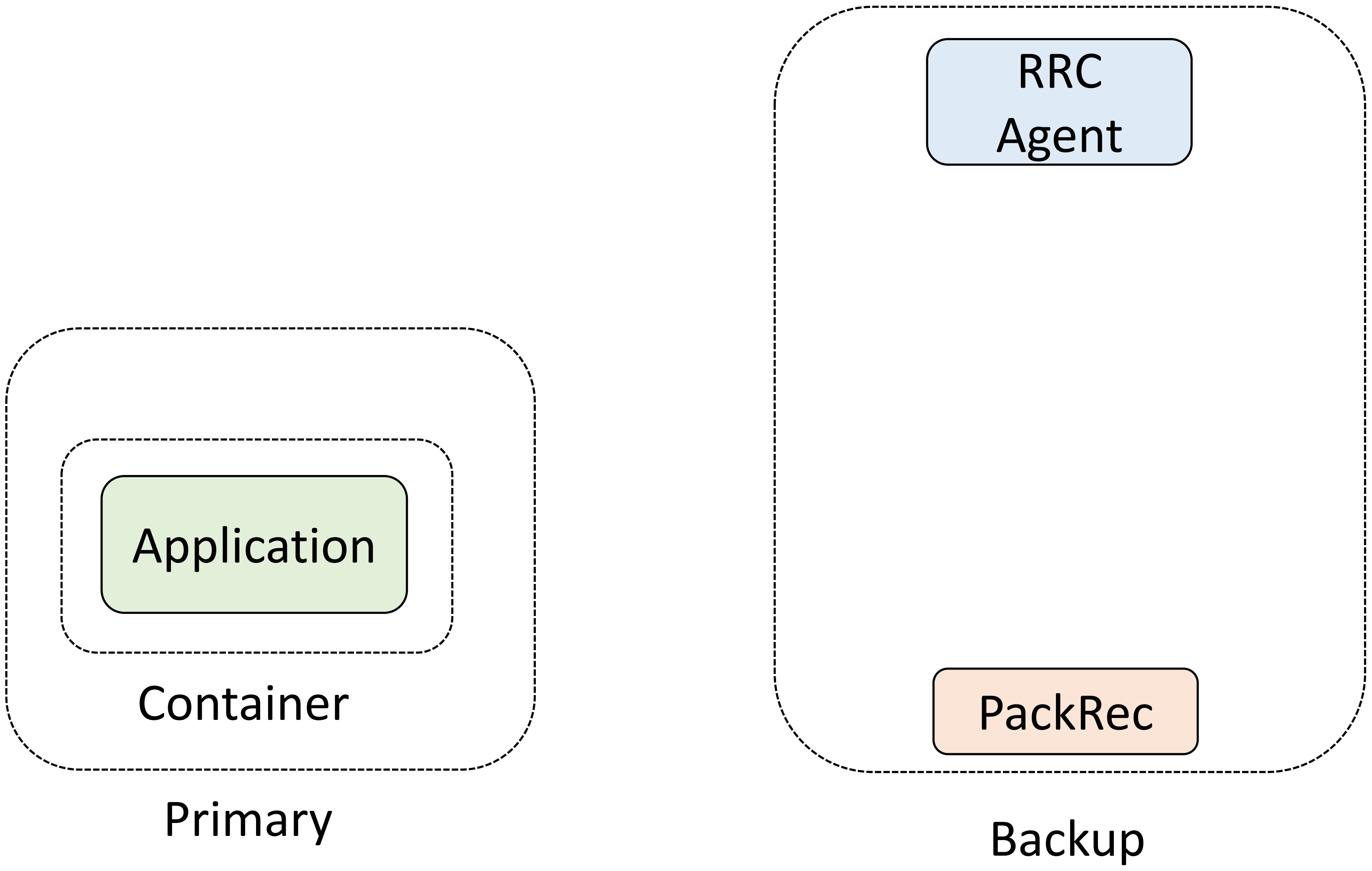
Normal operation



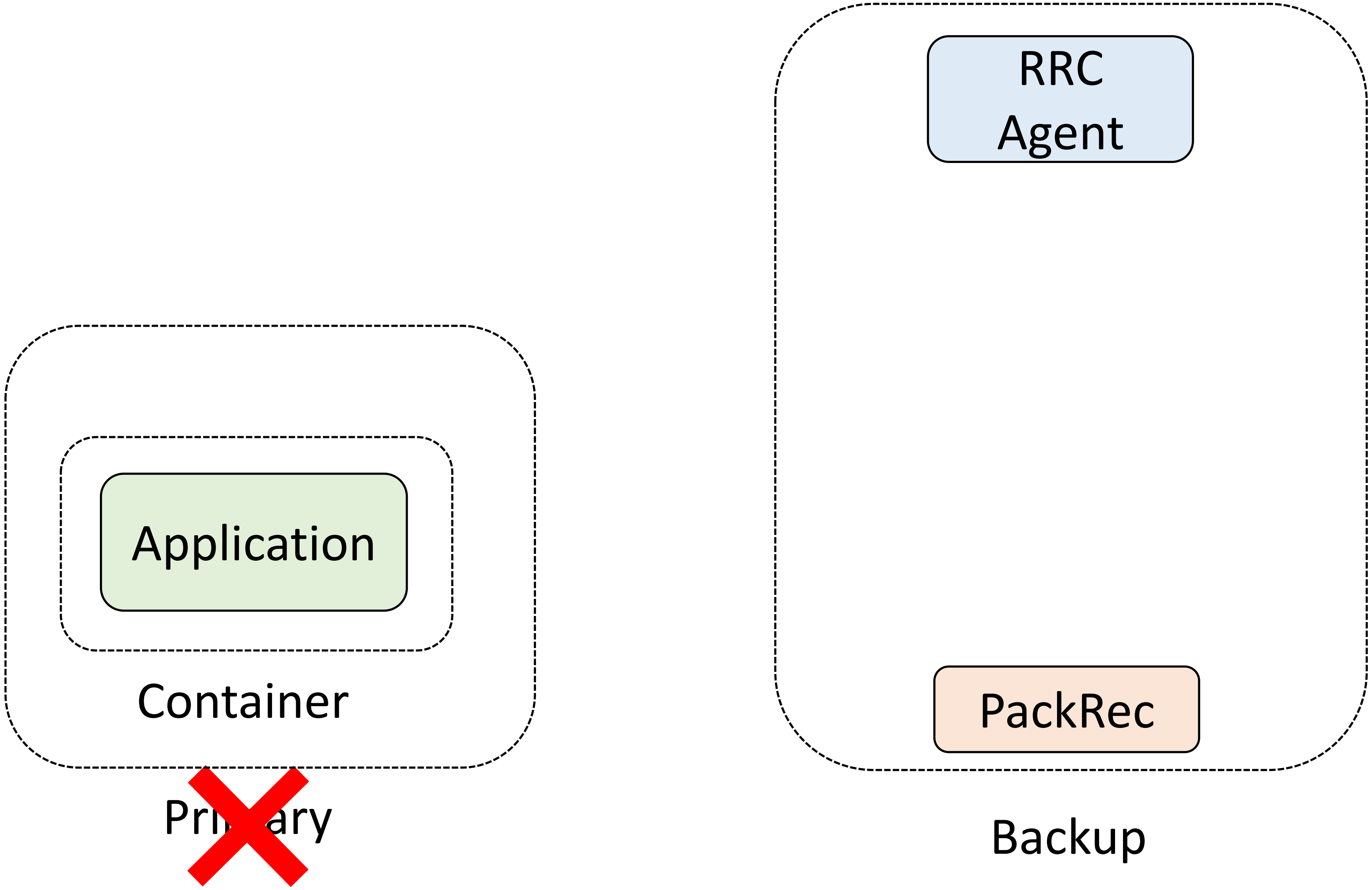
Normal operation



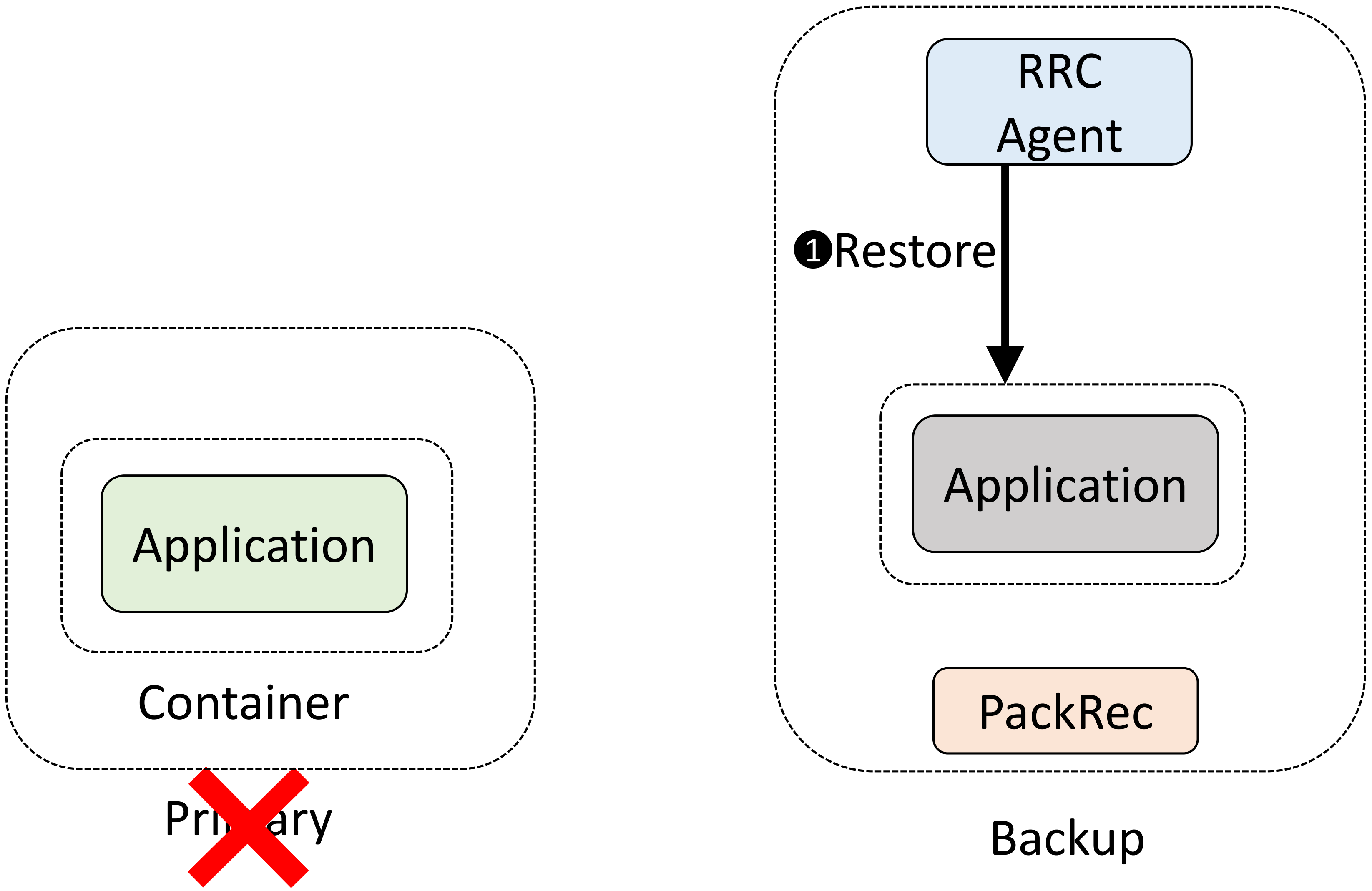
Handling Primary Failure



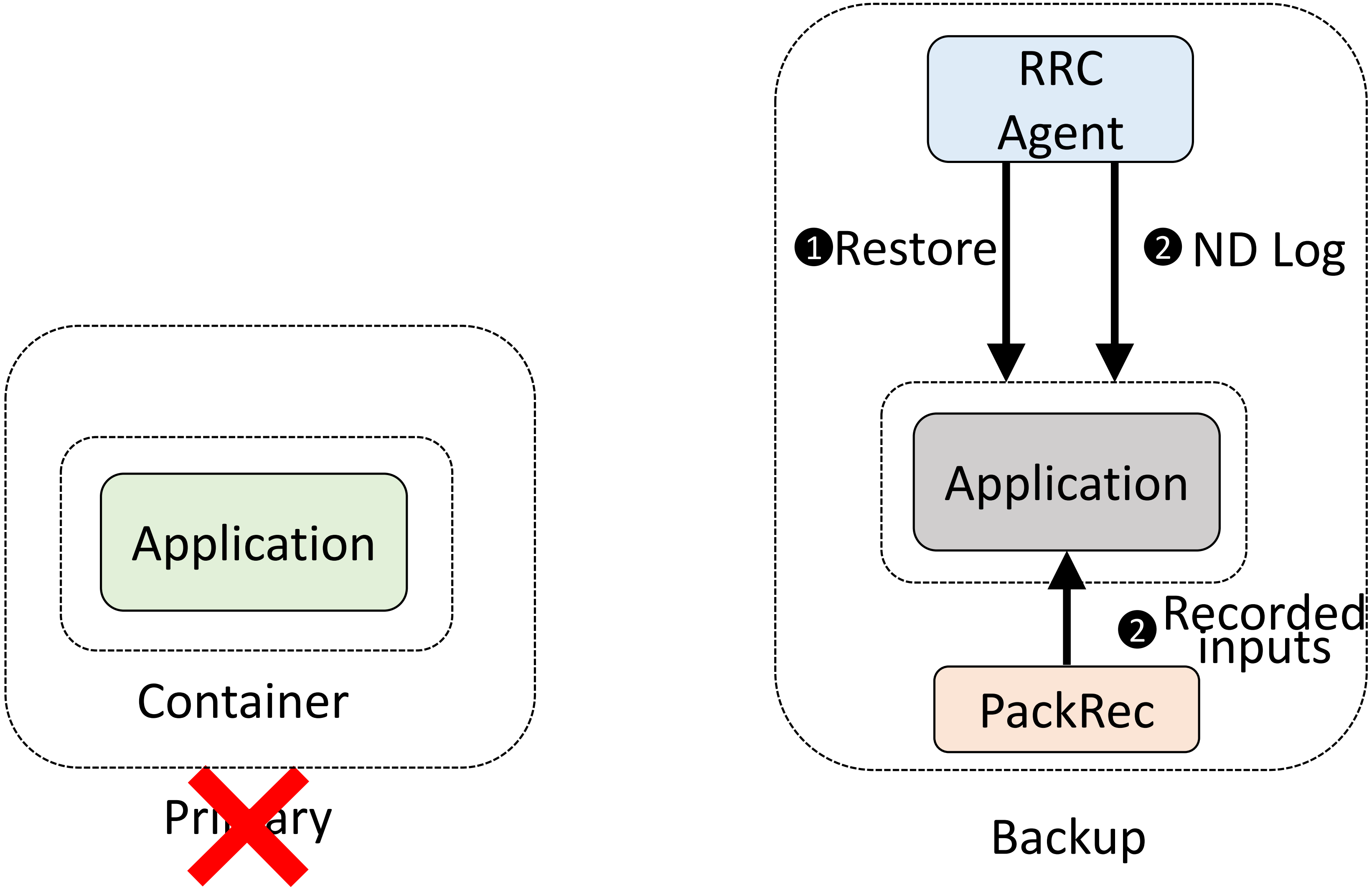
Handling Primary Failure



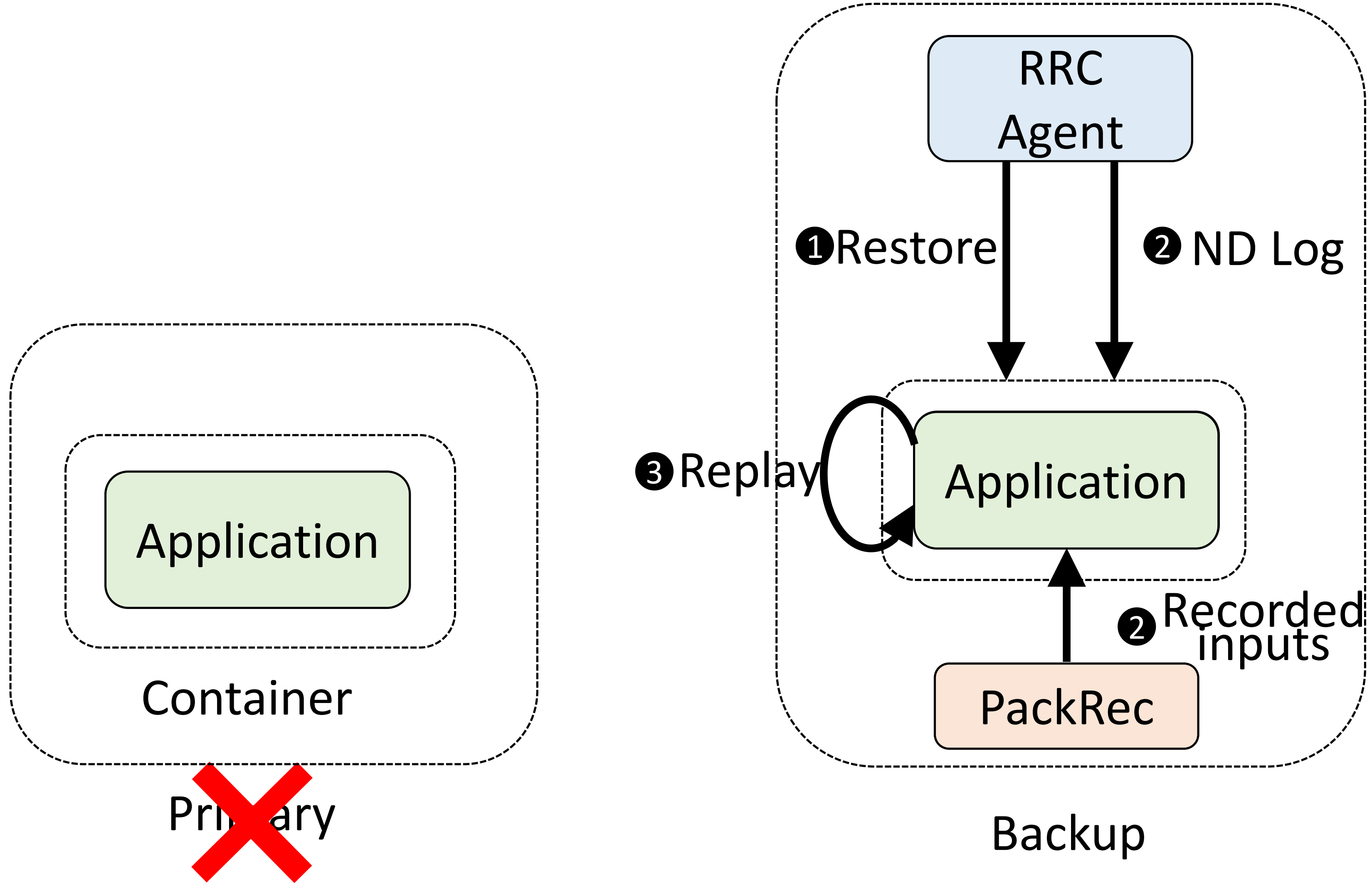
Handling Primary Failure



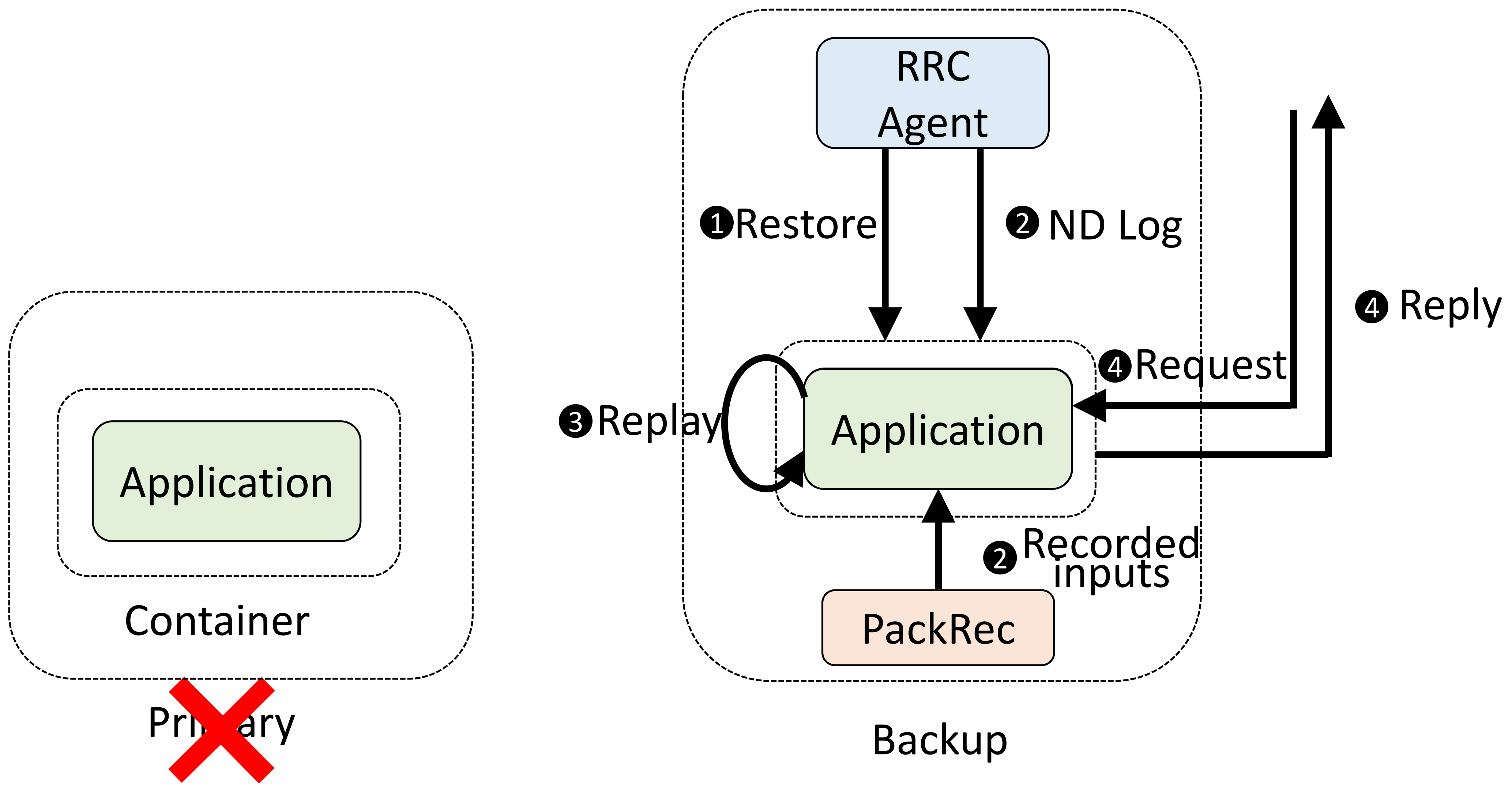
Handling Primary Failure



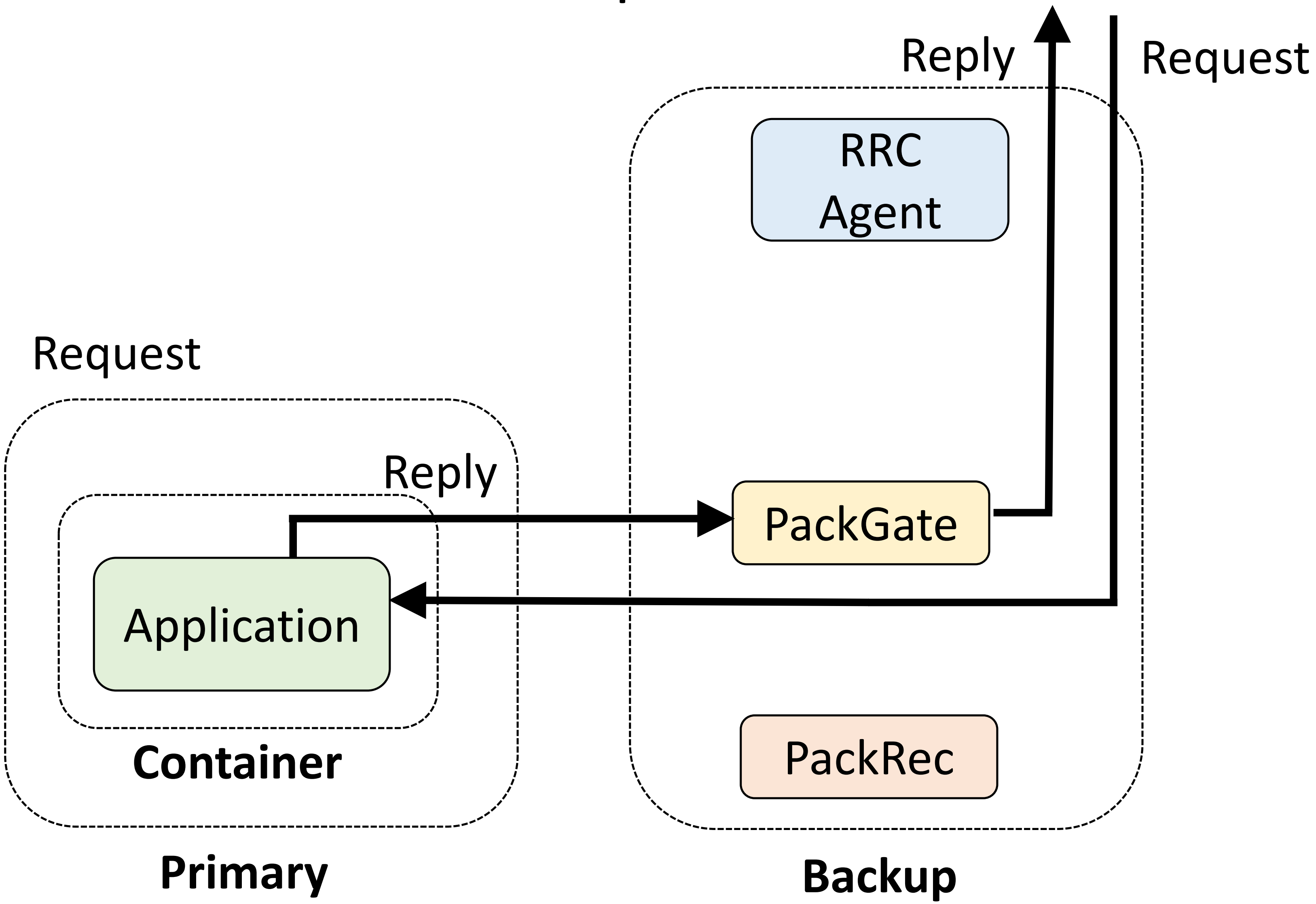
Handling Primary Failure



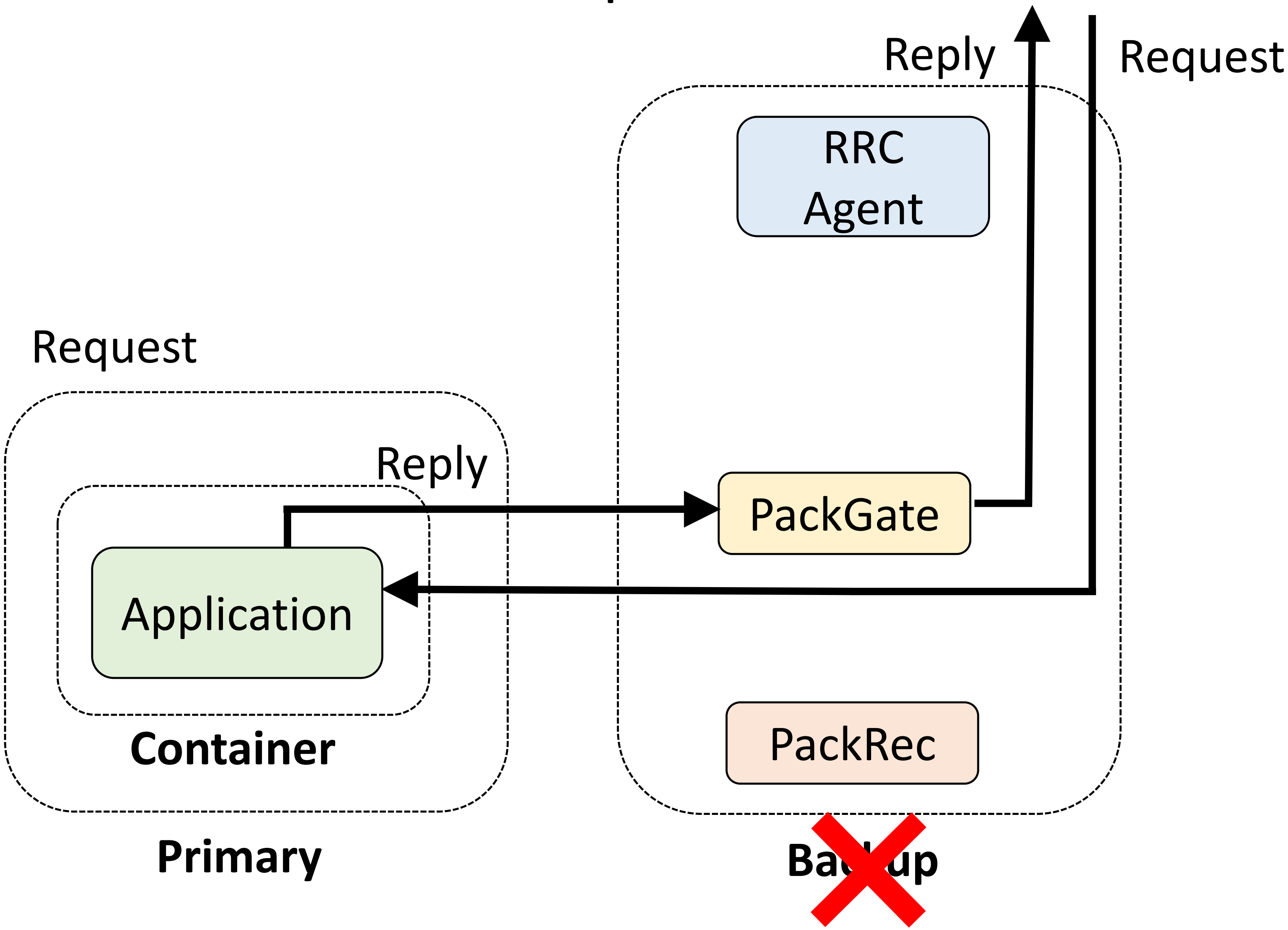
Handling Primary Failure



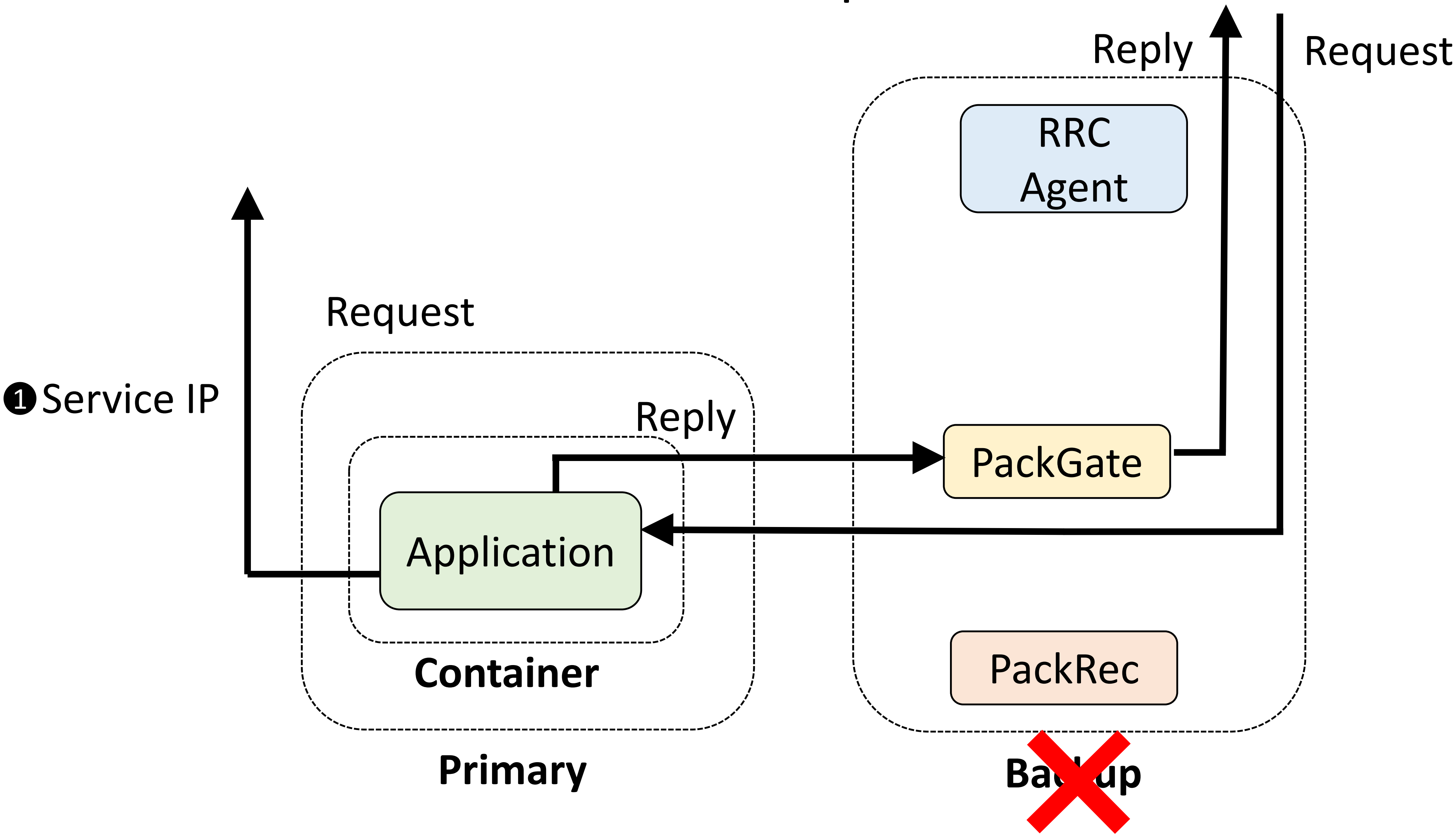
RRC: Backup Failure



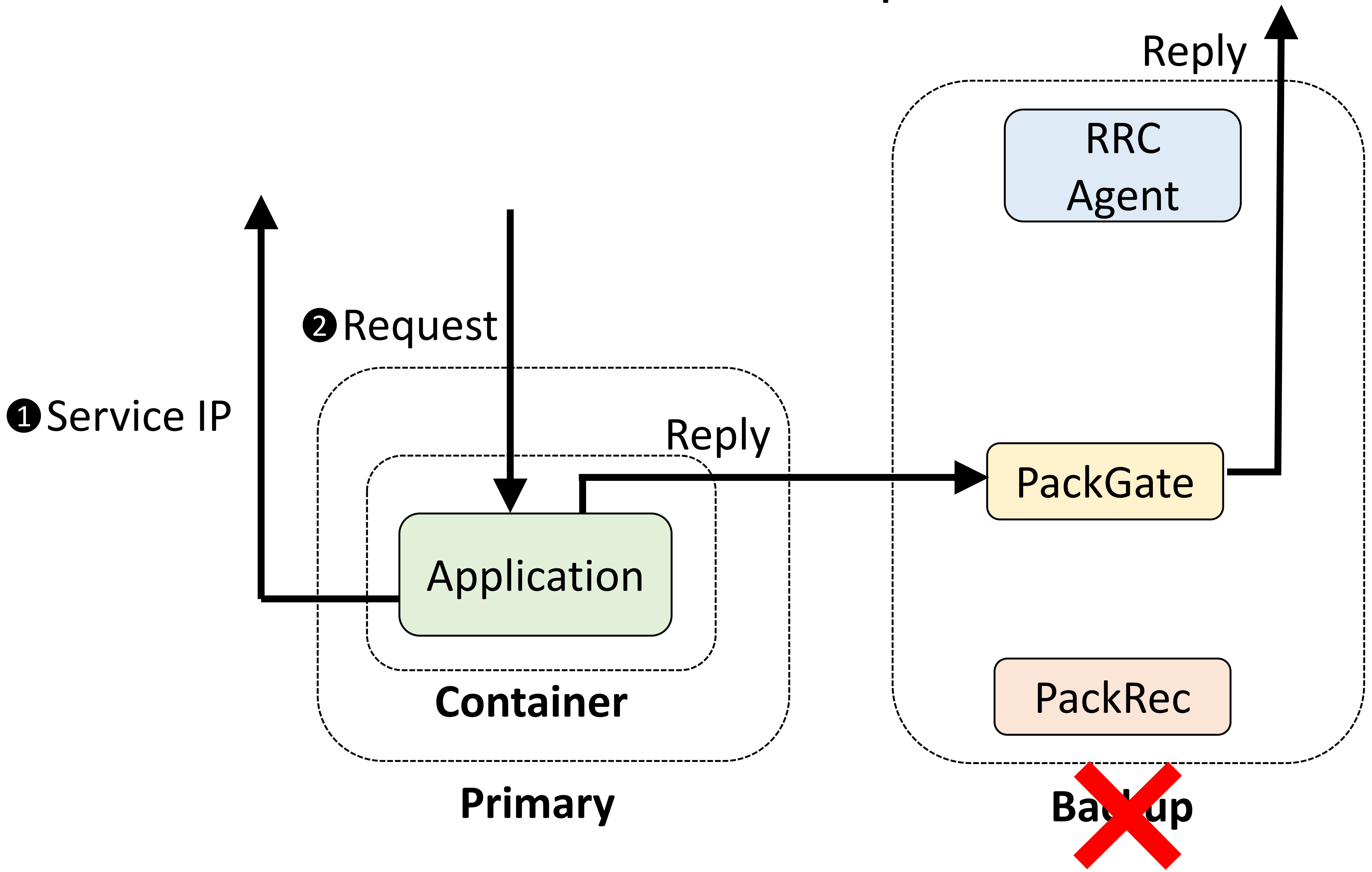
RRC: Backup Failure



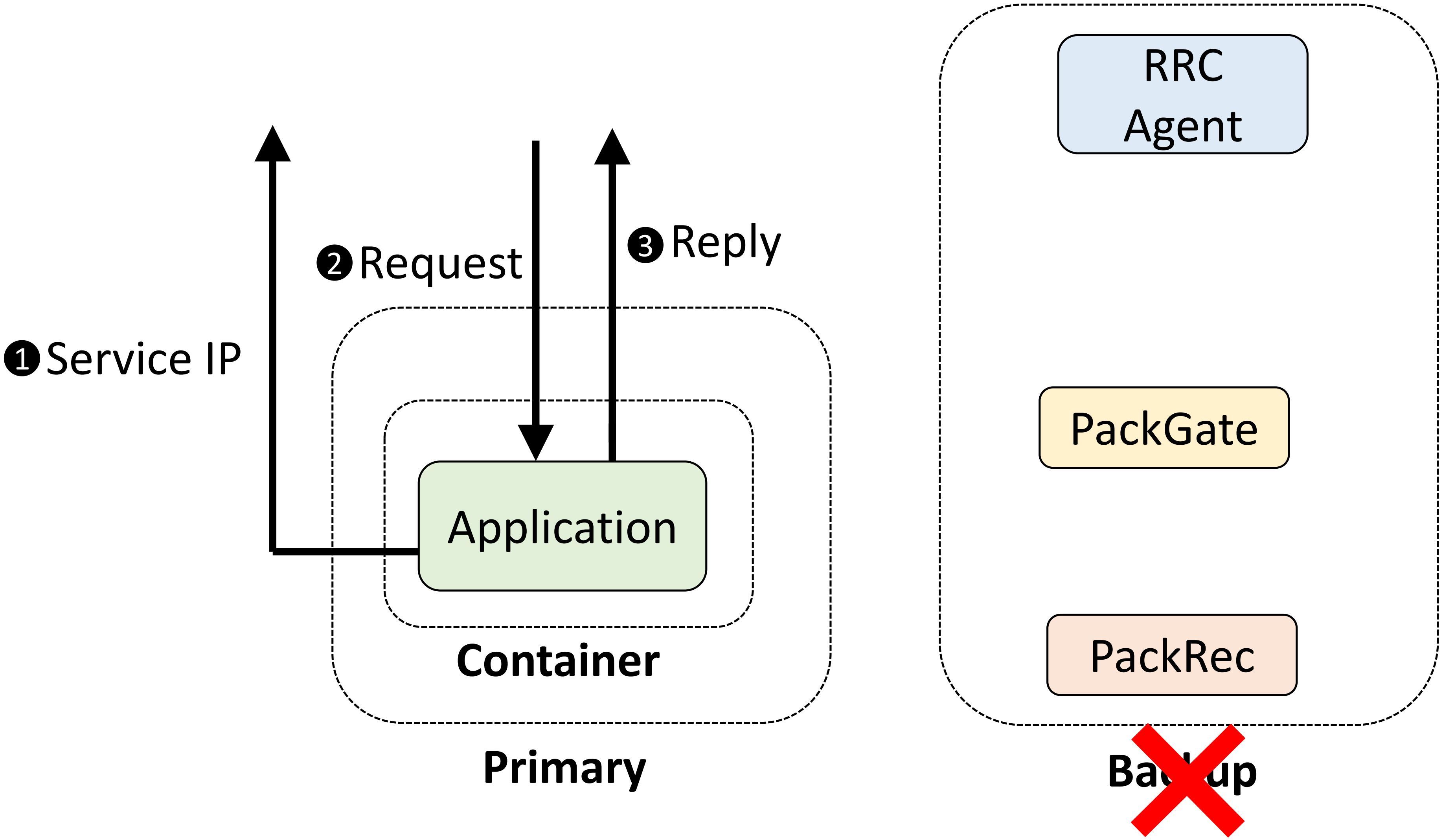
RRC: Backup Failure



RRC: Backup Failure



RRC: Backup Failure



Talk Outline

- Preface
- Motivation
- RRC overview
- **Overcoming design and implementation challenges**
- Evaluation

Key Design and Implementation Challenges

- Minimizing pause time during checkpointing
- Handling untracked nondeterministic events
- Robust integration of asynchronous checkpointing and recording of nondeterministic events
- Minimizing the overhead for collection and transfer of nondeterministic event logs
- Integration of TCP failover with replay during recovery

Key Design and Implementation Challenges

- Minimizing pause time during checkpointing
- Handling untracked nondeterministic events
- Robust integration of asynchronous checkpointing and recording of nondeterministic events
- Minimizing the overhead for collection and transfer of nondeterministic event logs
- Integration of TCP failover with replay during recovery

Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state

→ Execution must pause during checkpointing

→ Service pause time during checkpointing

Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state

→ Execution must pause during checkpointing

→ Service pause time during checkpointing

Container: tight state coupling with the underlying kernel

→ Significant in-kernel container state must be checkpointed

→ Retrieving the in-kernel container state is slow**: thousands of syscalls**

Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state

→ Execution must pause during checkpointing

→ Service pause time during checkpointing

Container: tight state coupling with the underlying kernel

→ Significant in-kernel container state must be checkpointed

→ Retrieving the in-kernel container state is slow**: thousands of syscalls**

Checkpointing a container is **slow**

Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state

→ Execution must pause during checkpointing

→ Service pause time during checkpointing

Container: tight state coupling with the underlying kernel

→ Significant in-kernel container state must be checkpointed

→ Retrieving the in-kernel container state is slow**: thousands of syscalls**

Checkpointing a container is **slow**

Challenge: minimize the pause time despite slow checkpointing

Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

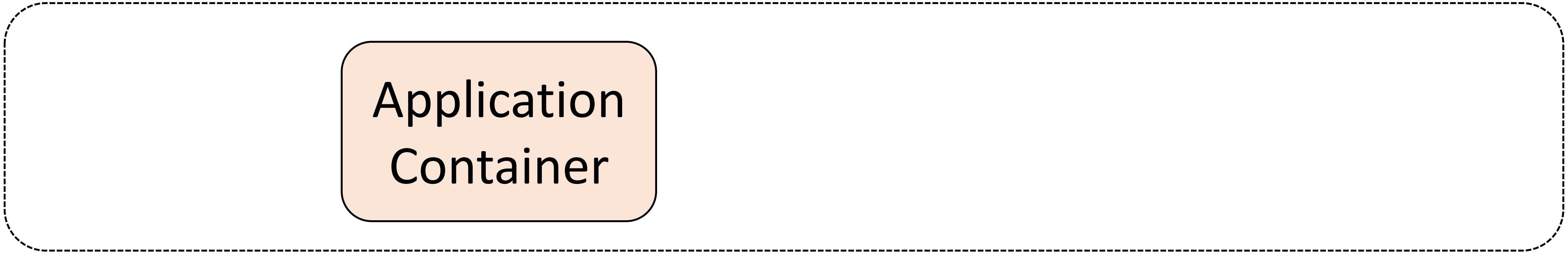
Design: New kernel primitive – Container fork

Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

Design: New kernel primitive – Container fork

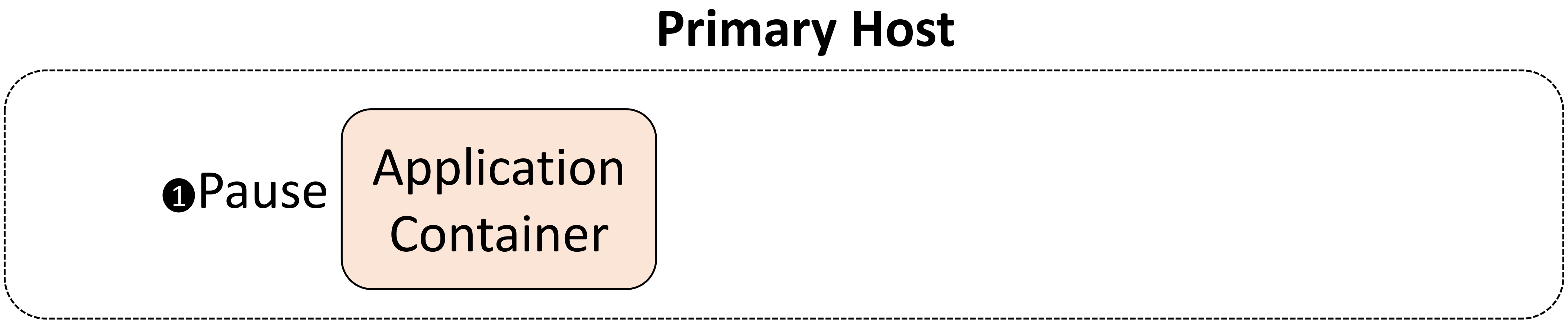
Primary Host



Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

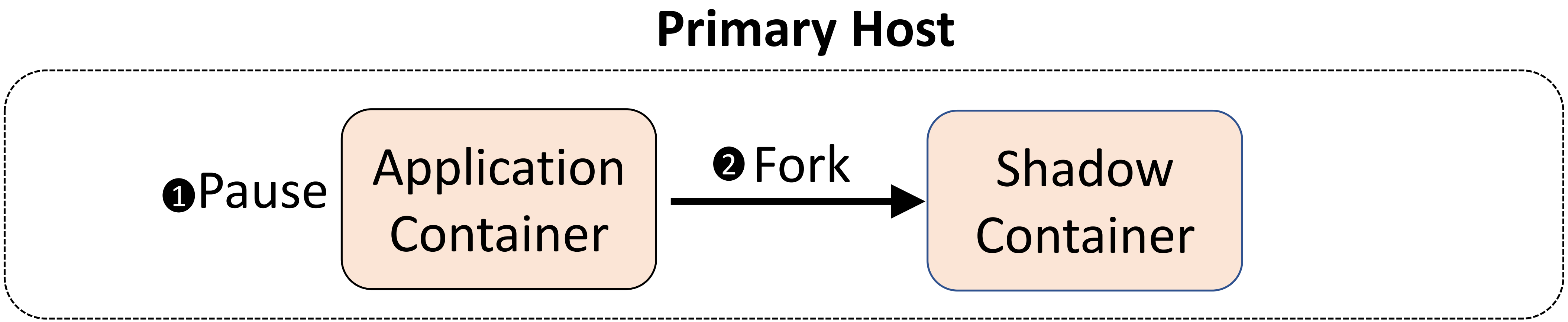
Design: New kernel primitive – Container fork



Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

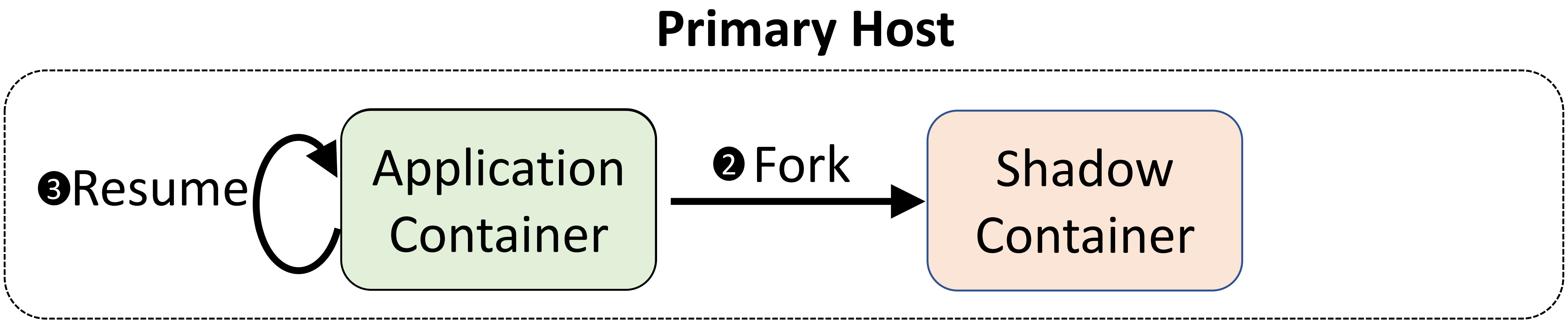
Design: New kernel primitive – Container fork



Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

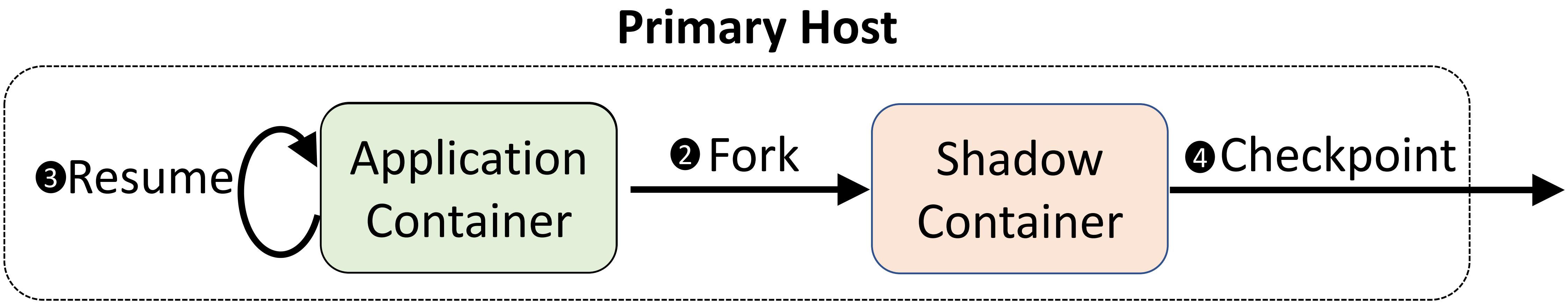
Design: New kernel primitive – Container fork



Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

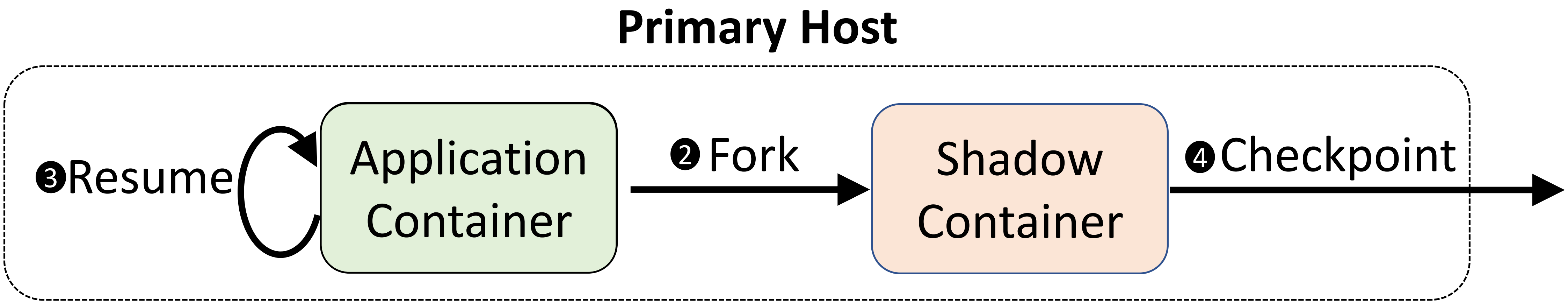
Design: New kernel primitive – Container fork



Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state
from container execution

Design: New kernel primitive – Container fork



Result: Service Pause time [5.9ms - 42.9ms] → [0.5ms - 3.2ms]

Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:

Execution replay **only** during recovery

→ Vulnerability **only** to nondeterministic events occurring during the epoch of failure

Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:

Execution replay **only** during recovery

→ Vulnerability **only** to nondeterministic events occurring during the epoch of failure

RRC's handling of nondeterministic events:

- Replay nondeterministic event logs

Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:

Execution replay **only** during recovery

→ Vulnerability **only** to nondeterministic events occurring during the epoch of failure

RRC's handling of nondeterministic events:

- Replay nondeterministic event logs

Multithreading: memory access ordering is nondeterministic

Solution:

- Record the order of all memory accesses

→ Unacceptably high overhead

Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:

Execution replay **only** during recovery

→ Vulnerability **only** to nondeterministic events occurring during the epoch of failure

RRC's handling of nondeterministic events:

- Replay nondeterministic event logs

Multithreading: memory access ordering is nondeterministic

Solution:

- Record the order of all memory accesses
  → Unacceptably high overhead
- Record the outcomes of synchronization operations
  → Challenge: data races – unsynchronized memory accesses

Data Race Considerations

- Data races are **bugs**
- Impossible to eliminate **all** data races with languages like C/C++

Data Race Considerations

- Data races are **bugs**
- Impossible to eliminate **all** data races with languages like C/C++

- Existing tools can effectively detect frequently-manifested data races
- Deployed server applications go through testing / debugging

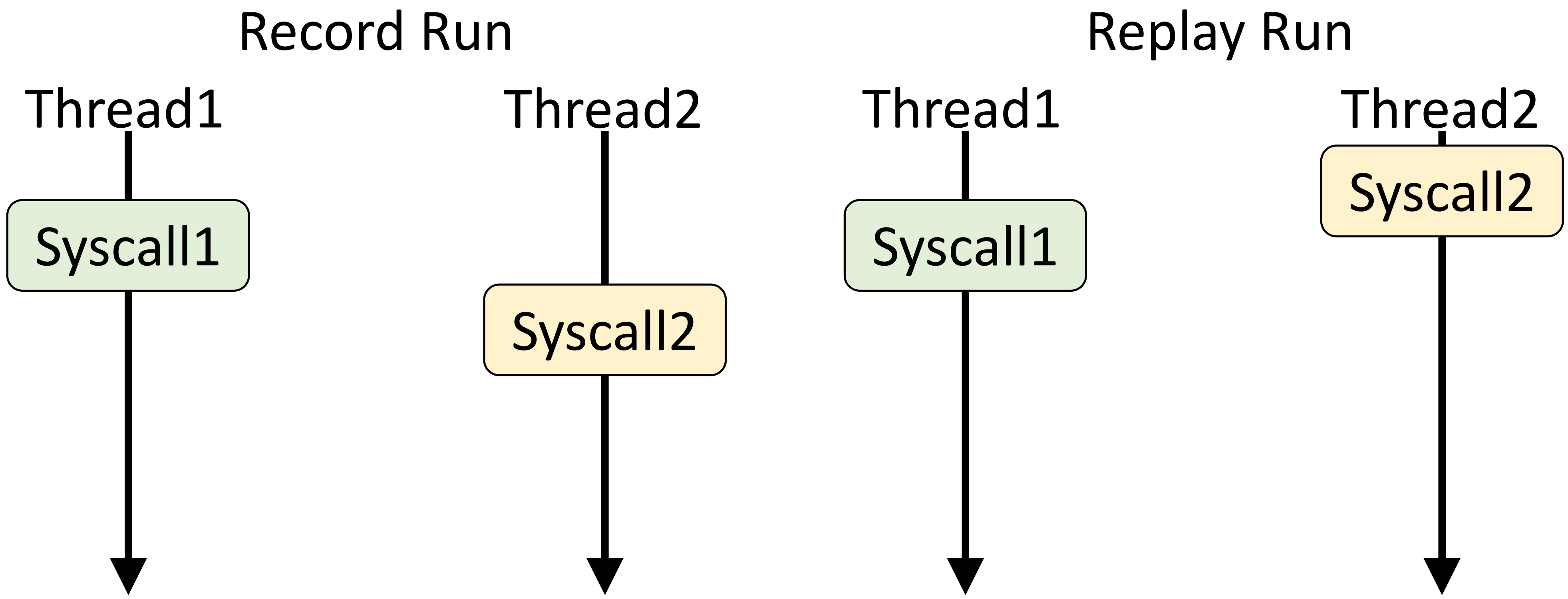
Data Race Considerations

- Data races are **bugs**
- Impossible to eliminate **all** data races with languages like C/C++

- Existing tools can effectively detect frequently-manifested data races
- Deployed server applications go through testing / debugging

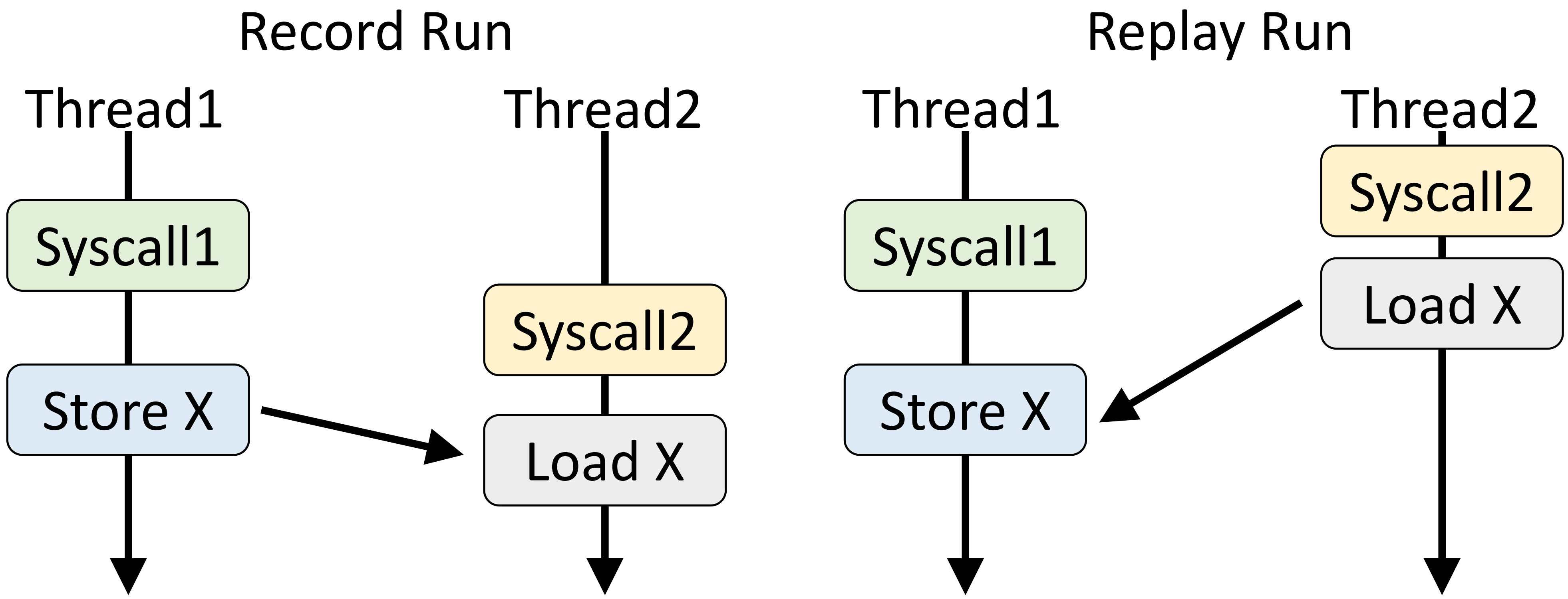
→ RRC focuses on **infrequently-manifested** data races

The Potential Impact of Data Races



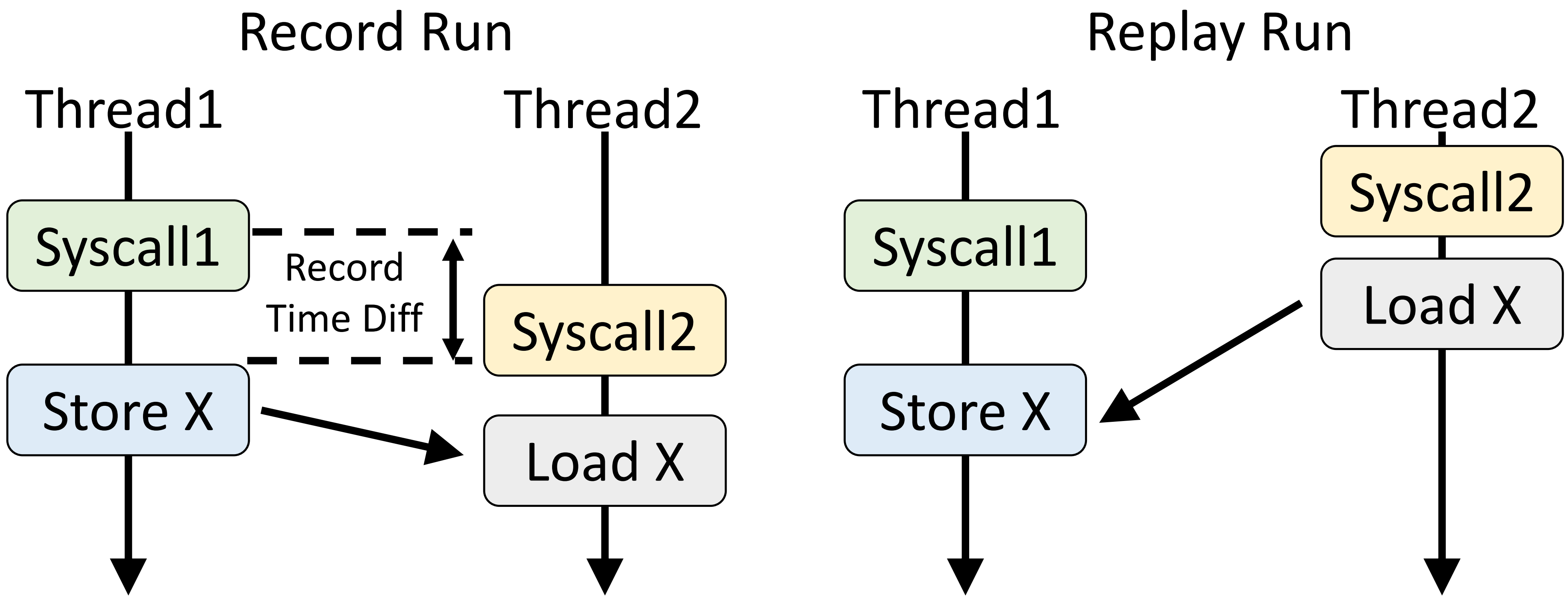
- During replay on the backup, most of system calls not actually executed
  → Significantly different timing of thread execution

The Potential Impact of Data Races



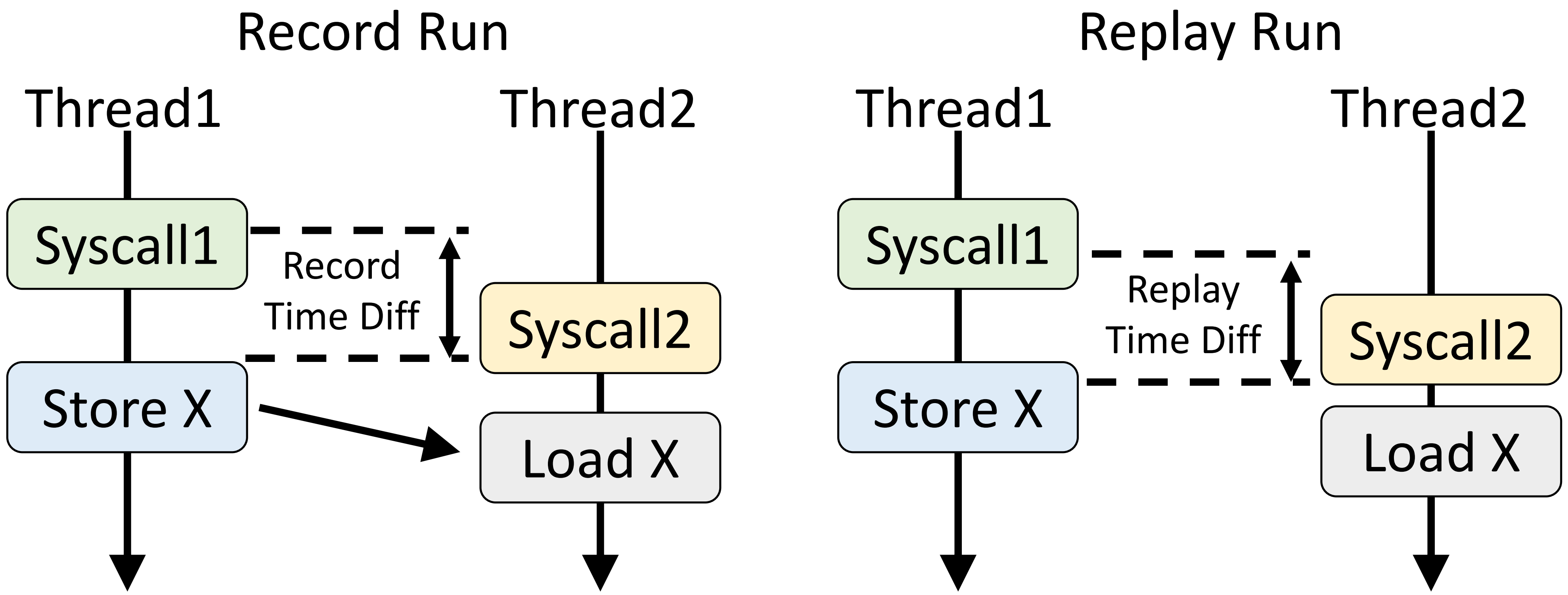
- During replay on the backup, most of system calls not actually executed
  → Significantly different timing of thread execution
    → Outcomes of data races
      → Different outcomes of replay

RRC’s Mitigation of the Impact of Data Races



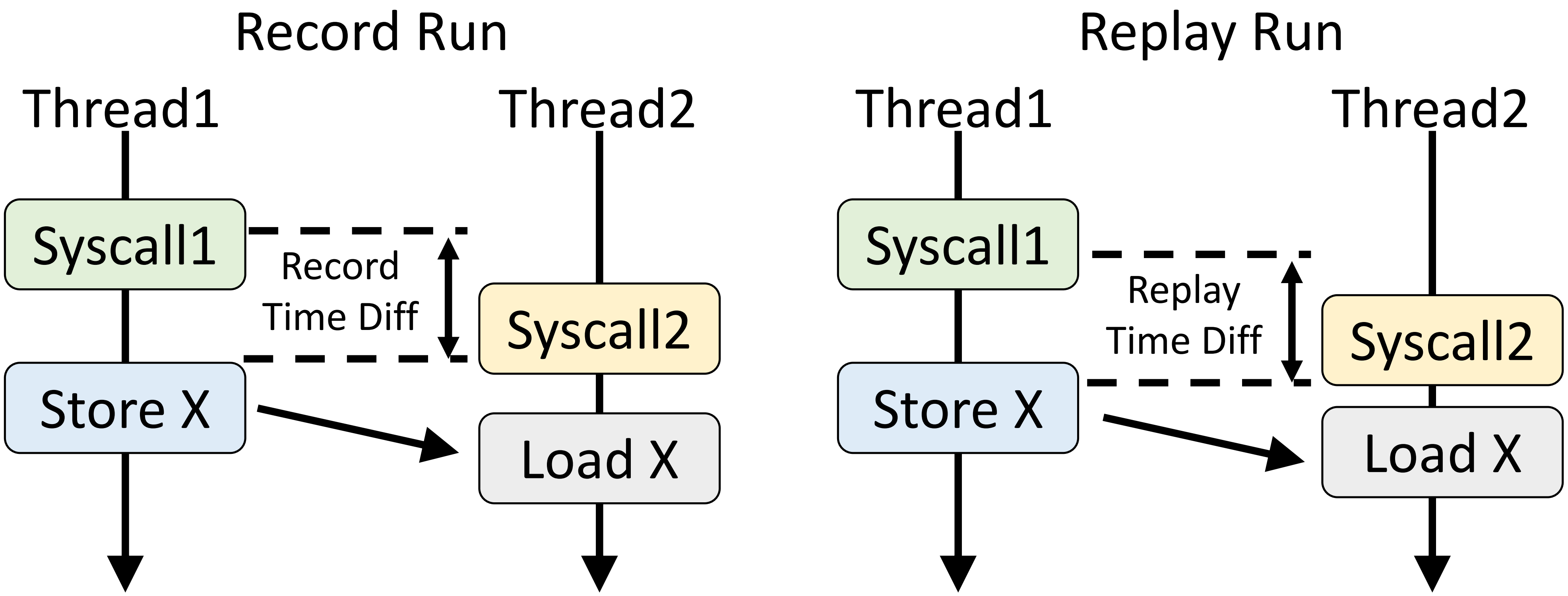
- Record time intervals between system call returns on the primary

RRC’s Mitigation of the Impact of Data Races



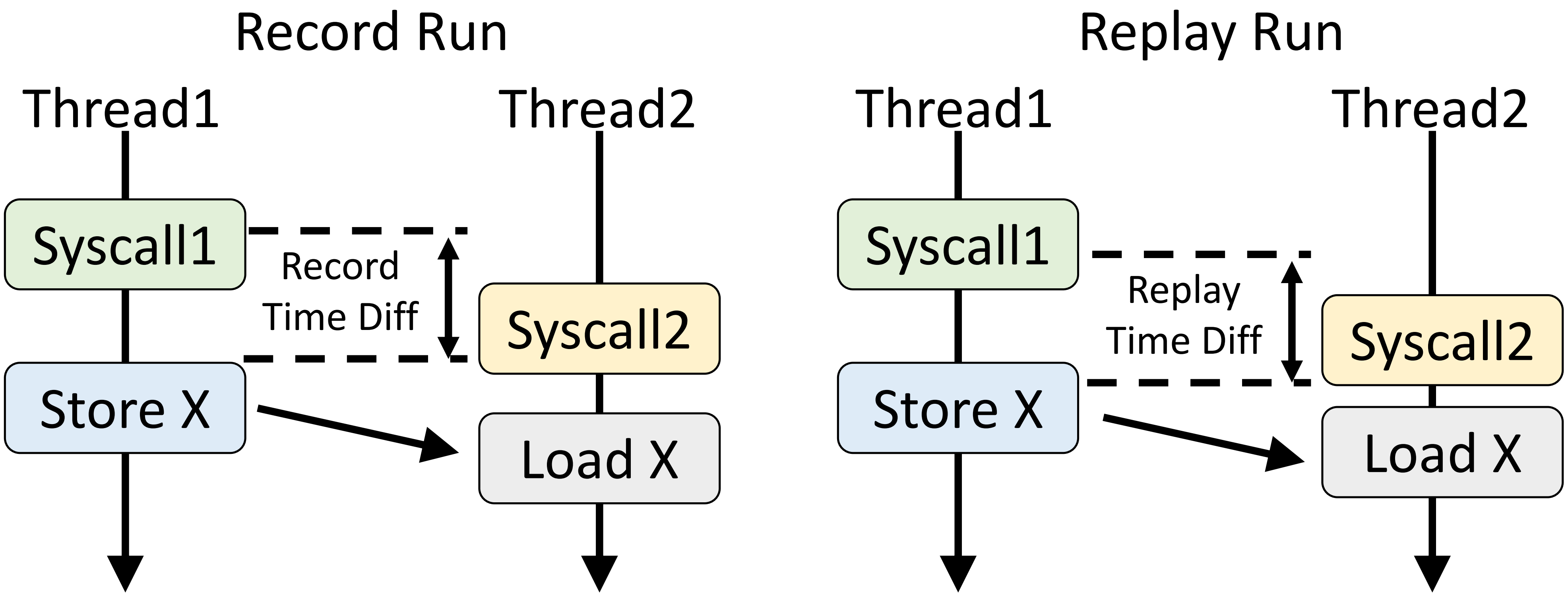
- Record time intervals between system call returns on the primary

RRC's Mitigation of the Impact of Data Races



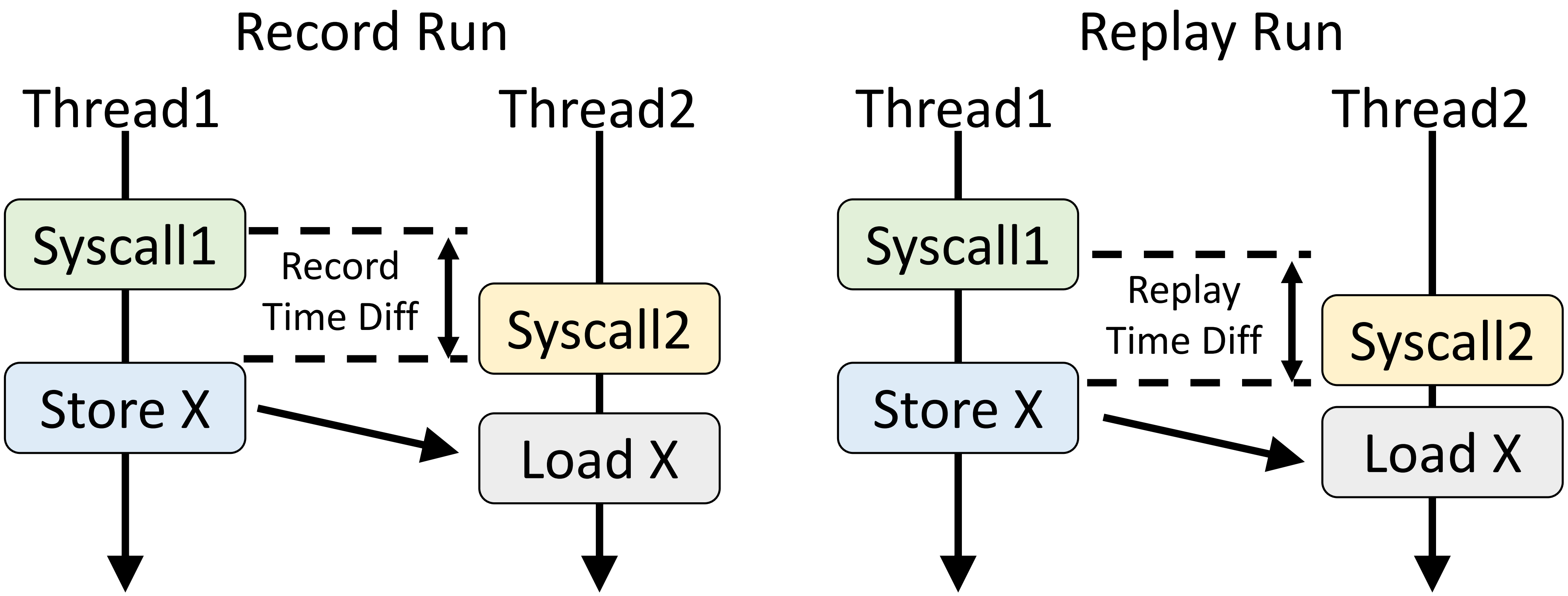
- Record time intervals between system call returns on the primary

RRC's Mitigation of the Impact of Data Races



- Record time intervals between system call returns on the primary
- Enforce inter-syscall interval during replay ≥ recorded interval

RRC's Mitigation of the Impact of Data Races



- Record time intervals between system call returns on the primary
- Enforce inter-syscall interval during replay ≥ recorded interval

Recovery success rate with infrequent data races: {35%, 51%} → > 99%

Talk Outline

- Preface
- Motivation
- RRC overview
- Overcoming design and implementation challenges
- **Evaluation**

Key design and implementation challenges

- Latency overhead
- Throughput overhead
- Recovery success rate
- Impact of data races
- CPU utilization overhead
- Pause time
- Recovery latency
- Impact of checkpoint interval
- Impact of workload footprint size and working set size
- Comparison with custom application-specific mechanisms

Key design and implementation challenges

- Latency overhead
- Throughput overhead
- Recovery success rate
- Impact of data races
- CPU utilization overhead
- Pause time
- Recovery latency
- Impact of checkpoint interval
- Impact of workload footprint size and working set size
- Comparison with custom application-specific mechanisms

Evaluation Setup

- Baseline:
  NiLiCon: Container replication, checkpointing to a passive backup

Evaluation Setup

- Baseline:
  NiLiCon: Container replication, checkpointing to a passive backup

- Workloads:
  - In-memory databases: Redis, Tarantool, SSDB, Memcached, Aerospike
  - Webserver: Lighttpd

Evaluation Setup

- Baseline:
  NiLiCon: Container replication, checkpointing to a passive backup

- Workloads:
  - In-memory databases: Redis, Tarantool, SSDB, Memcached, Aerospike
  - Webserver: Lighttpd

- RRC configuration:
  - 100ms checkpointing interval

Latency Overhead: RRC vs. NiLiCon

Average Latency Overhead

	Lig	Redis	Taran	SSDB	Mem$	Aero
RRC	144µs	198µs	211µs	263µs	169µs	290µs
NiLiCon	37ms	41ms	41ms	44ms	44ms	50ms

Latency Overhead: RRC vs. NiLiCon

Average Latency Overhead

	Lig	Redis	Taran	SSDB	Mem$	Aero
RRC	144µs	198µs	211µs	263µs	169µs	290µs
NiLiCon	37ms	41ms	41ms	44ms	44ms	50ms

Average: RRC: 144us – 290µs NiLiCon: 37ms – 50ms

Latency Overhead: RRC vs. NiLiCon

Average Latency Overhead

	Lig	Redis	Taran	SSDB	Mem$	Aero
RRC	144µs	198µs	211µs	263µs	169µs	290µs
NiLiCon	37ms	41ms	41ms	44ms	44ms	50ms

Average: RRC: 144us – 290µs NiLiCon: 37ms – 50ms

99th% : RRC: 235µs – 959µs NiLiCon: 39ms – 63ms

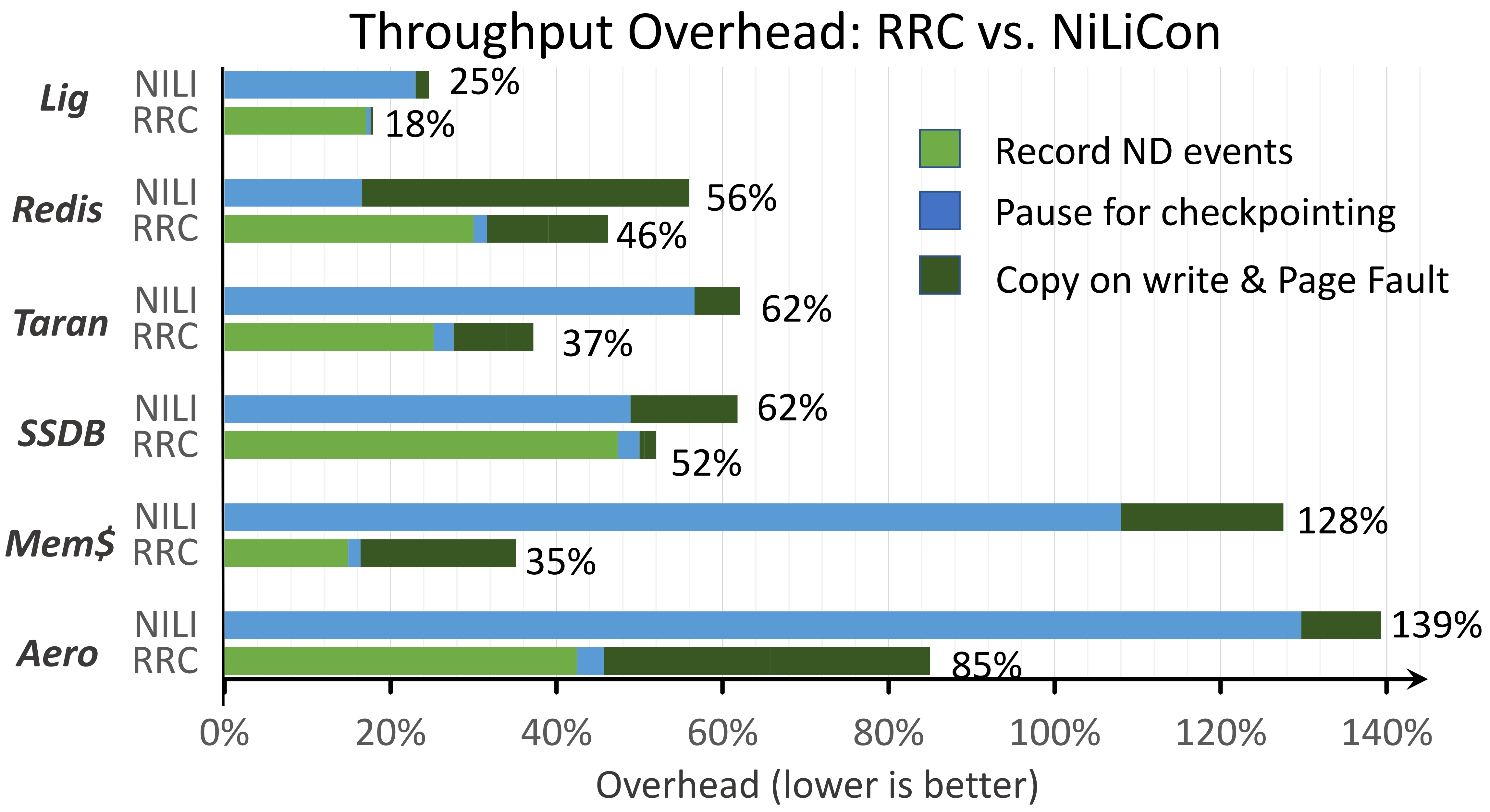
Latency Overhead: RRC vs. NiLiCon

Average Latency Overhead

	Lig	Redis	Taran	SSDB	Mem$	Aero
RRC	144µs	198µs	211µs	263µs	169µs	290µs
NiLiCon	37ms	41ms	41ms	44ms	44ms	50ms

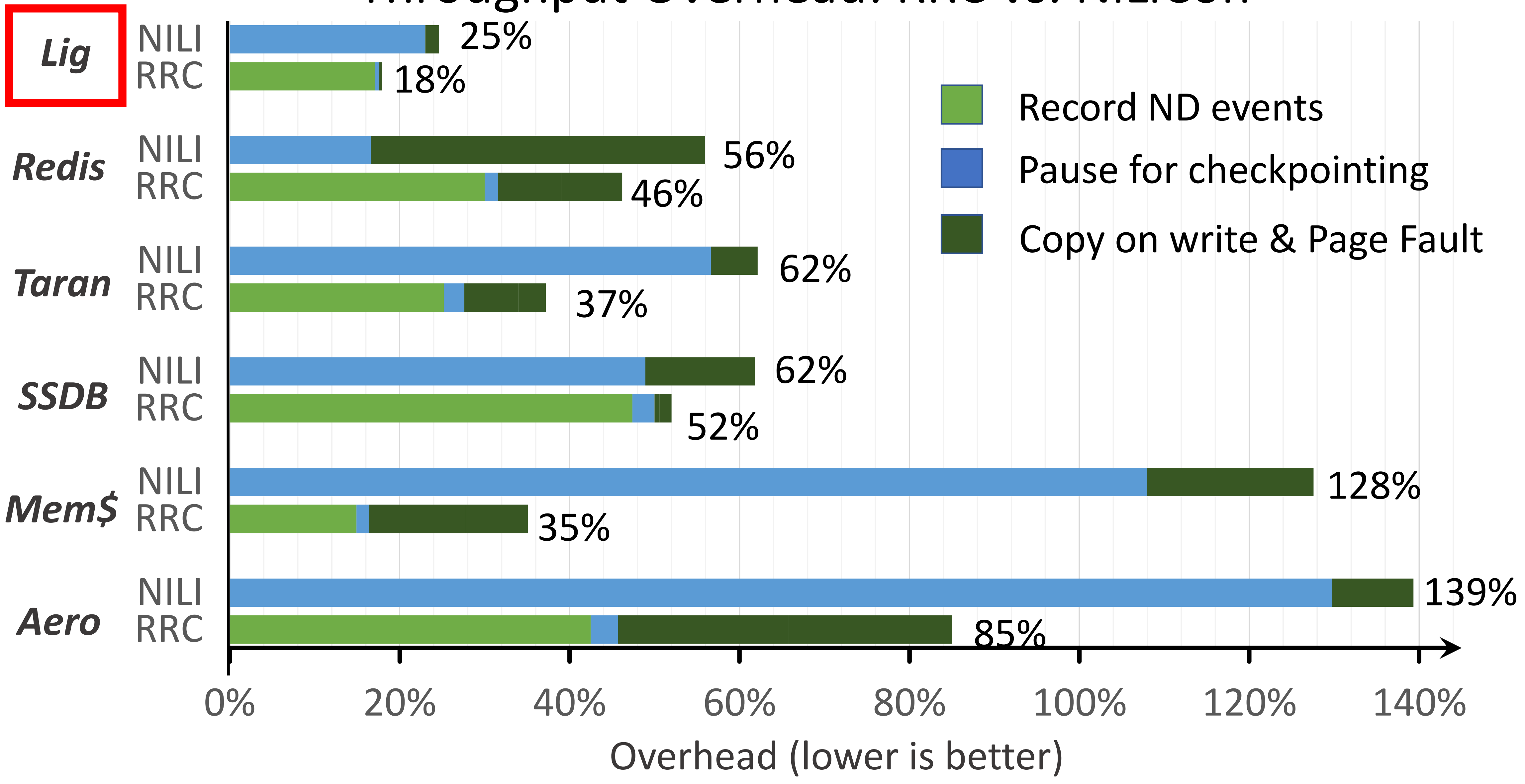
Average: RRC: 144us – 290µs NiLiCon: 37ms – 50ms
99th% : RRC: 235µs – 959µs NiLiCon: 39ms – 63ms

RRC: Hybrid replication + Container fork →
two orders of magnitude lower latency overhead

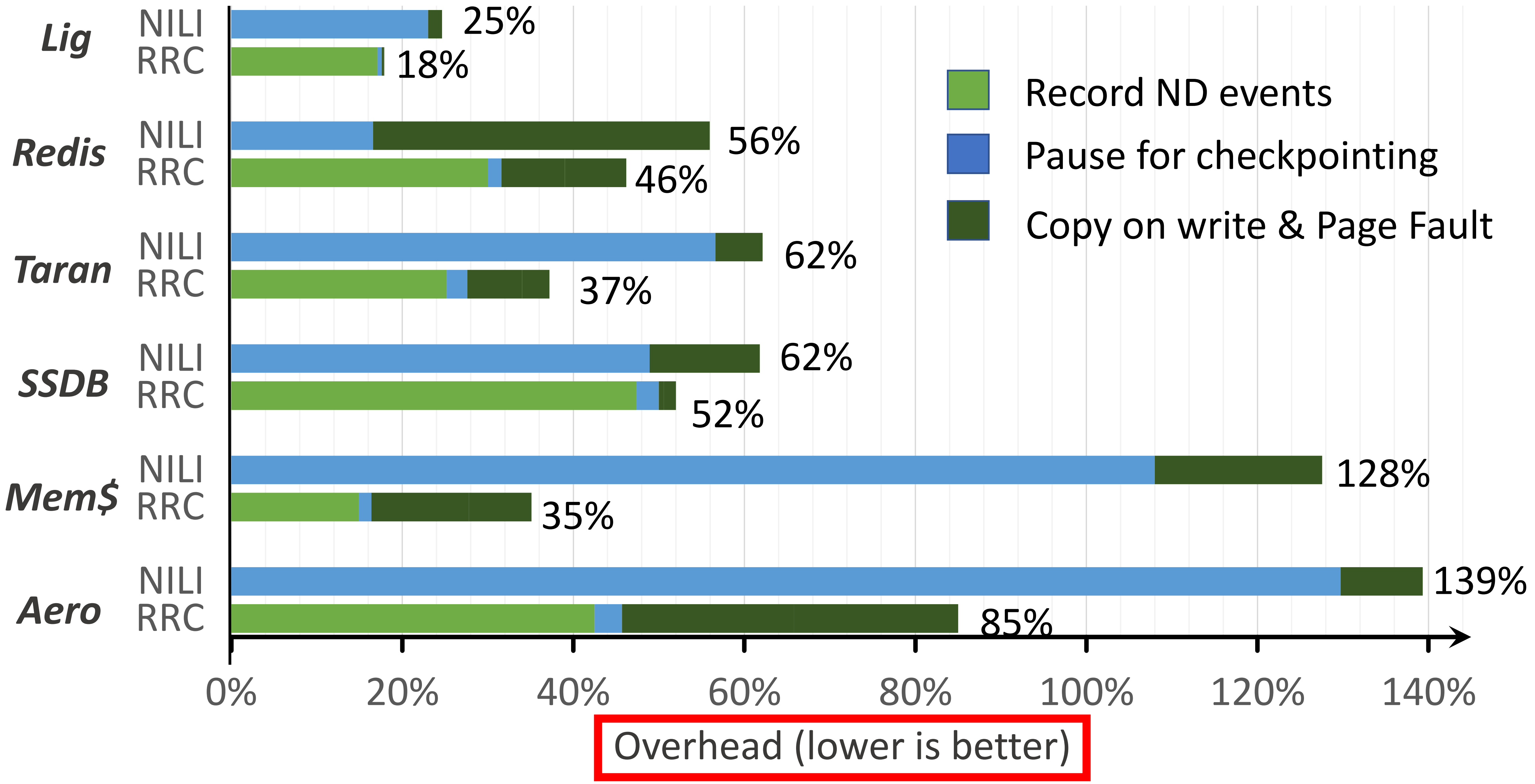


Throughput Overhead: RRC vs. NiLiCon
Lig
NILI
25%
RRC
18%
Redis
NILI
56%
RRC
46%
Taran
NILI
62%
RRC
37%
SSDB
NILI
62%
RRC
52%
Mem$
NILI
128%
RRC
35%
Aero
NILI
139%
RRC
85%
Record ND events
Pause for checkpointing
Copy on write & Page Fault
0%
20%
40%
60%
80%
100%
120%
140%
Overhead (lower is better)

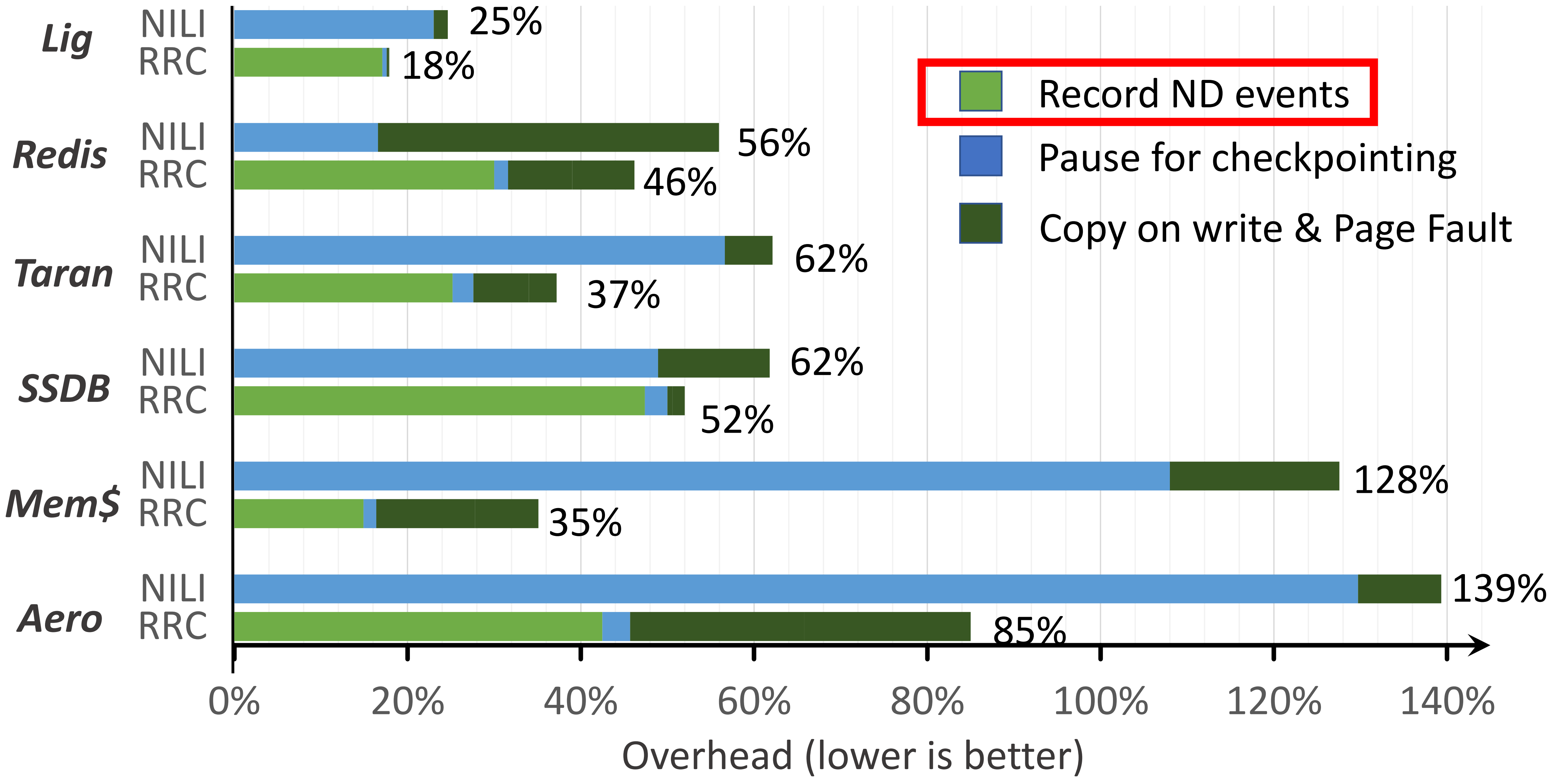
Throughput Overhead: RRC vs. NiLiCon

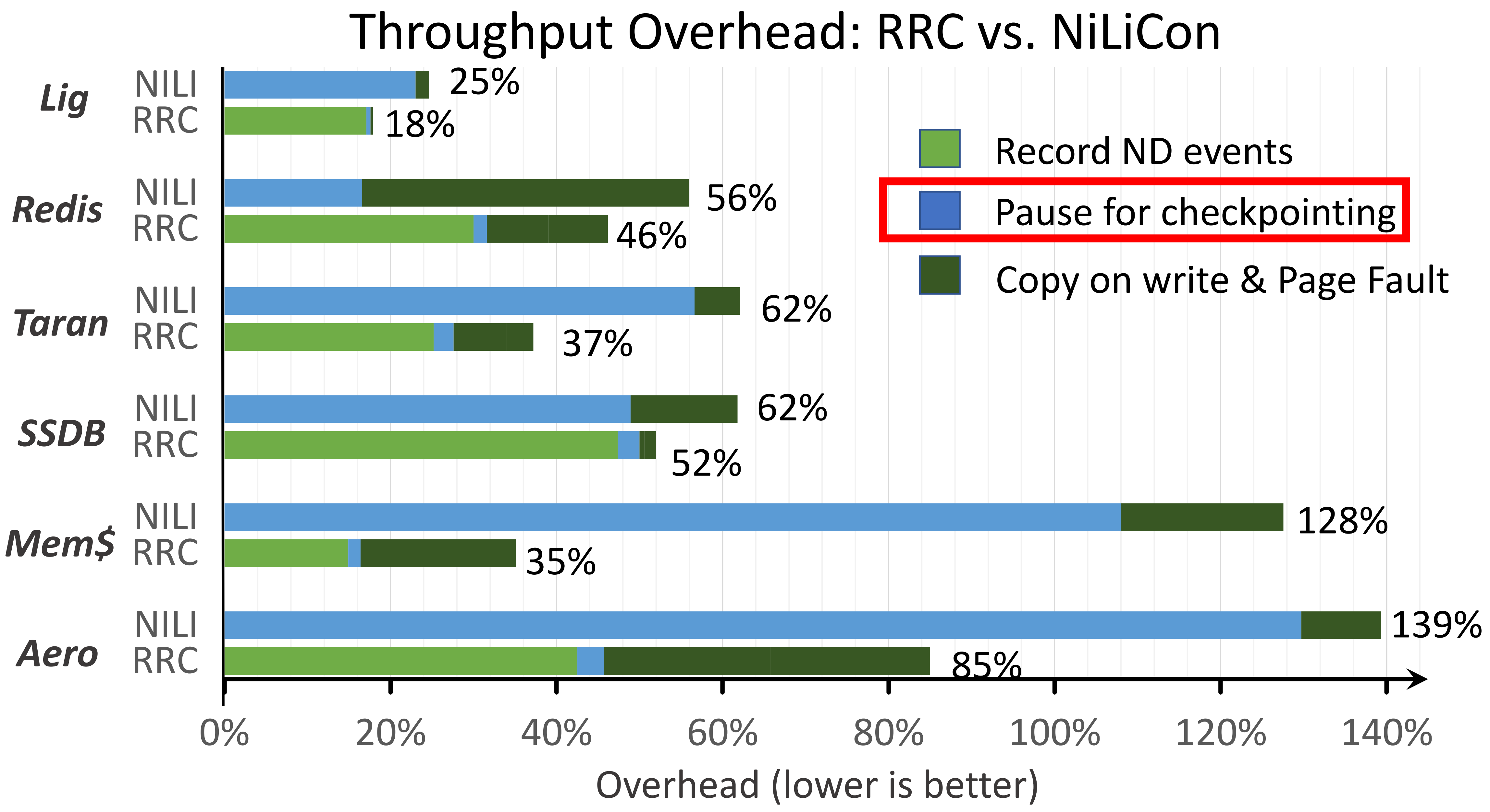


Throughput Overhead: RRC vs. NiLiCon



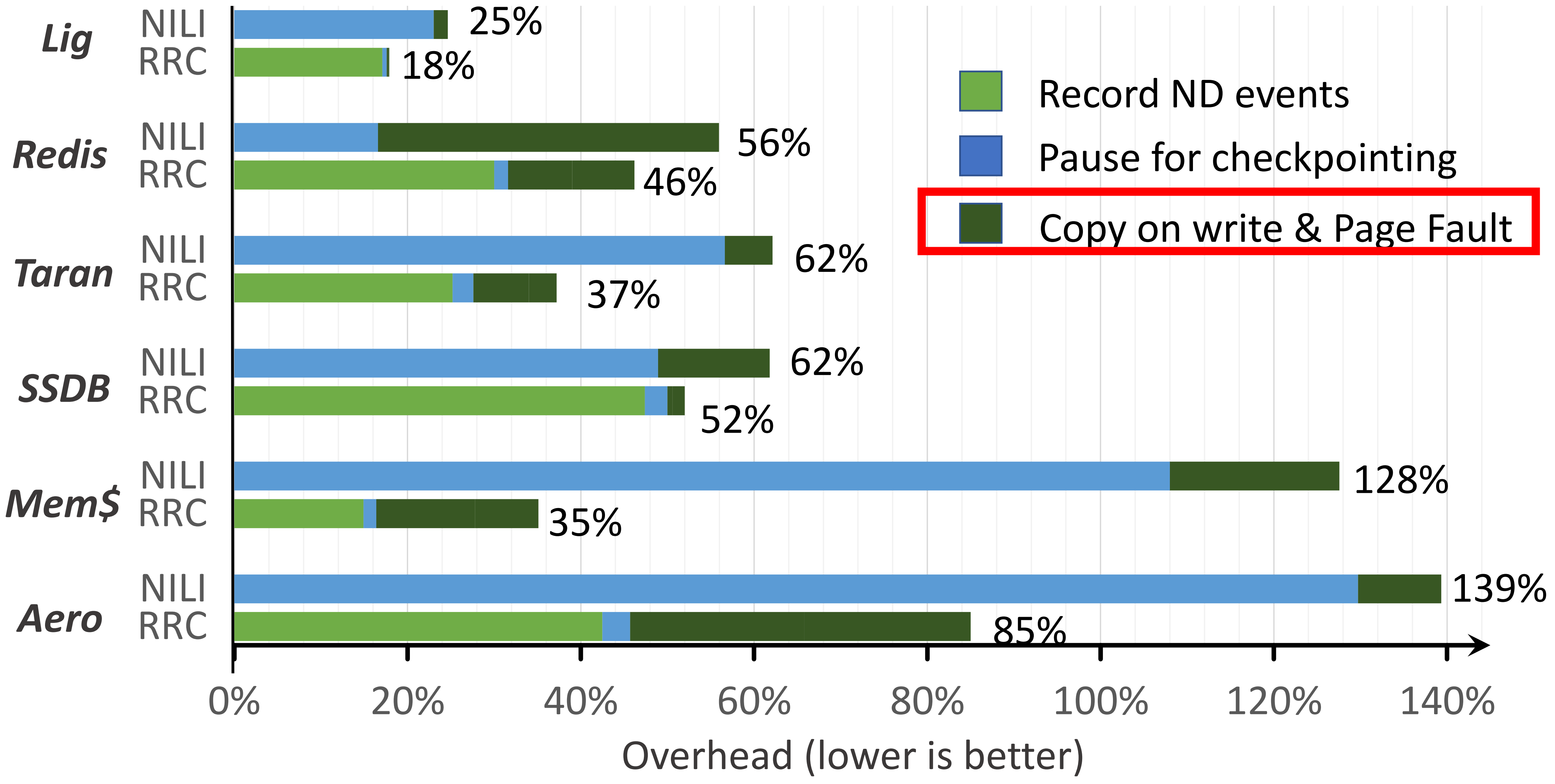
Throughput Overhead: RRC vs. NiLiCon



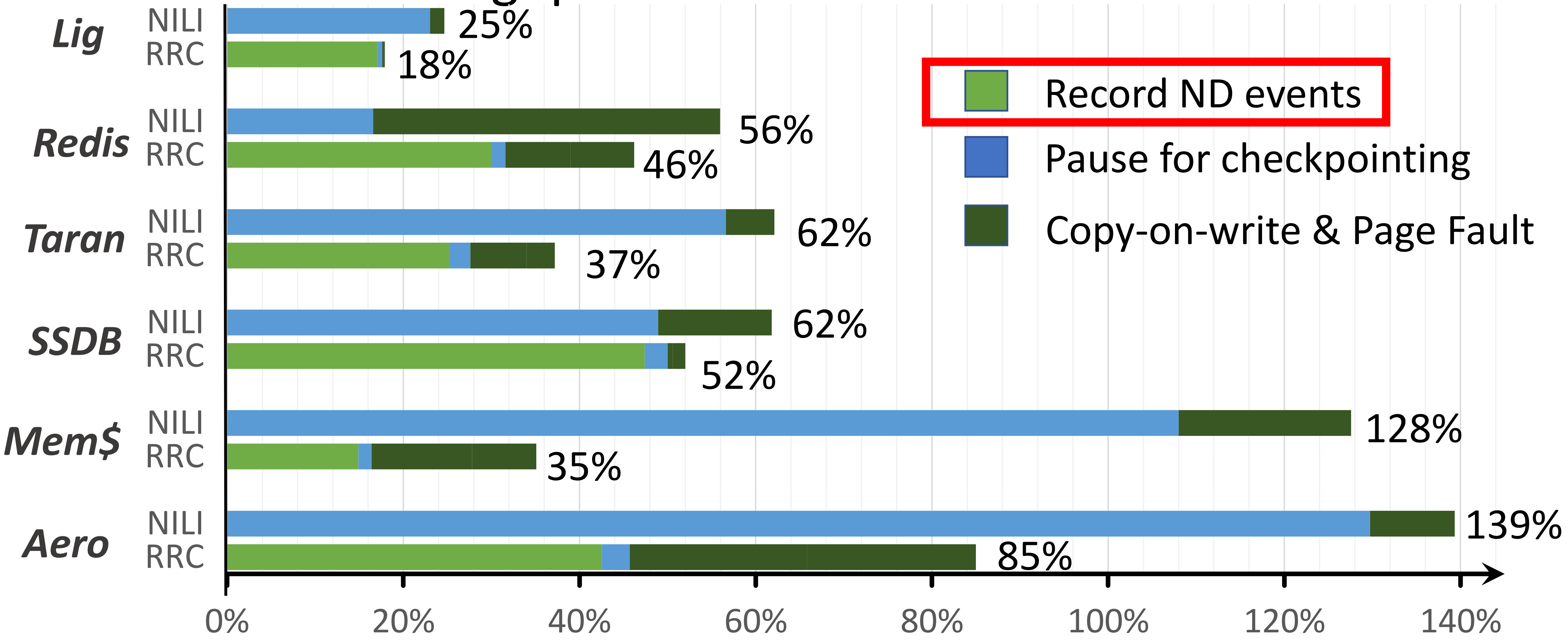


Throughput Overhead: RRC vs. NiLiCon
Lig
Redis
Taran
SSDB
Mem$
Aero
NILI
RRC
25%
18%
56%
46%
62%
37%
62%
52%
128%
35%
139%
85%
Record ND events
Pause for checkpointing
Copy on write & Page Fault
0%
20%
40%
60%
80%
100%
120%
140%
Overhead (lower is better)

Throughput Overhead: RRC vs. NiLiCon

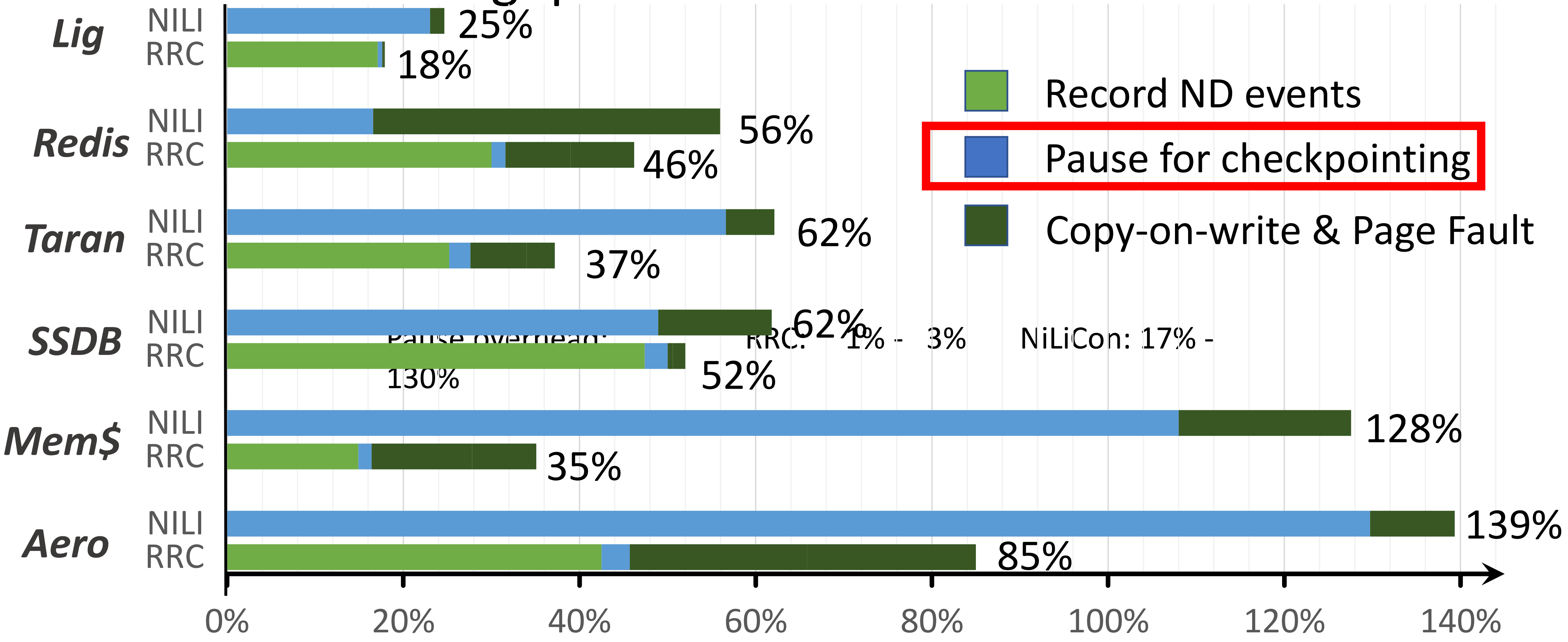


Throughput Overhead: RRC vs. NiLiCon



Recording overhead: RRC: 14% - 47% NiLiCon: 0%

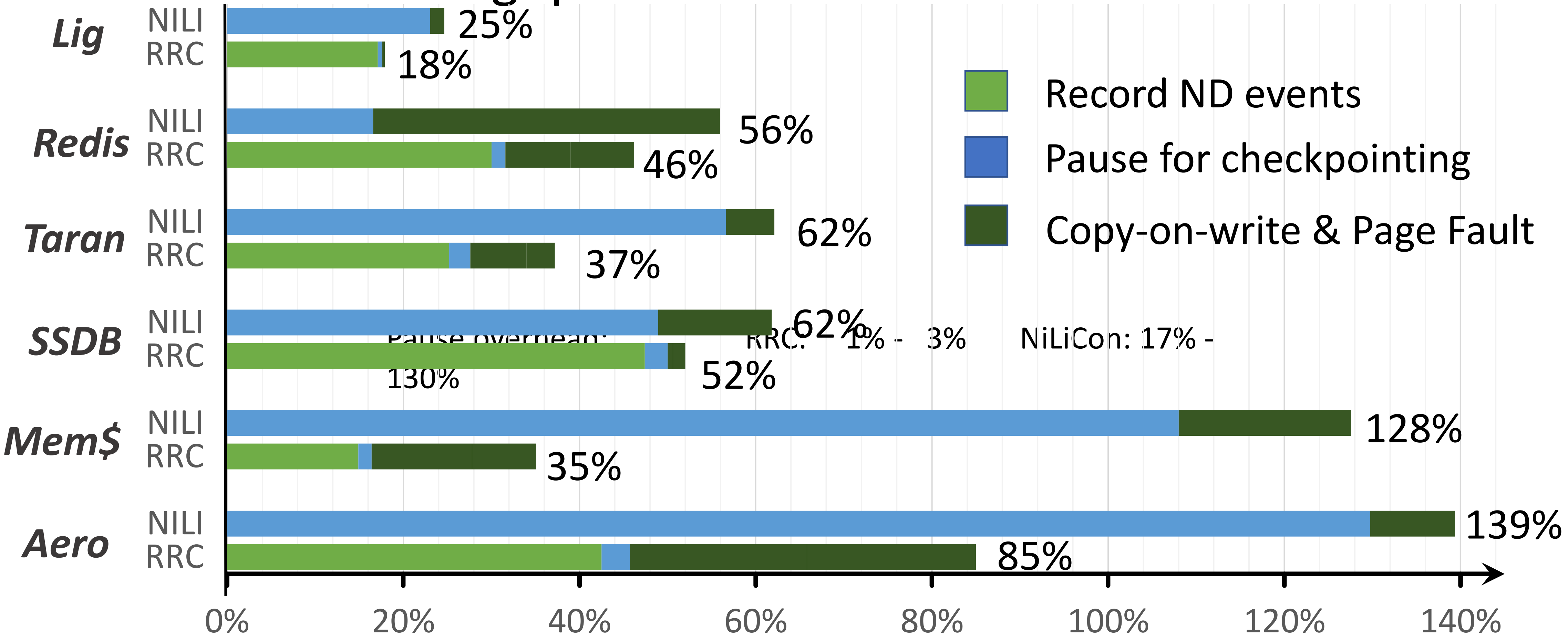
Throughput Overhead: RRC vs. NiLiCon



Recording overhead: RRC: 14% - 47% NiLiCon: 0%

Pause overhead: RRC: 1% - 3% NiLiCon: 17% - 130%

Throughput Overhead: RRC vs. NiLiCon



Recording overhead: RRC: 14% - 47% NiLiCon: 0%

Pause overhead: RRC: 1% - 3% NiLiCon: 17% - 130%

Overall: RRC: 18% - 85% NiLiCon: 25% - 139%

Recovery Success Rate

Fault injection setups:

- Fail-stop failures
- 1000s of fault injections
- Injection into both the primary and the backup host

Recovery Success Rate

Fault injection setups:

- Fail-stop failures
- 1000s of fault injections
- Injection into both the primary and the backup host

Recovery rate:

- >99% with real-world examples of data races
- **100%** without data races

Summary

- *Key goals:* Application-transparent fault tolerance for server applications
  - Multithreading
  - Minimize latency and throughput overhead

Summary

- *Key goals:* Application-transparent fault tolerance for server applications
  - Multithreading
  - Minimize latency and throughput overhead
- *Key insight:* **decouple** replication-related operations from normal operations
  - checkpoint interval ↔ delay in releasing outputs
  - time to take a checkpoint ↔ service interruption
  - Untracked nondeterminism ↔ service interruption

Summary

- *Key goals:* Application-transparent fault tolerance for server applications
  - Multithreading
  - Minimize latency and throughput overhead
- *Key insight:* **decouple** replication-related operations from normal operations
  - checkpoint interval ↔ delay in releasing outputs
  - time to take a checkpoint ↔ service interruption
  - Untracked nondeterminism ↔ service interruption
- *Key mechanisms:* hybrid replication: checkpointing + deterministic replay
  container fork
  passive backup
  mitigation of the impact of data races

Summary

- *Key goals:* Application-transparent fault tolerance for server applications
  - Multithreading
  - Minimize latency and throughput overhead
- *Key insight:* **decouple** replication-related operations from normal operations
  - checkpoint interval ↔ delay in releasing outputs
  - time to take a checkpoint ↔ service interruption
  - Untracked nondeterminism ↔ service interruption
- *Key mechanisms:* hybrid replication: checkpointing + deterministic replay
  container fork
  passive backup
  mitigation of the impact of data races
- *Key results:* average latency overhead < 290us vs. 10s of ms with passive backup
  throughput overhead < 85% vs. < 139% with passive backup
  recovery rate for fail-stop failures:
  - >99% with real-world examples of data races
  - 100% without data races

Support for Deterministic Replay

Requirement:

- Record nondeterministic events on the primary
- Transfer the log to the backup
- Replay the log for recovery on the backup

Nondeterministic events:

- External inputs – e.g., network packets from the clients
- Synchronization operations – e.g., lock acquire/release
- Certain local operations -- e.g., gettimeofday()