CS 591 K1:
Data Stream Processing and Analytics

Spring 2020

3/24: Exactly-once fault-tolerance in Apache Flink

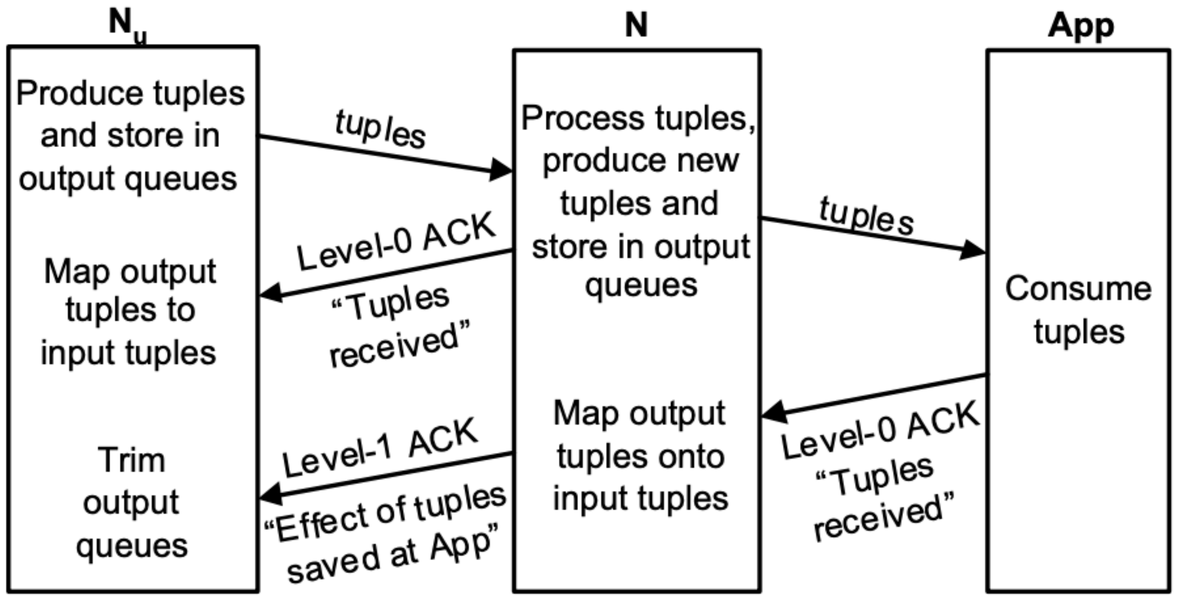
Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Some slides in this lecture have been generously provided by Paris Carbone, KTH <parisc@kth.se>

Go read his PhD thesis:
http://kth.diva-portal.org/smash/get/diva2:1240814/FULLTEXT01.pdf

Fault-tolerance approaches recap

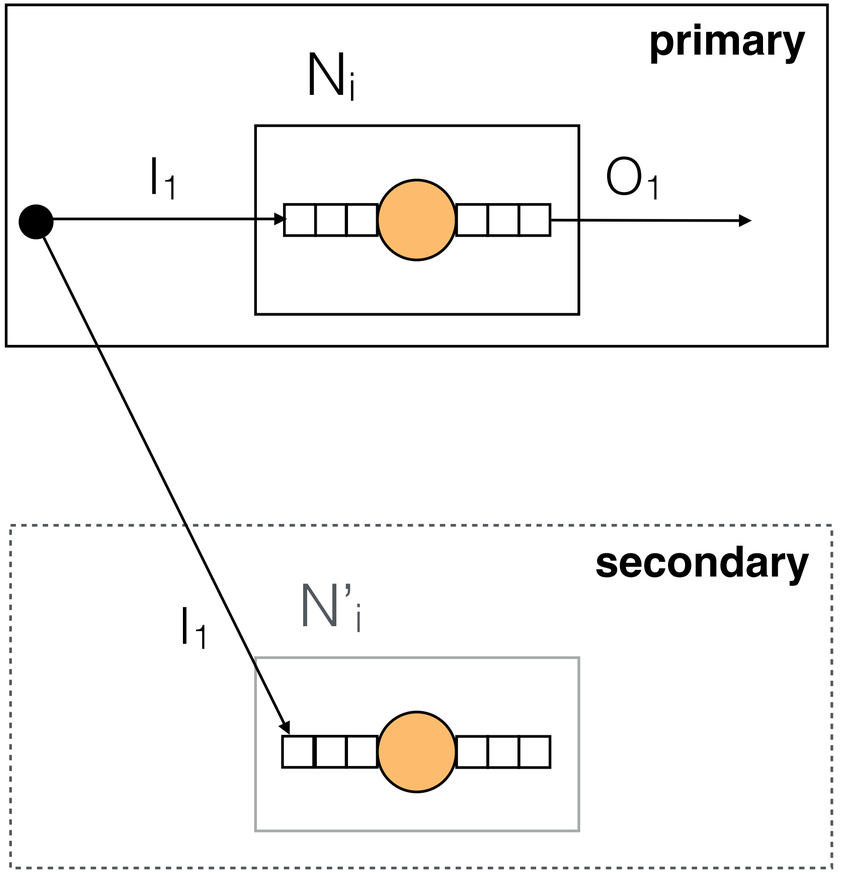
Upstream Backup



Upstream nodes act as backups for their downstream operators by logging tuples in their output queues until downstream operators have completely processed them.

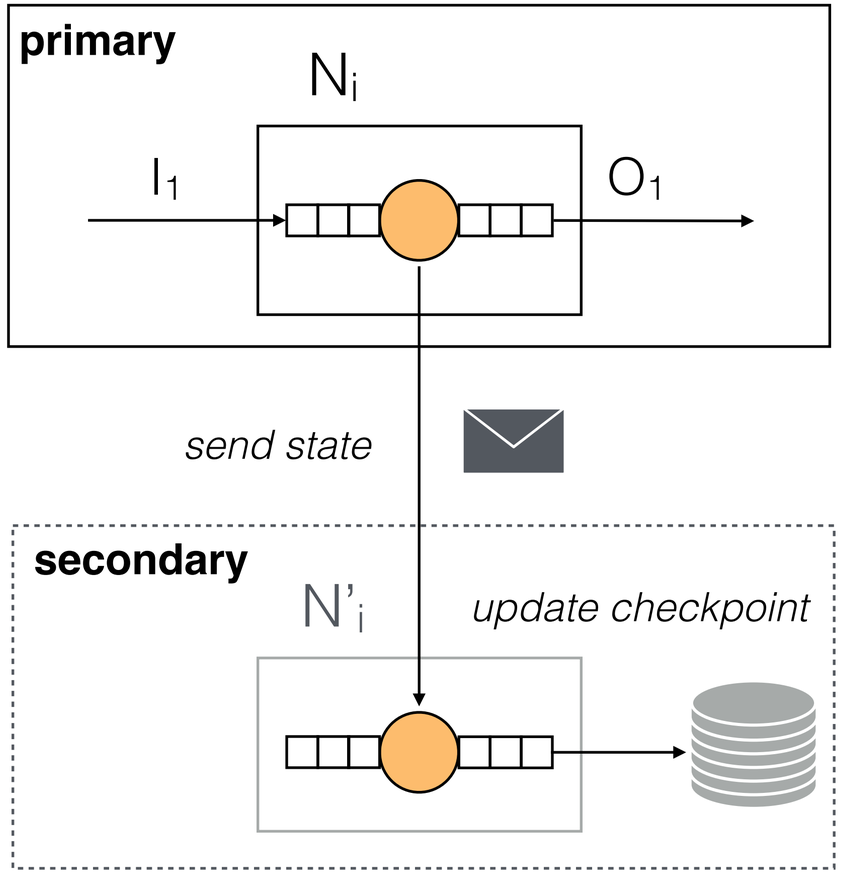
Active Standby

- The secondary receives tuples from upstream and processes them **in parallel with the primary**



Passive Standby

- Each primary periodically **checkpoints** its state and sends it to the secondary



How can we make sure that checkpoints are meaningful and coherent?

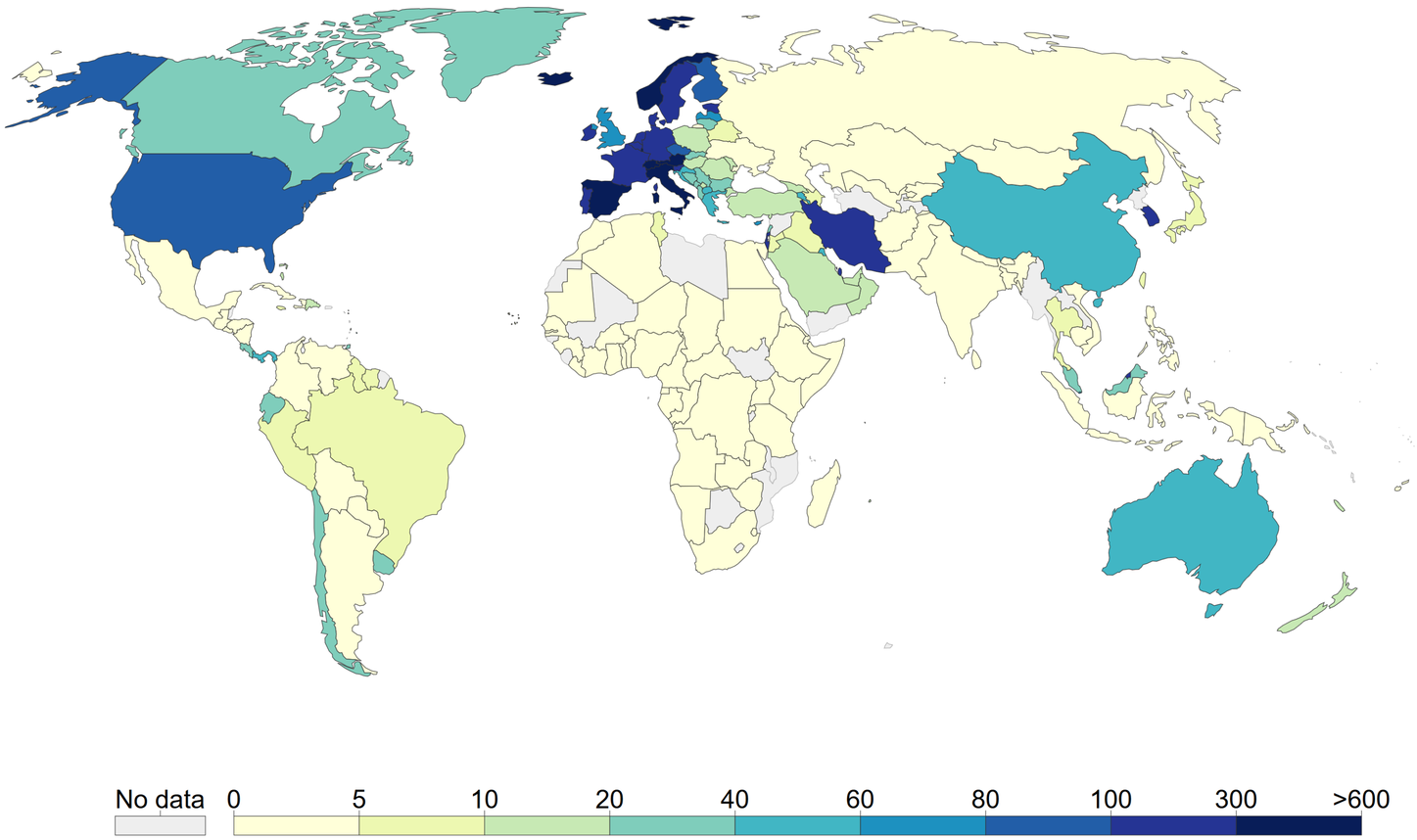
Global snapshots

Image credit: NASA

Total confirmed cases of COVID-19 per million people, Mar 22, 2020

The number of confirmed cases is lower than the number of total cases. The main reason for this is limited testing.

Our World in Data



Source: European CDC – Latest Situation Update Worldwide

OurWorldInData.org/coronavirus • CC BY

Note: The large number of cases globally and in China on Feb 17 is the result of a change in reporting methodology.

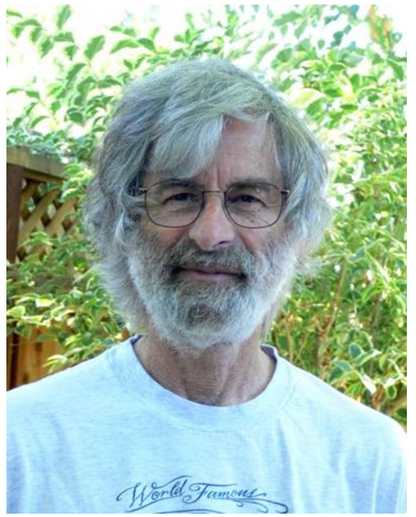

114 S Main St
Burleson, Texas
Google
Street View
©2018 Google
Google
Image capture: Mar 2008 © 2020 Google United States Terms Report a problem

Freeze the world

Naive algorithm

1. Pause the ingestion of all input streams
2. Wait for all in-flight data to be completely processed
3. Copy the state of each task to a remote, persistent storage
4. Wait until all tasks have finished their copies
5. Resume processing and stream ingestion

The distributed snapshot algorithm described here came about when I visited Chandy, who was then at the University of Texas in Austin. He posed the problem to me over dinner, but we had both had too much wine to think about it right then. The next morning, in the shower, I came up with the solution. When I arrived at Chandy's office, he was waiting for me with the same solution.



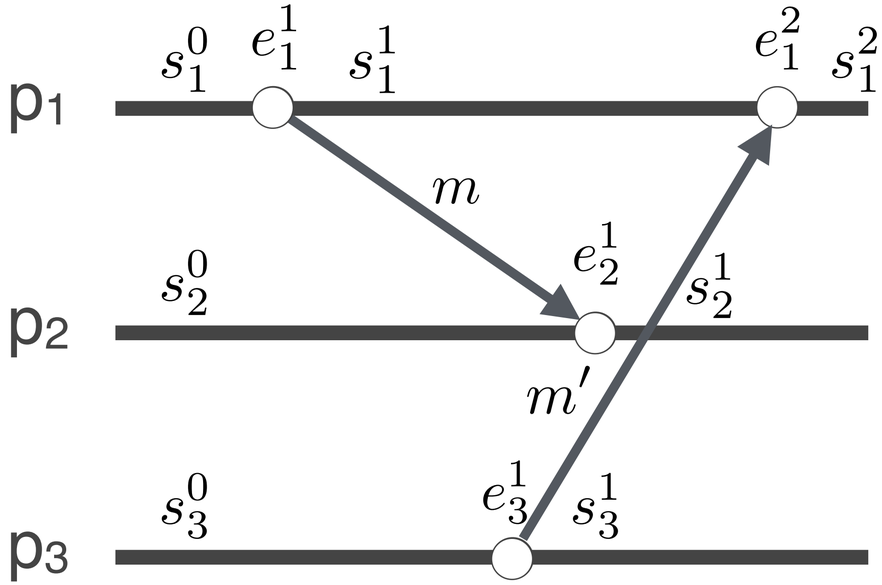
–Leslie Lamport

http://lamport.azurewebsites.net/pubs/pubs.html#chandy

Snapshotting Protocols

A snapshot algorithm attempts to capture a **coherent global state** of a distributed system

We need to retrieve a **distributed cut** in a system execution that yields a system configuration



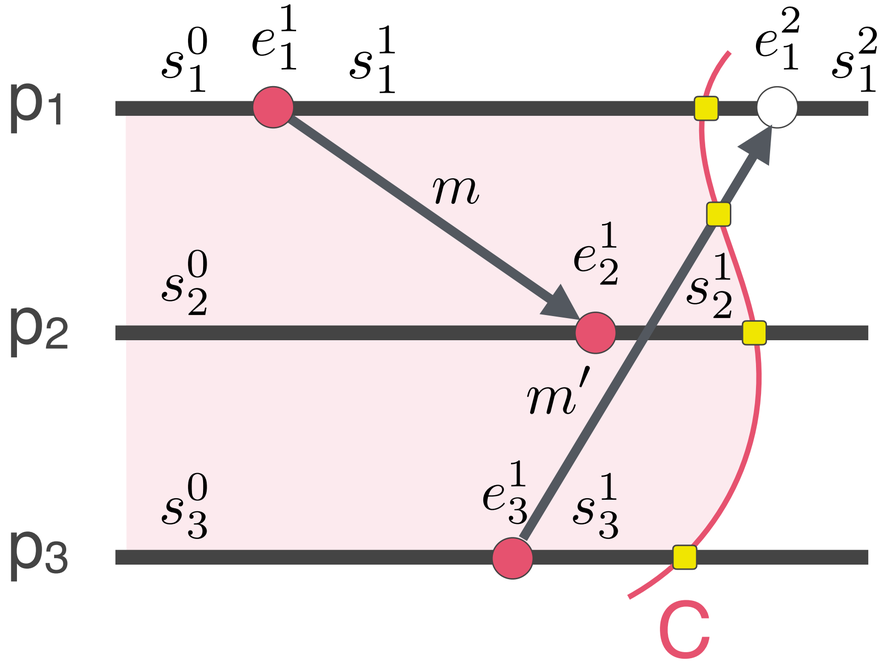
Validity (safety): Obtain a **valid** system configuration

Termination (liveness): A **full** system configuration is eventually captured

Snapshotting Protocols

A snapshot algorithm attempts to capture a **coherent global state** of a distributed system

We need to retrieve a **distributed cut** in a system execution that yields a system configuration



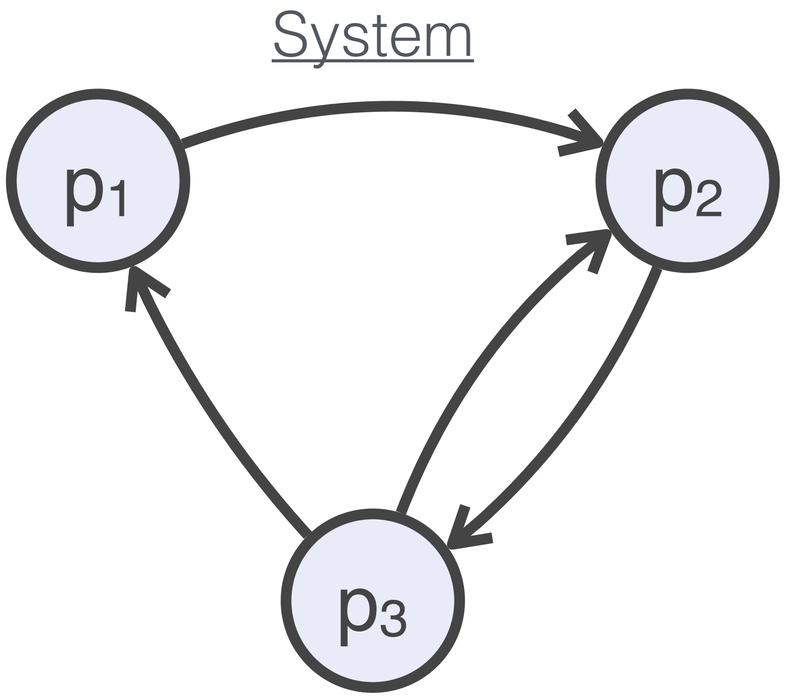
Snapshot of C

$\{s_1^1, s_2^1, s_3^1\}$
$\{m'\}$

Validity (safety): Obtain a **valid** system configuration

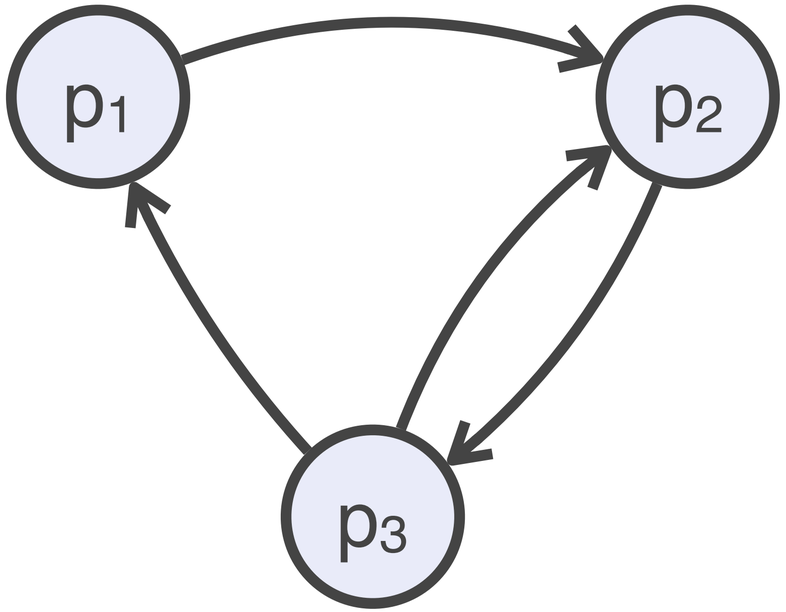
Termination (liveness): A **full** system configuration is eventually captured

Validity Explained

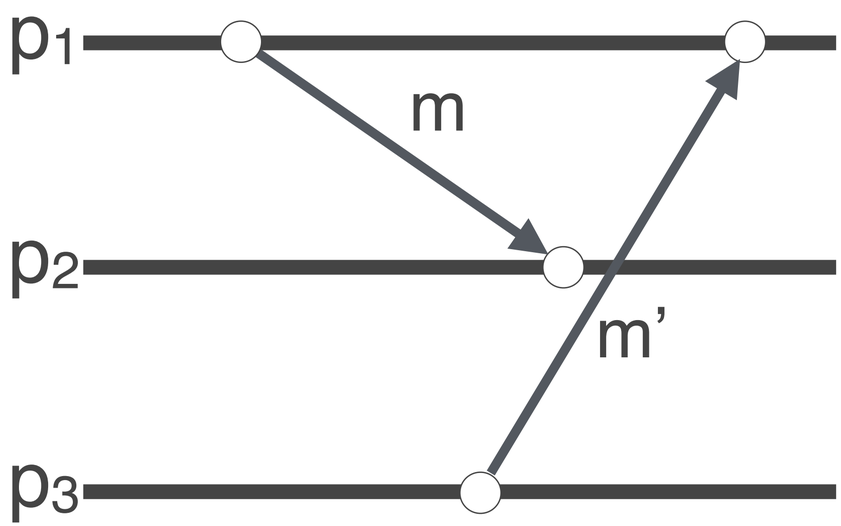


Validity Explained

System

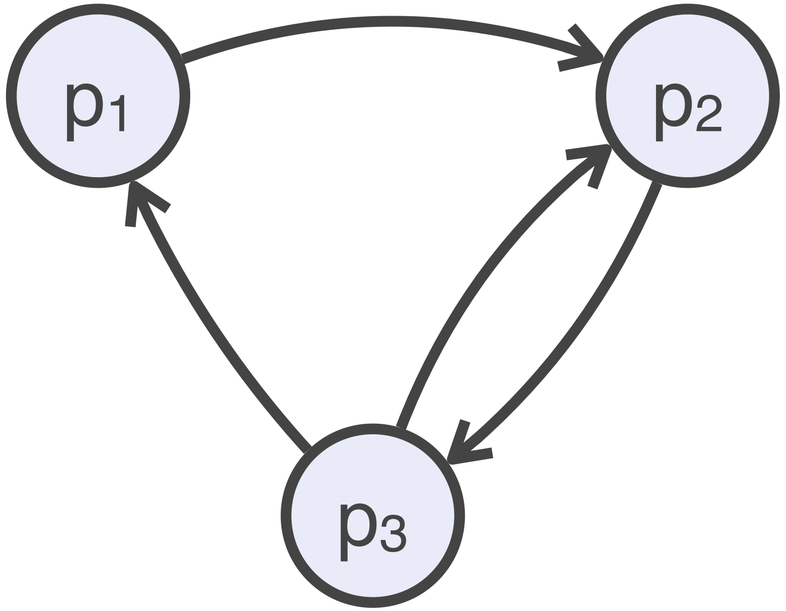


Possible Execution

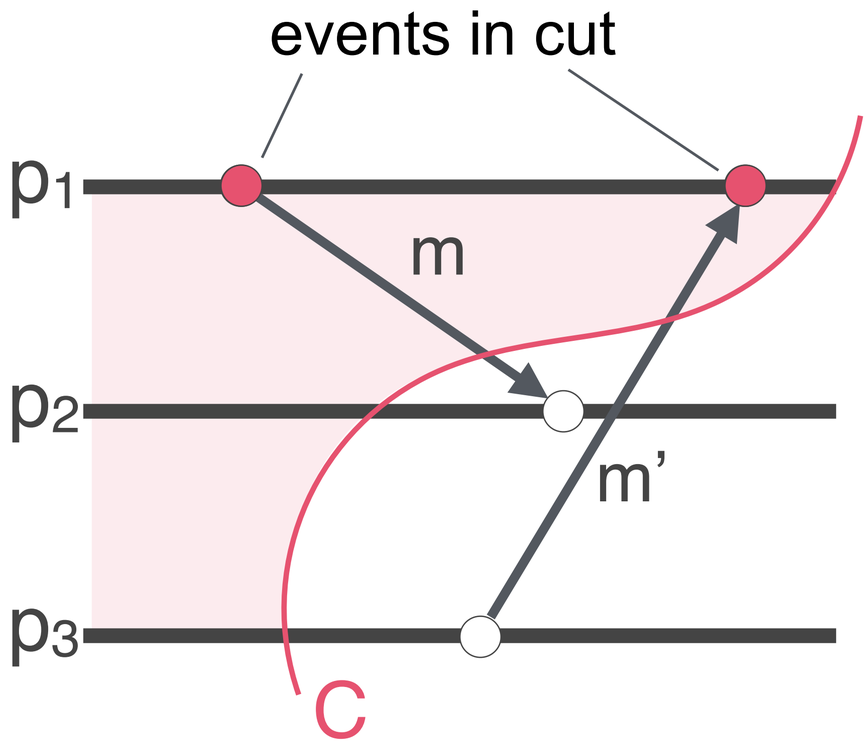


Validity Explained

System

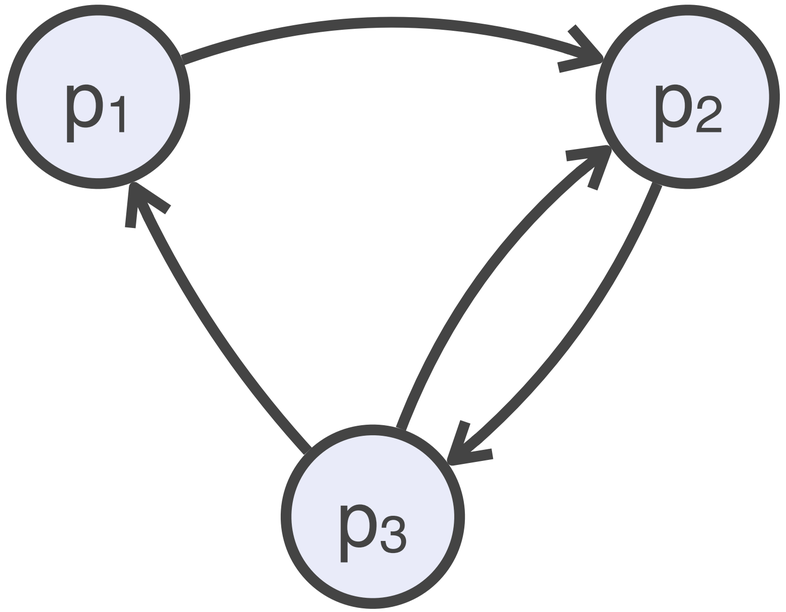


Possible Execution

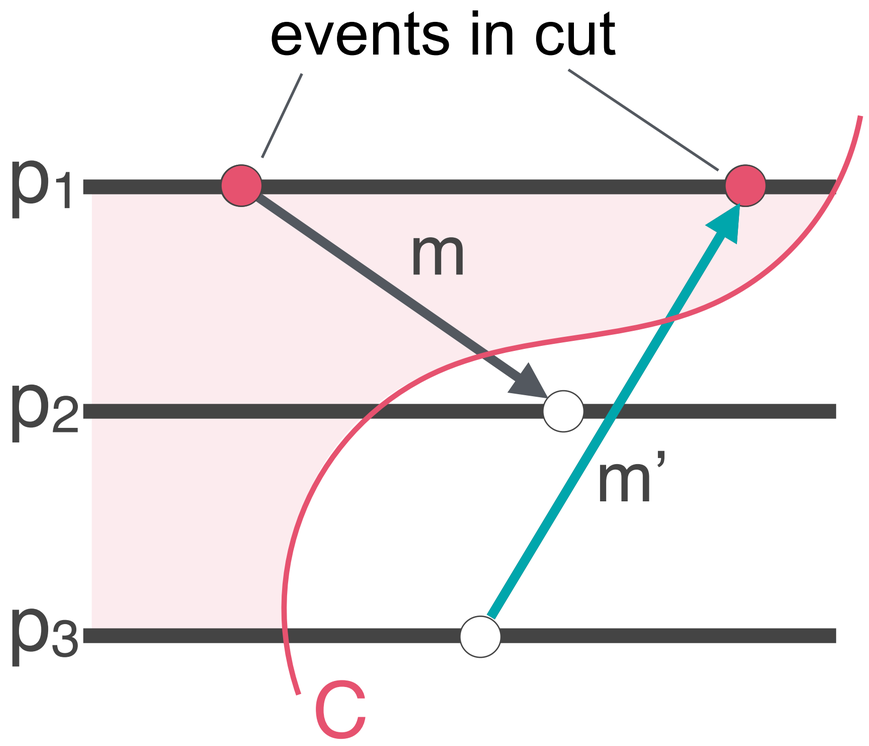


Validity Explained

System



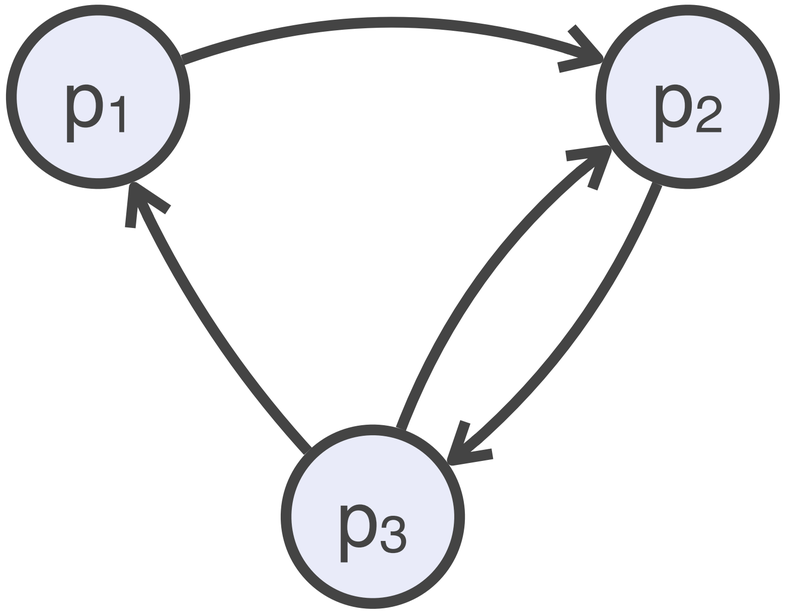
Possible Execution



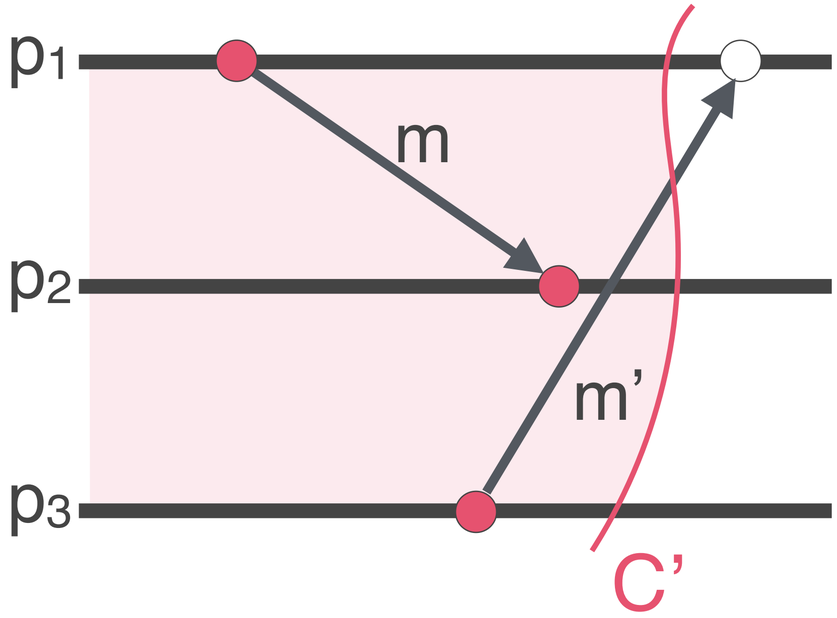
Not Valid: According to C, **m'** was received but never sent
(Violates Causality)

Validity Explained

System



Possible Execution

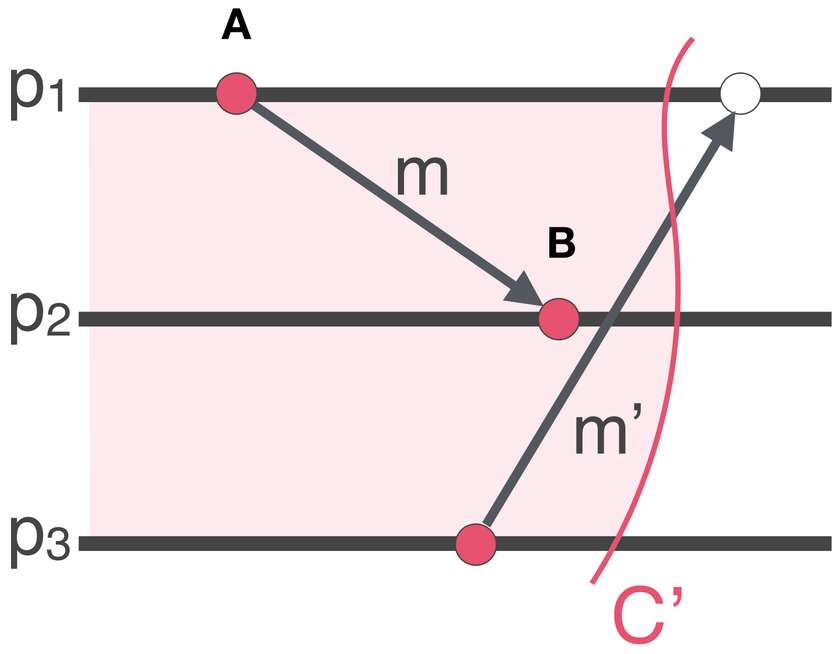


Valid: No causality violations in C' (**C' is a consistent cut)**

Causal consistency

A consistent cut satisfies **causality**:

- An event is **pre-snapshot** if it occurs *before* the local snapshot on a process, otherwise it is **post-snapshot**
- If event *A happens causally before B* and B is pre-snapshot, then A is also pre-snapshot



The Chandy-Lamport Algorithm

A snapshot algorithm that is used in distributed systems for recording a **consistent global state** of an **asynchronous** system

System model:

- No failures during snapshotting
- FIFO reliable channels: no lost or duplicate messages
- Strongly connected execution graph: each process can reach every other process in the system
- Single initiating process

The Chandy-Lamport Algorithm

Requirements:

- Taking a snapshot does not interfere with processing
  - processing and messages do not stop
- Each process cast locally record its own state
- Any process can initiate the algorithm

Initiating a snapshot

The initiator process:

1. Records its own state.
2. Sends a *marker* out on each of its outgoing channels.
    a. The marker is a special message that is not recorded in the snapshot but enforces the causal consistency.
3. Starts recording all data (application) messages it receives on all of its incoming channels.

On receiving a marker (I)

A process receiving a marker for the **first time**:

1. Records its own state.
2. Marks the channel that the marker came in on as *empty*.
    a. Future messages arriving on this channel will no be part of the snapshot.
3. Sends markers to all its outgoing channels.
4. Starts recording incoming messages on all its incoming channels *except* the one marked as empty.

Otherwise:

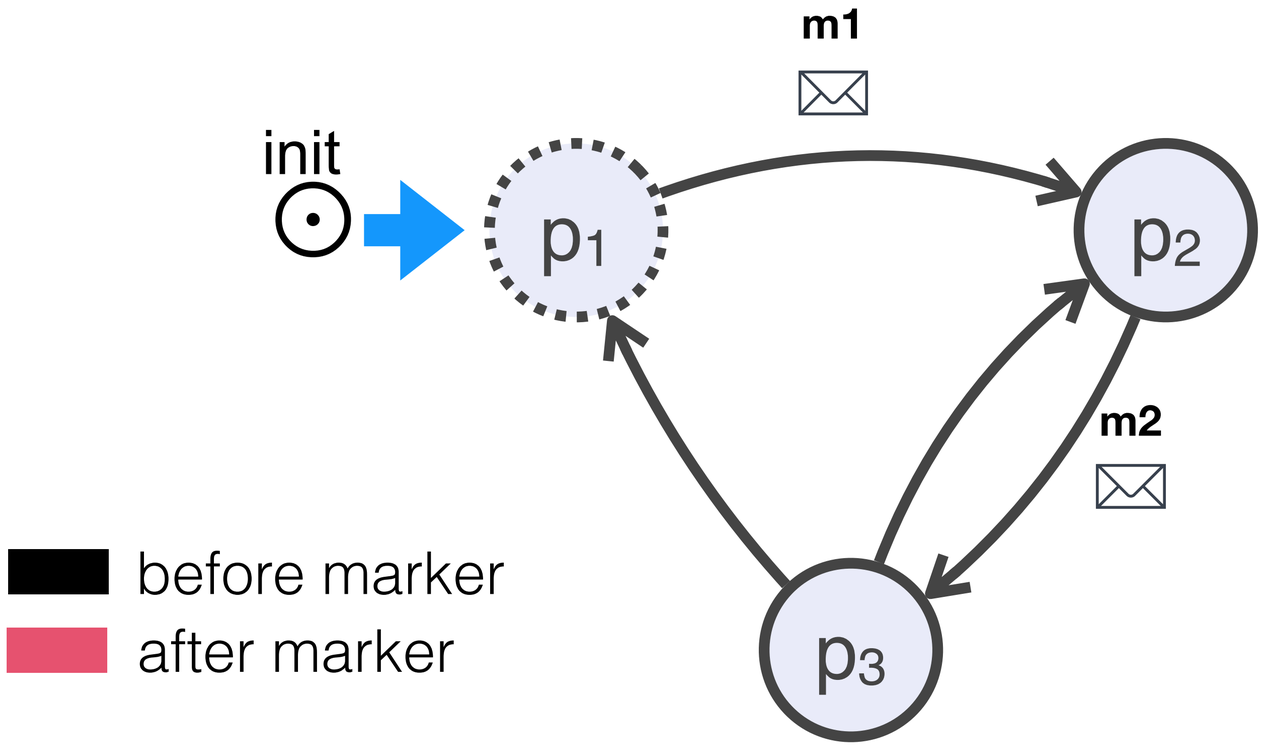
1. It stops recording on the channel it received the marker from.

Completing a snapshot

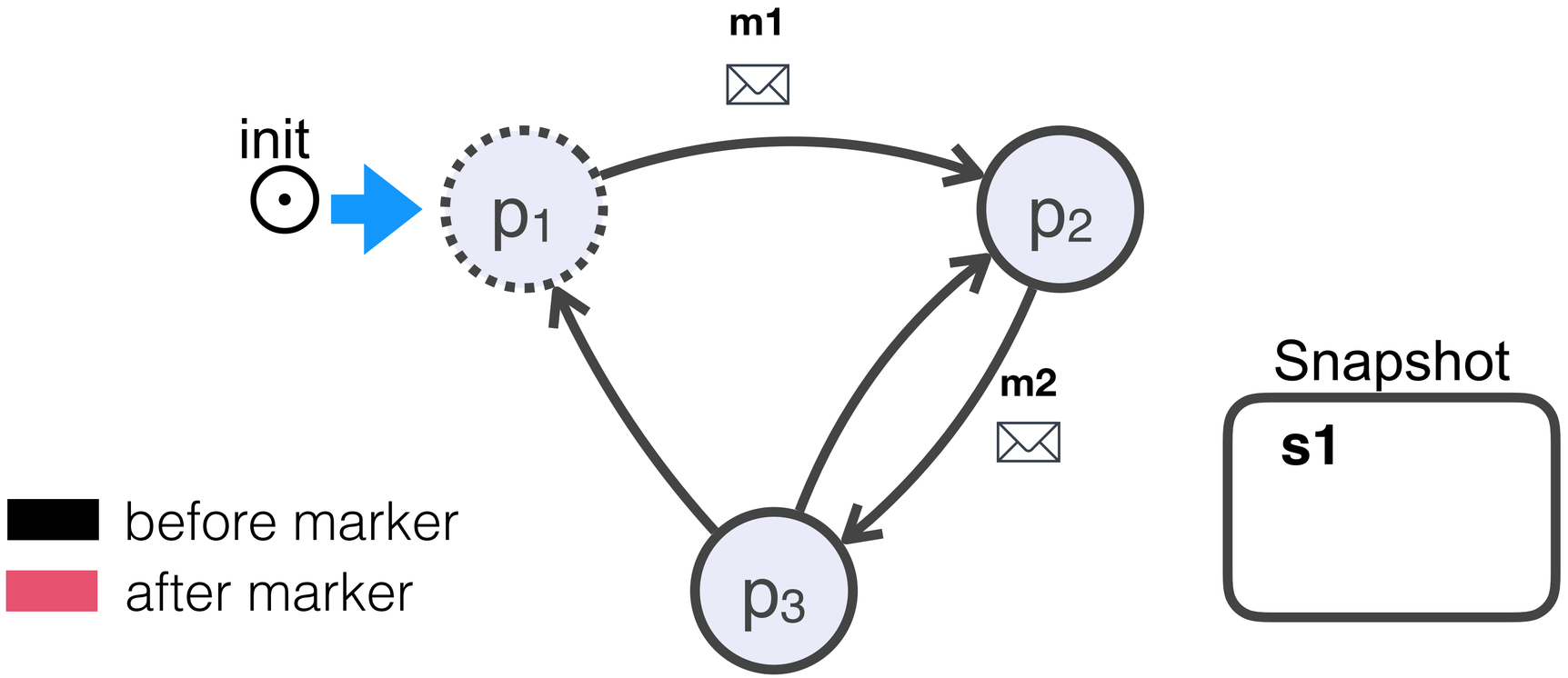
When all processes have received a marker and recorded their local state and ll processes have received markers on all incoming channels and have recorded all channel states.

We can then collect the locally recorded states and construct a global snapshot.

Example

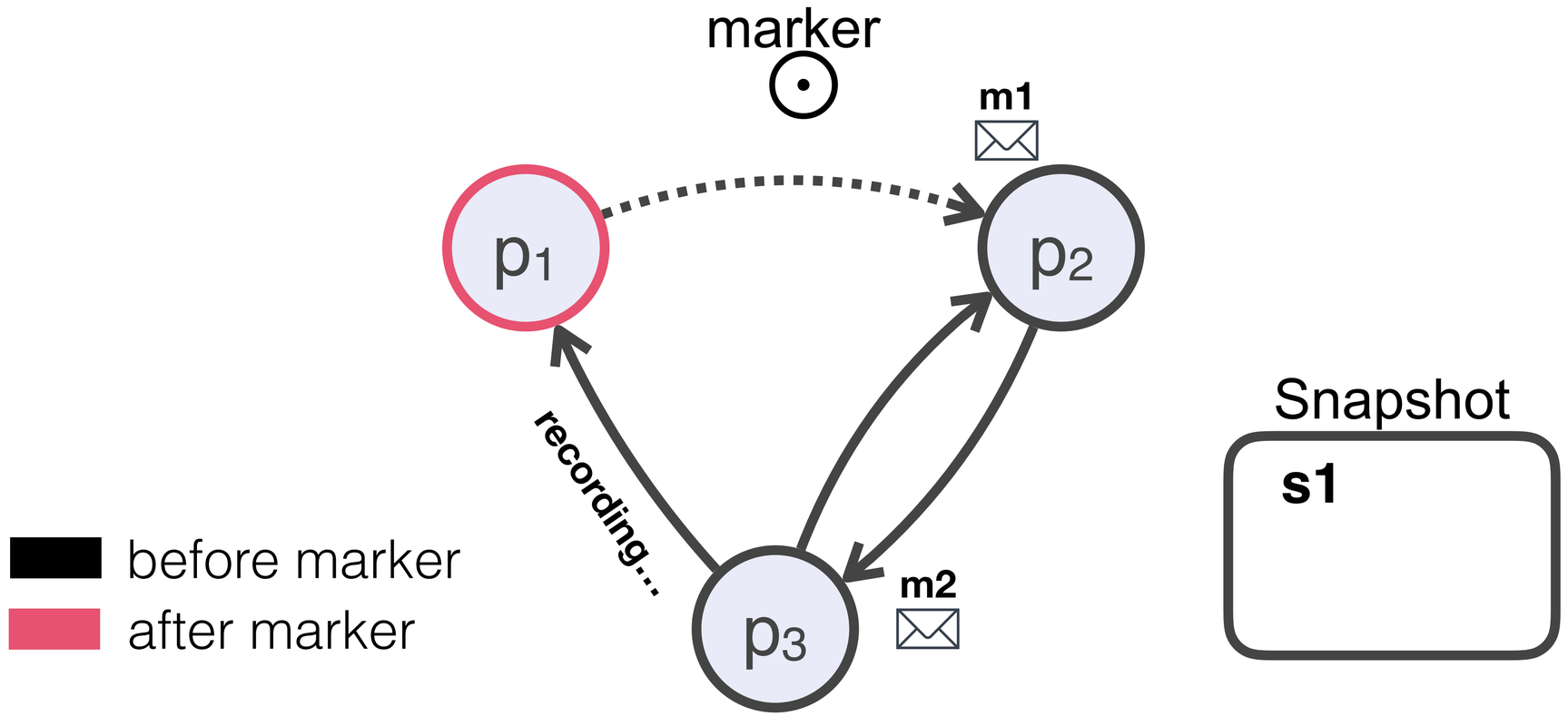


Example



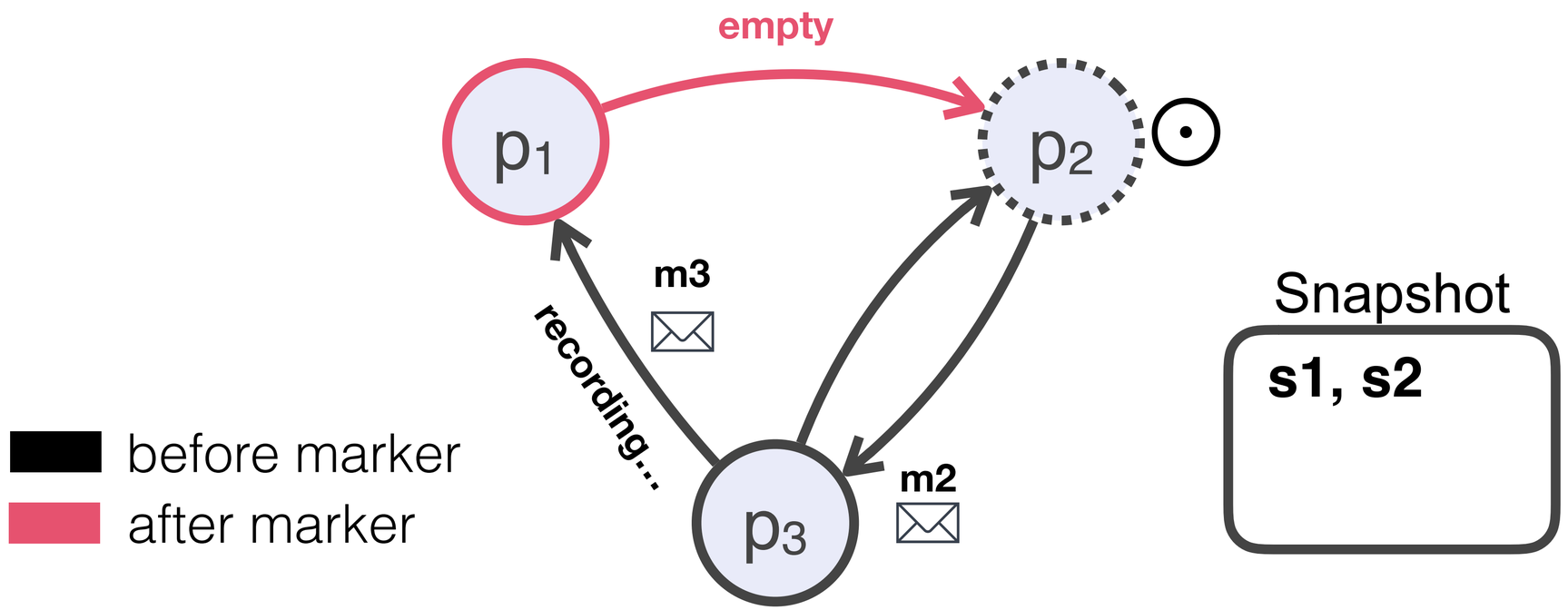
Example

p_1 records its state, forwards the marker, and starts recording on incoming channels



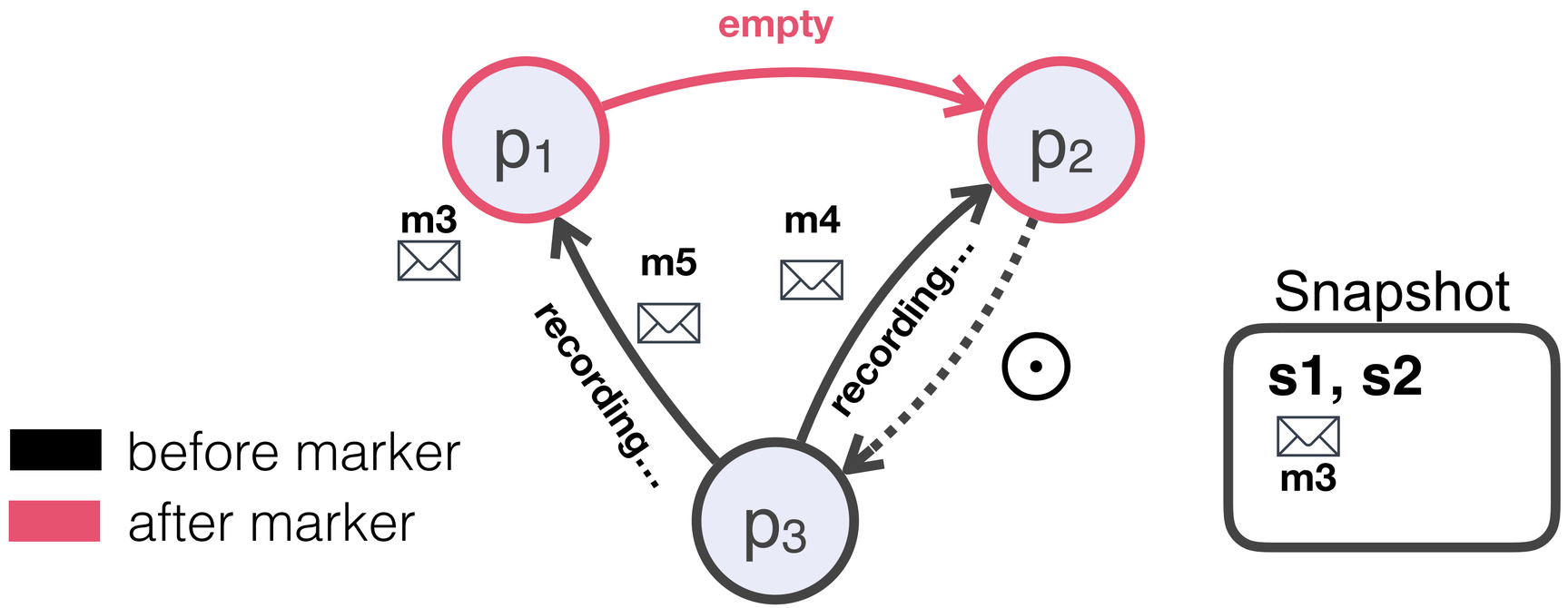
Example

p_2 receives the marker, records its state, marks marker channel as empty,



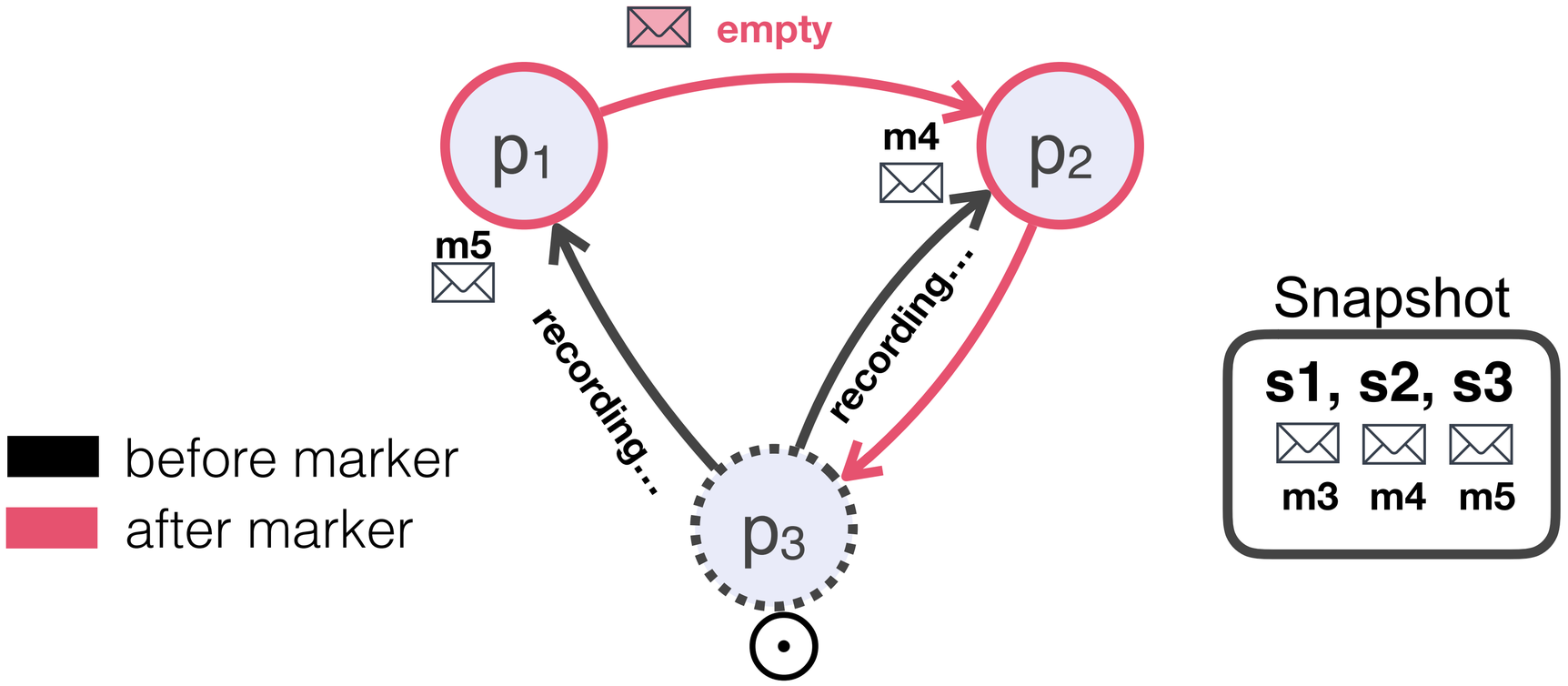
Example

p_2 receives the marker, records its state, marks marker channel as empty, forwards the marker, and starts recording on other incoming channels



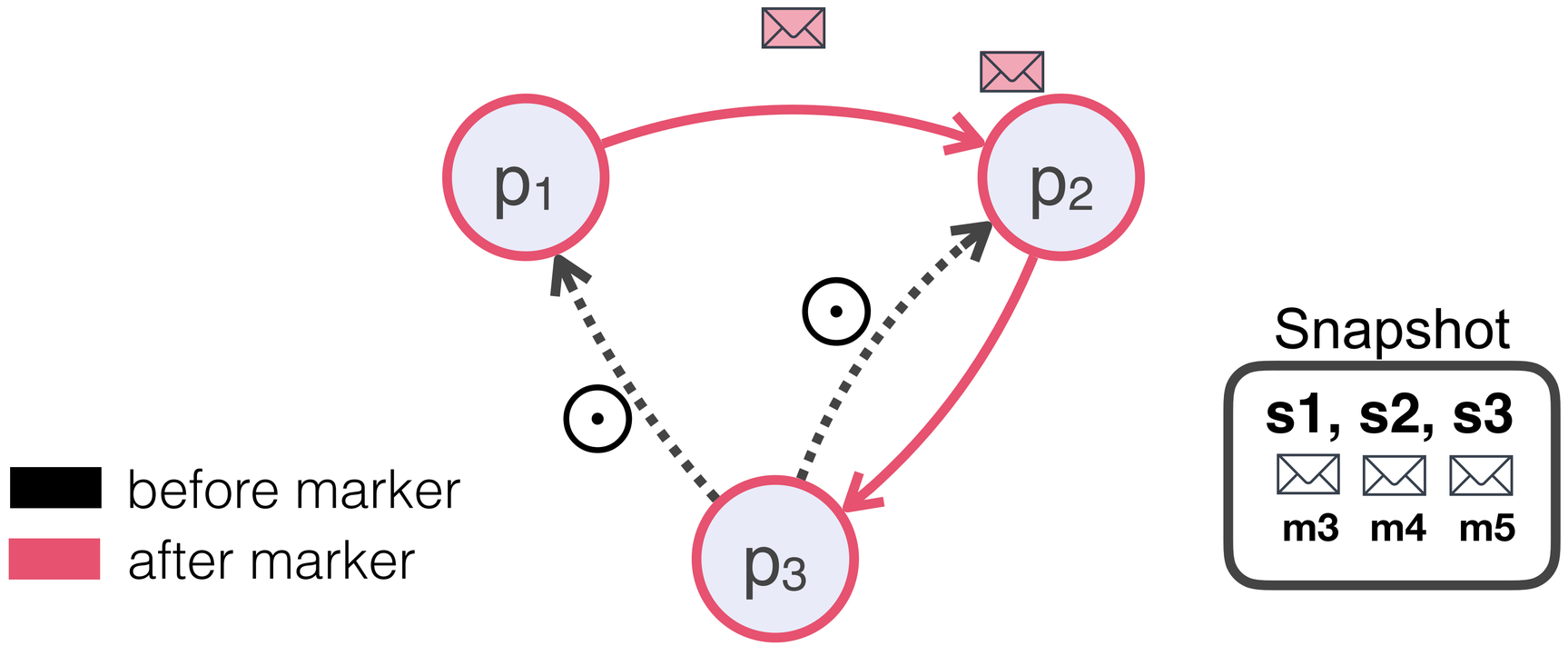
Example

p_3 receives the marker, records its state, and forwards the marker

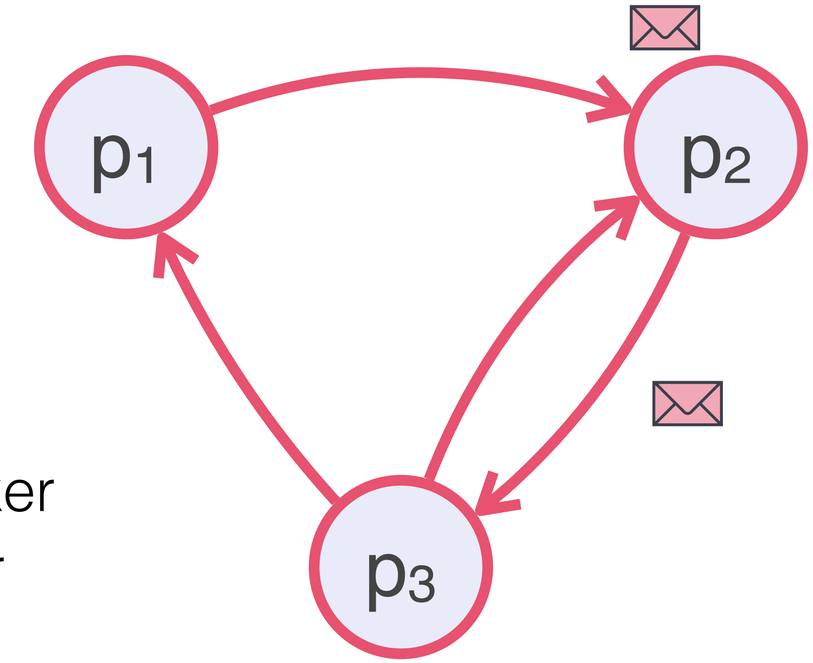


Example

p_1 and p_2 receive the marker and their remaining input channels and stop recording



Example



before marker

after marker

Snapshot

s1, s2, s3

m3 m4 m5

Does the algorithm satisfy causality?

Does the algorithm satisfy causality?

A consistent cut satisfies **causality**:

- An event is **pre-snapshot** if it occurs *before* the local snapshot on a process, otherwise it is **post-snapshot**
- If event *A happens causally before B* and B is pre-snapshot, then A is also pre-snapshot

Does the algorithm satisfy causality?

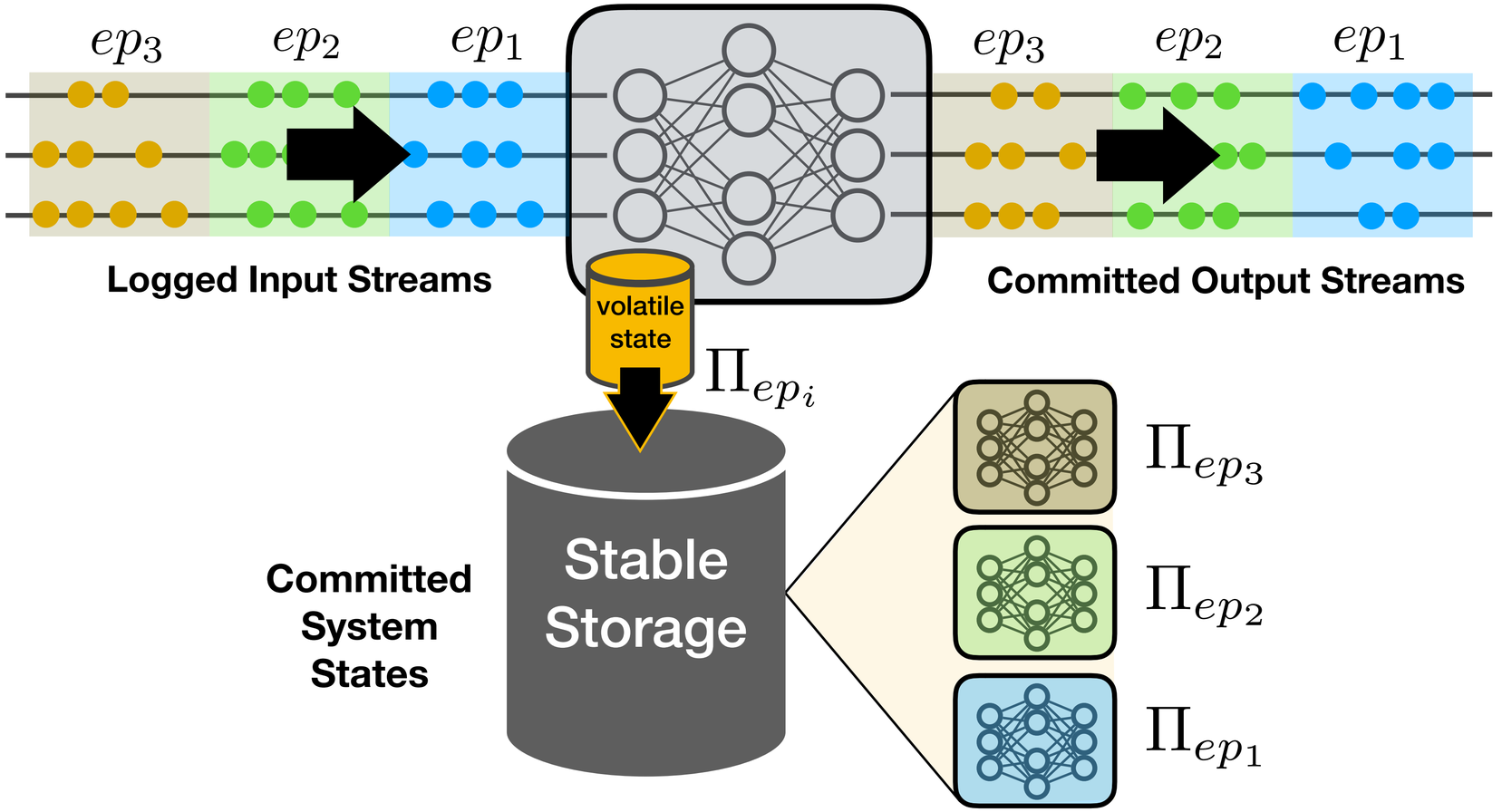
When is a message included in the snapshot?

A consistent cut satisfies **causality**:

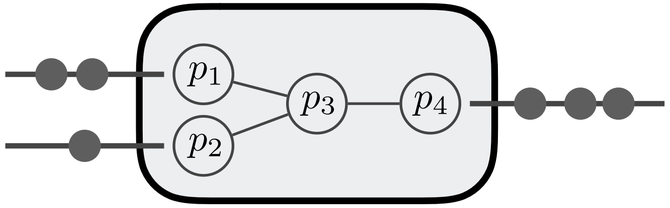
- An event is **pre-snapshot** if it occurs *before* the local snapshot on a process, otherwise it is **post-snapshot**
- If event *A happens causally before B* and B is pre-snapshot, then A is also pre-snapshot

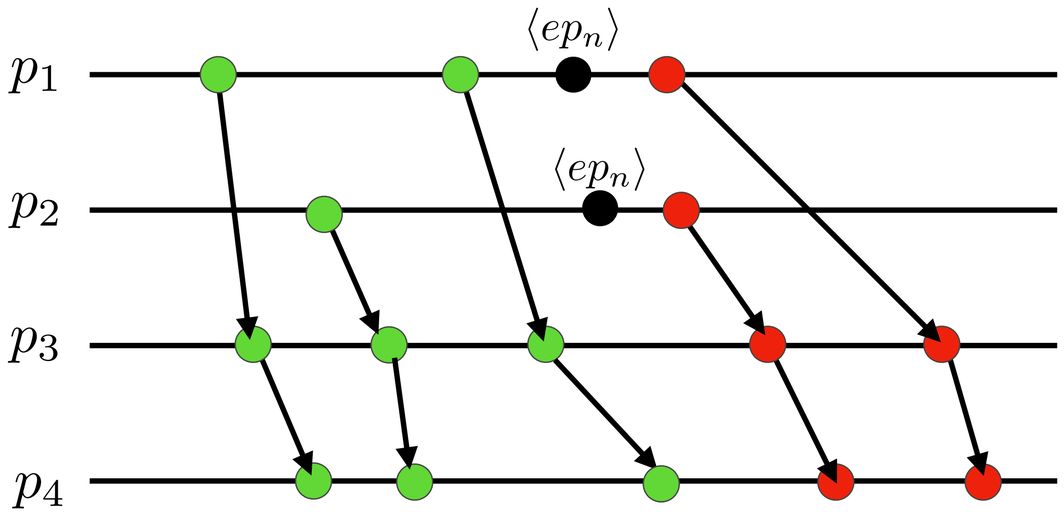
Can we apply this algorithm to retrieve a consistent snapshot of a stream processing application?

Epoch-Based Stream Execution

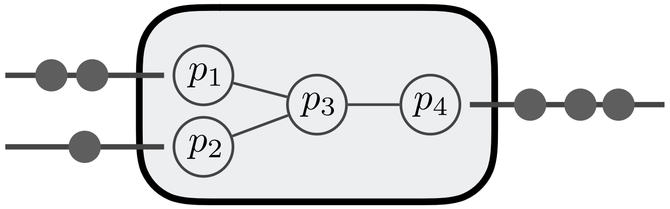


Validity is not sufficient

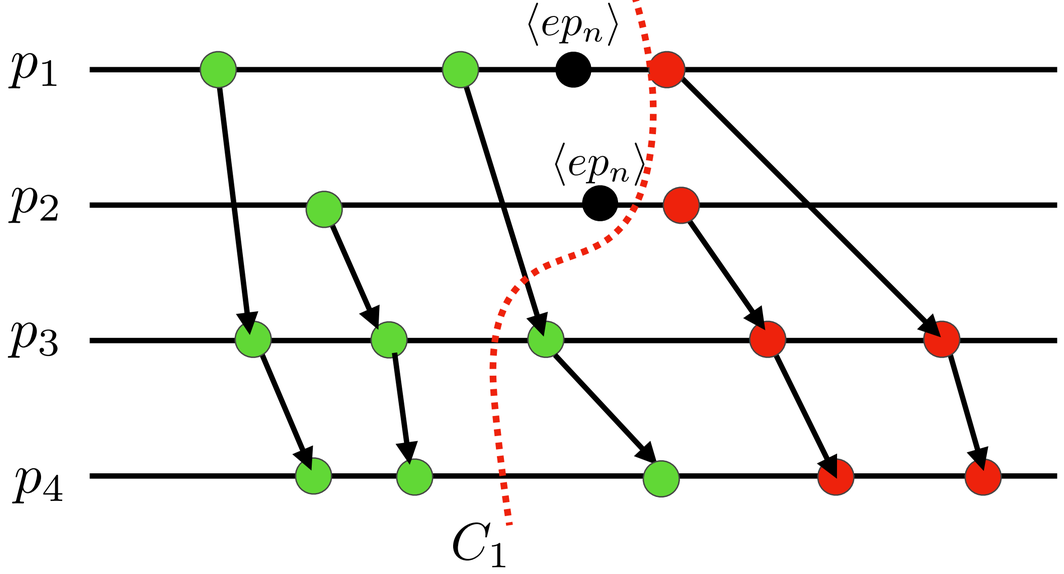




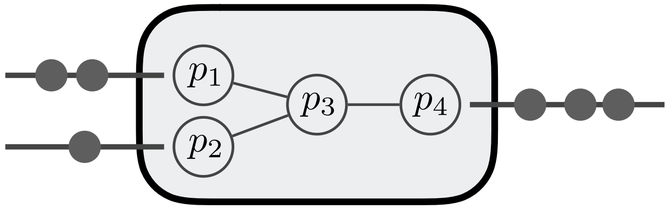
Validity is not sufficient



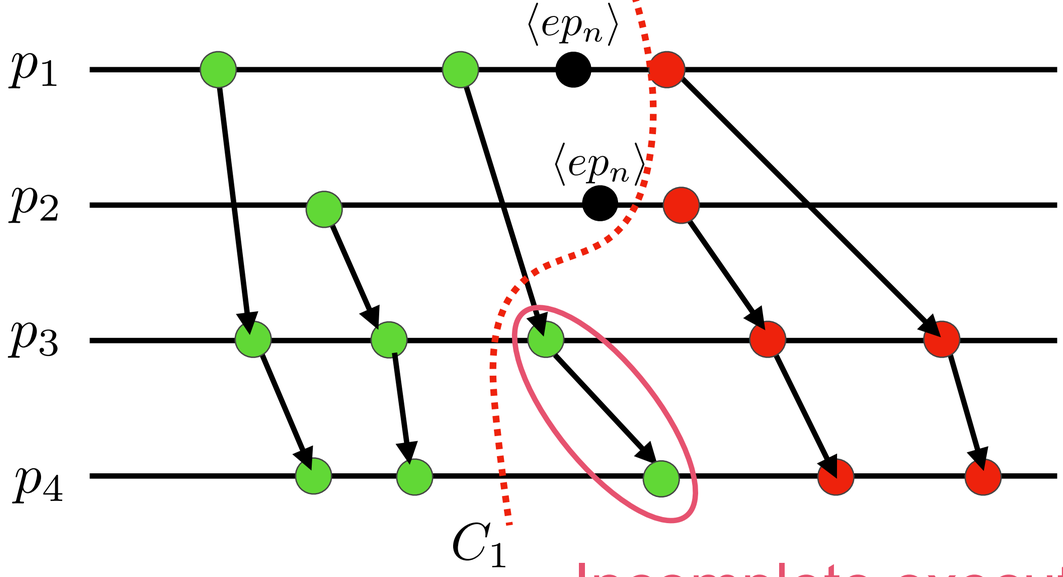
chandy-lamport snapshot



Validity is not sufficient



chandy-lamport snapshot



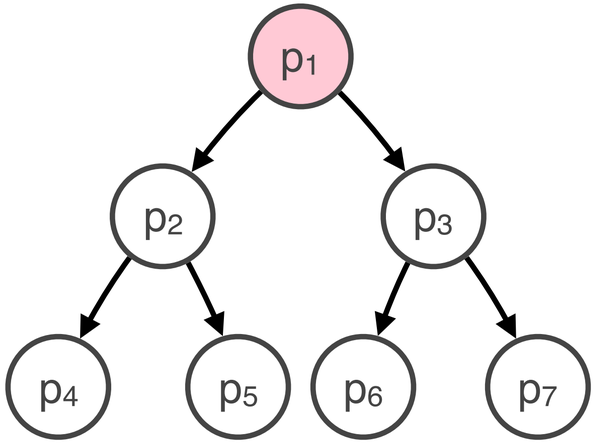
Incomplete execution of an epoch

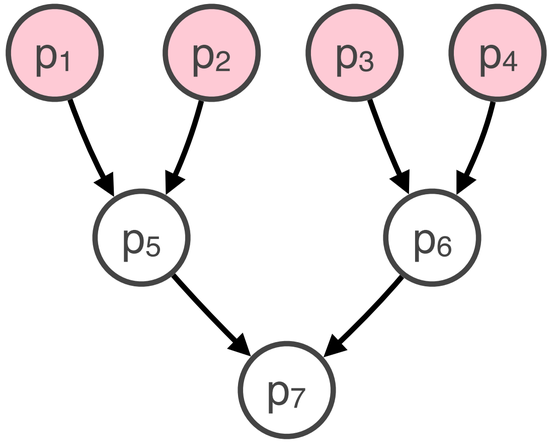
Algorithm Generalizations

The marker-forwarding logic itself guarantees **validity:**

Each local snapshotting action produces markers that **separate** pre-snapshot and post-snapshot events (order maintained by FIFO channels)

Termination is satisfied if initiator can reach all tasks (possible in DAGs via multiple initiators, e.g., sources.)

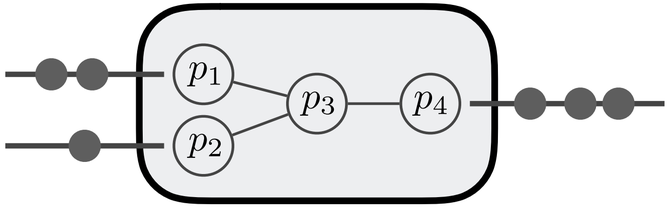


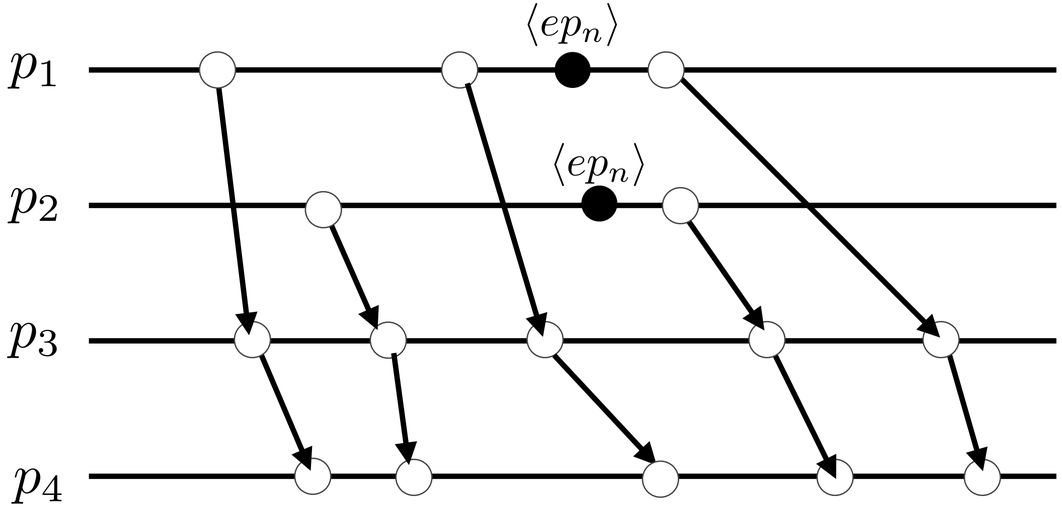


Epoch Snapshotting

- Assumptions:
  - DAG of tasks
  - Epoch change events triggered on each source task (⟨ep1⟩,⟨ep2⟩,…)
    - Issued by a coordinator or generated periodically
- We want to snapshot stream process graphs after the complete computation of an epoch.

Epoch cuts

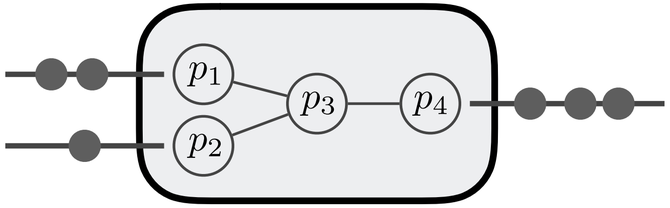


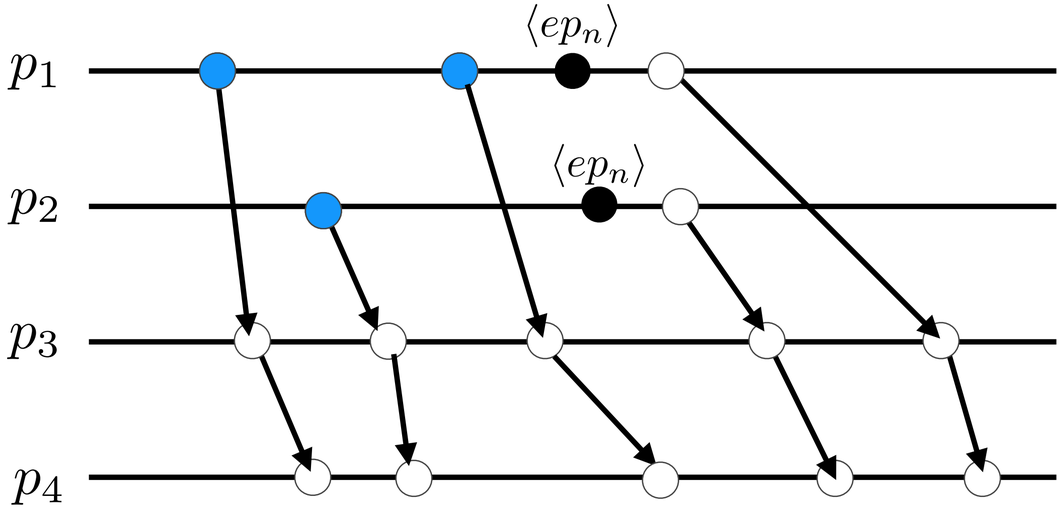


A *epoch-complete* consistent cut that includes events that

Epoch-Completeness: Obtain an epoch-complete system configuration

Epoch cuts



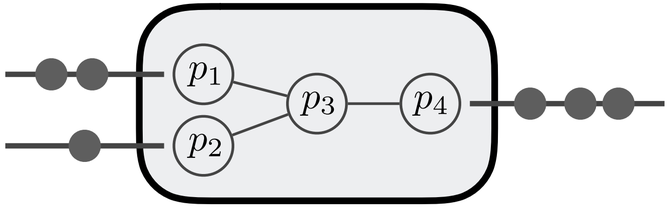


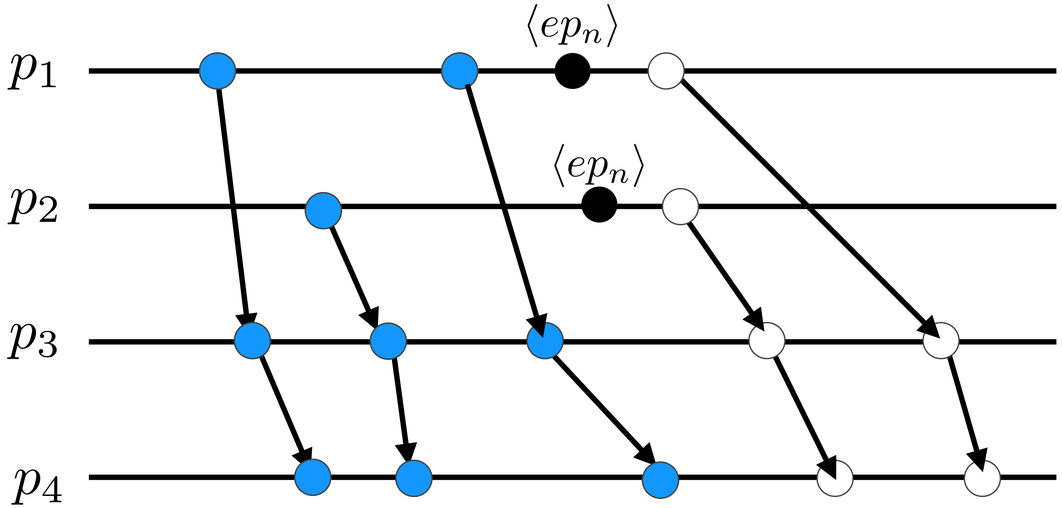
A *epoch-complete* consistent cut that includes events that

1. precede epoch change

Epoch-Completeness: Obtain an epoch-complete system configuration

Epoch cuts





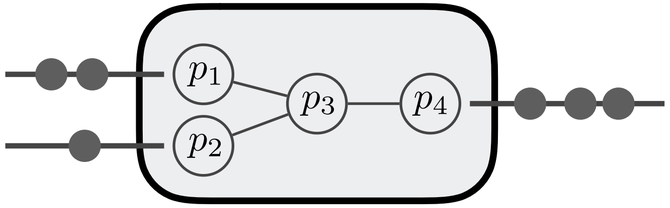
A *epoch-complete* consistent cut that includes events that

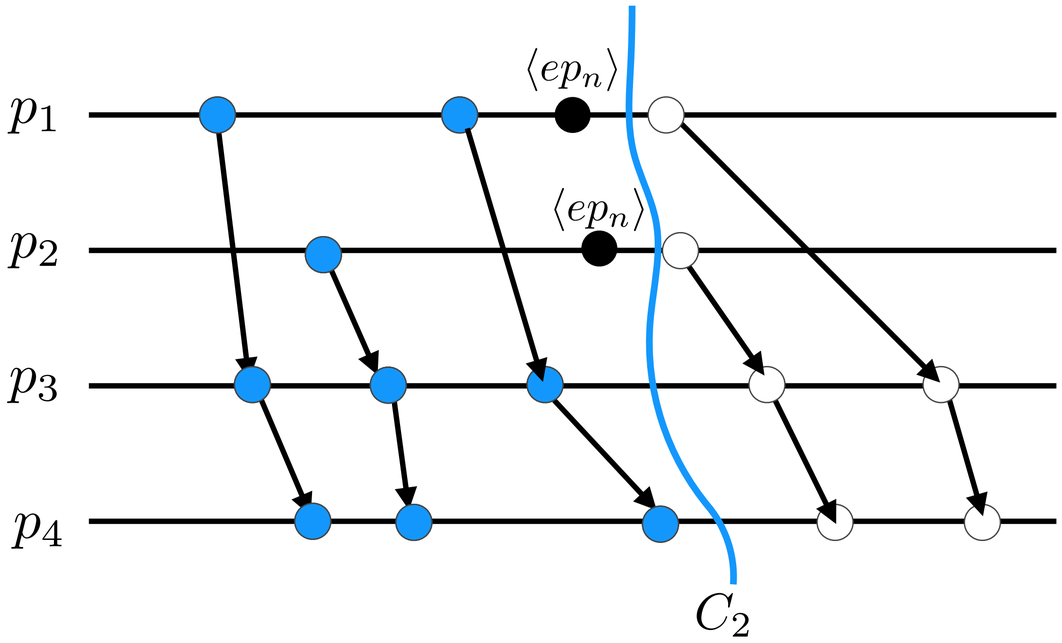
1. precede epoch change

2. are produced by events in cut

Epoch-Completeness: Obtain an epoch-complete system configuration

Epoch cuts





A *epoch-complete* consistent cut that includes events that

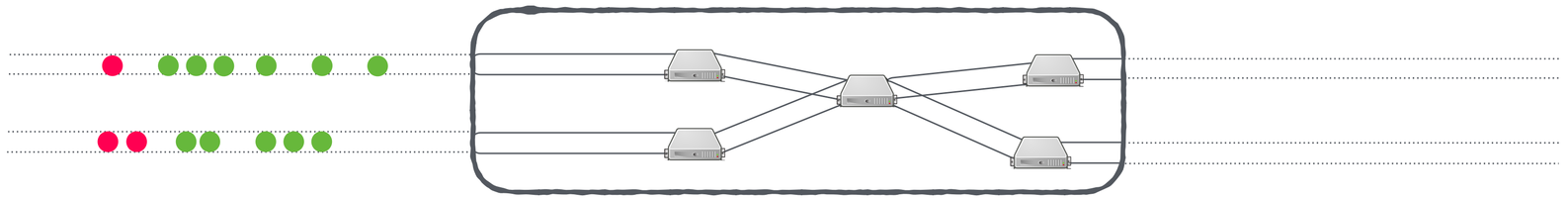
1. precede epoch change

2. are produced by events in cut

3. do **not** causally succeed epoch change

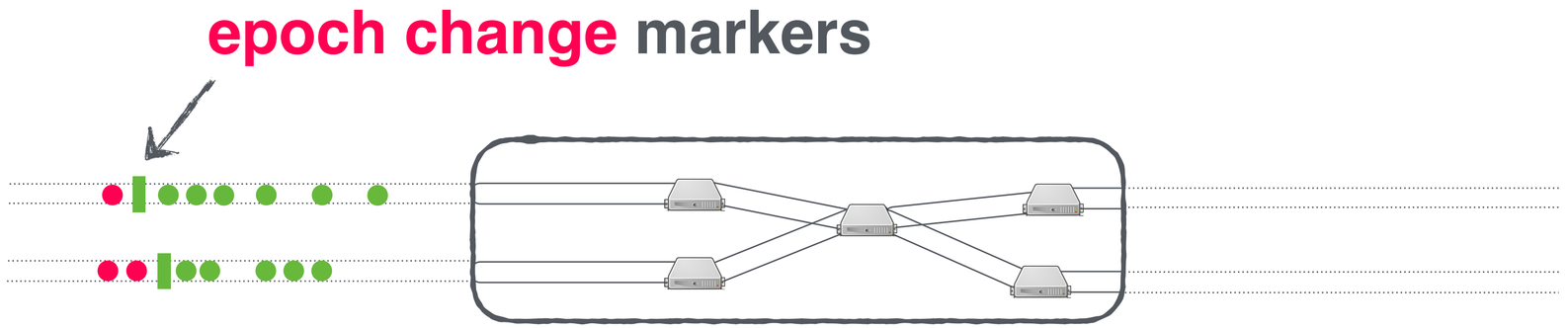
Epoch-Completeness: Obtain an epoch-complete system configuration

Epoch Snapshotting Algorithm

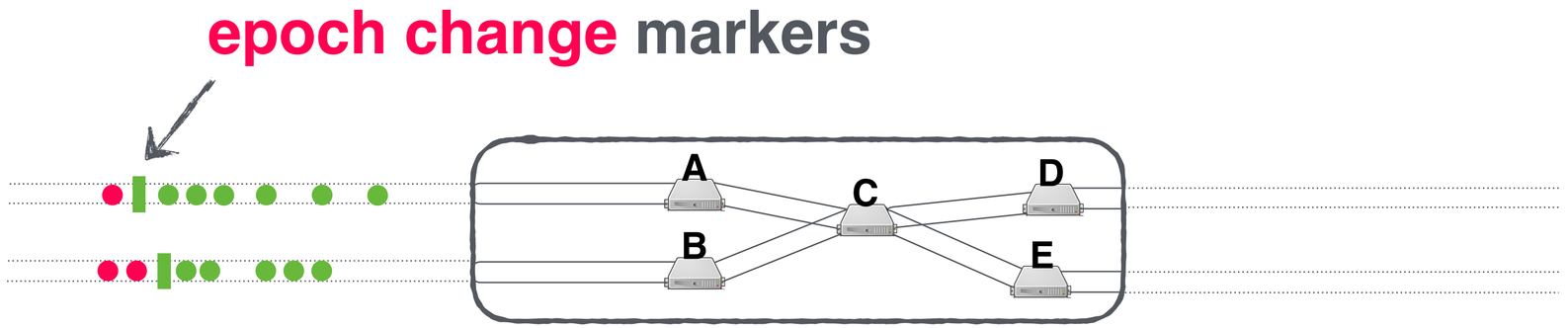


Snapshot Store

Epoch Snapshotting Algorithm

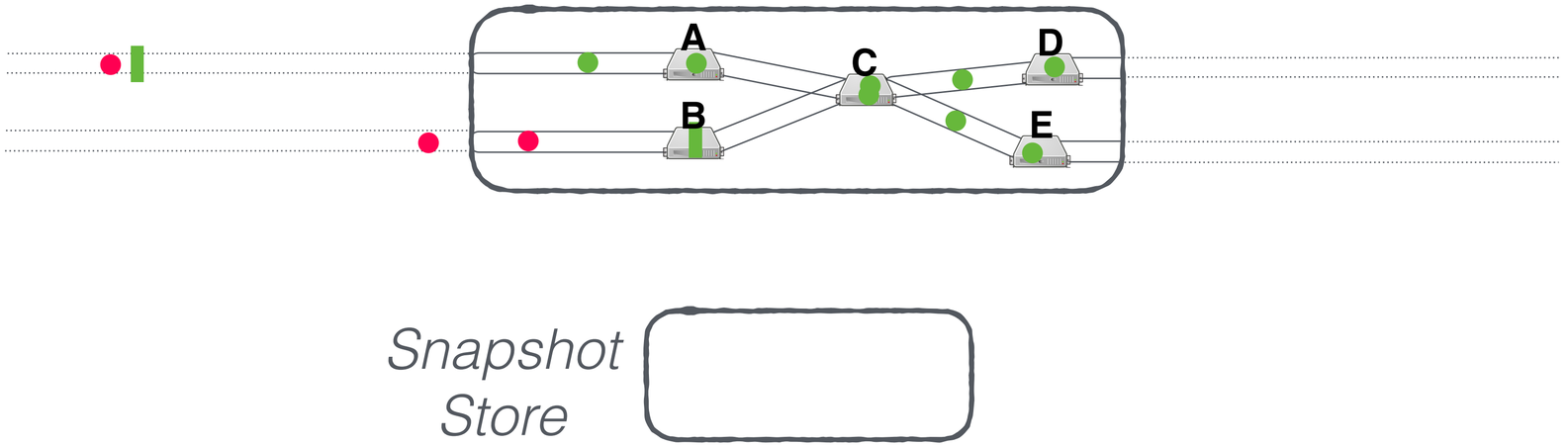


Epoch Snapshotting Algorithm

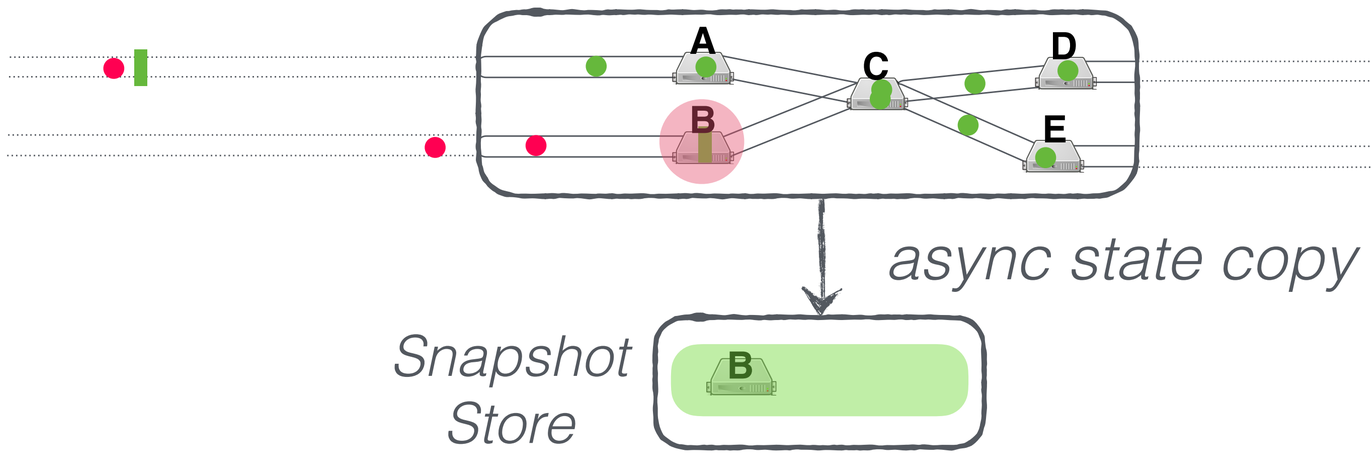


Snapshot Store

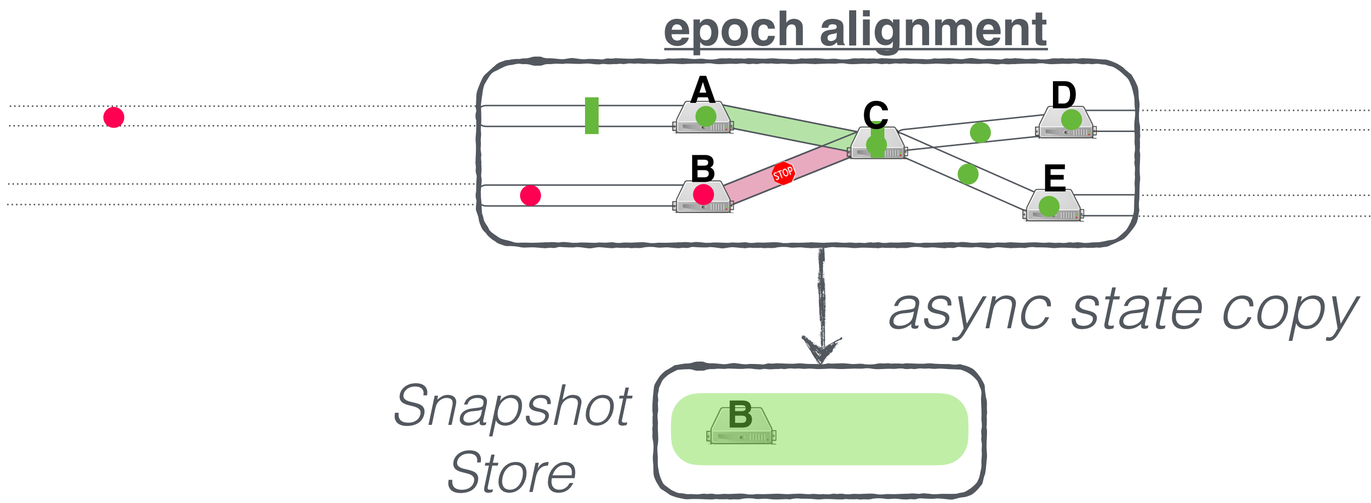
Epoch Snapshotting Algorithm



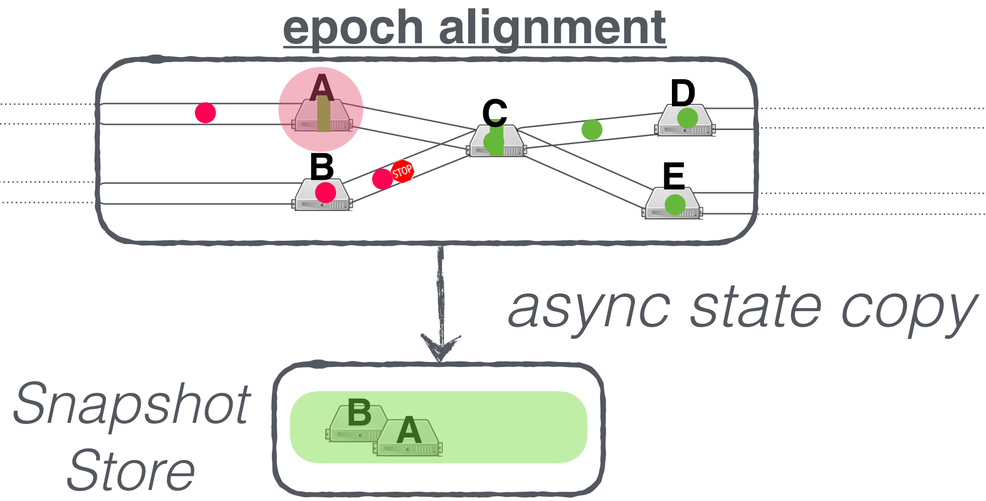
Epoch Snapshotting Algorithm



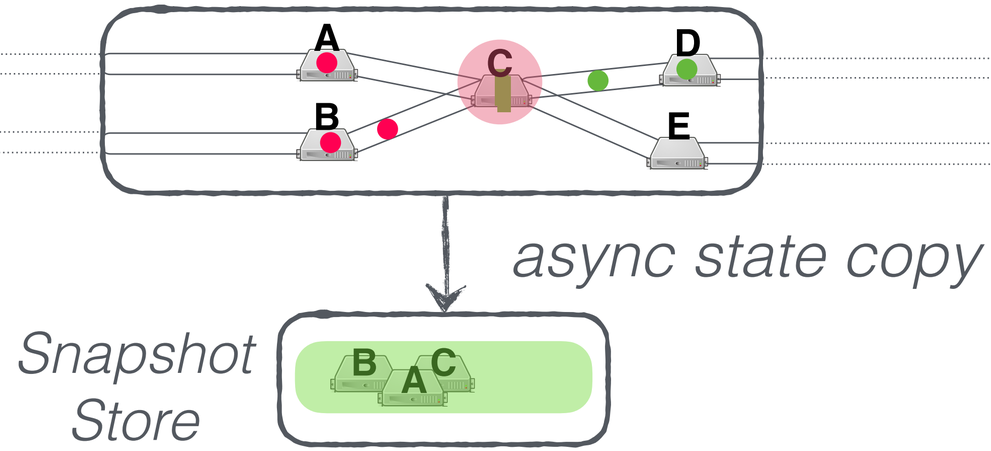
Epoch Snapshotting Algorithm



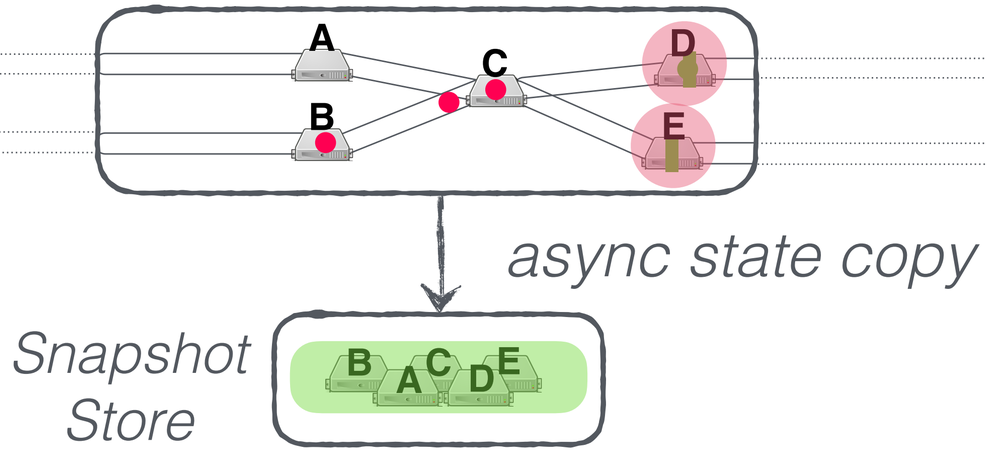
Epoch Snapshotting Algorithm



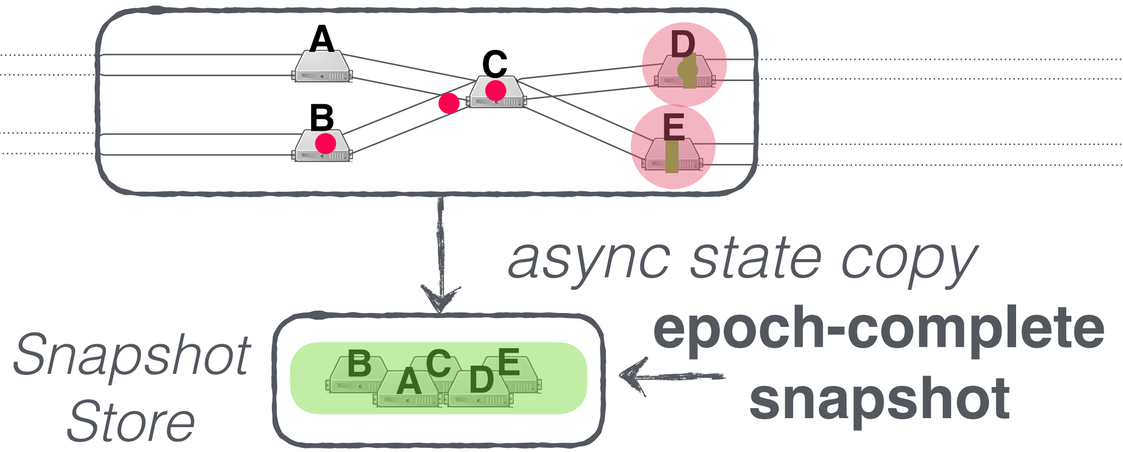
Epoch Snapshotting Algorithm



Epoch Snapshotting Algorithm



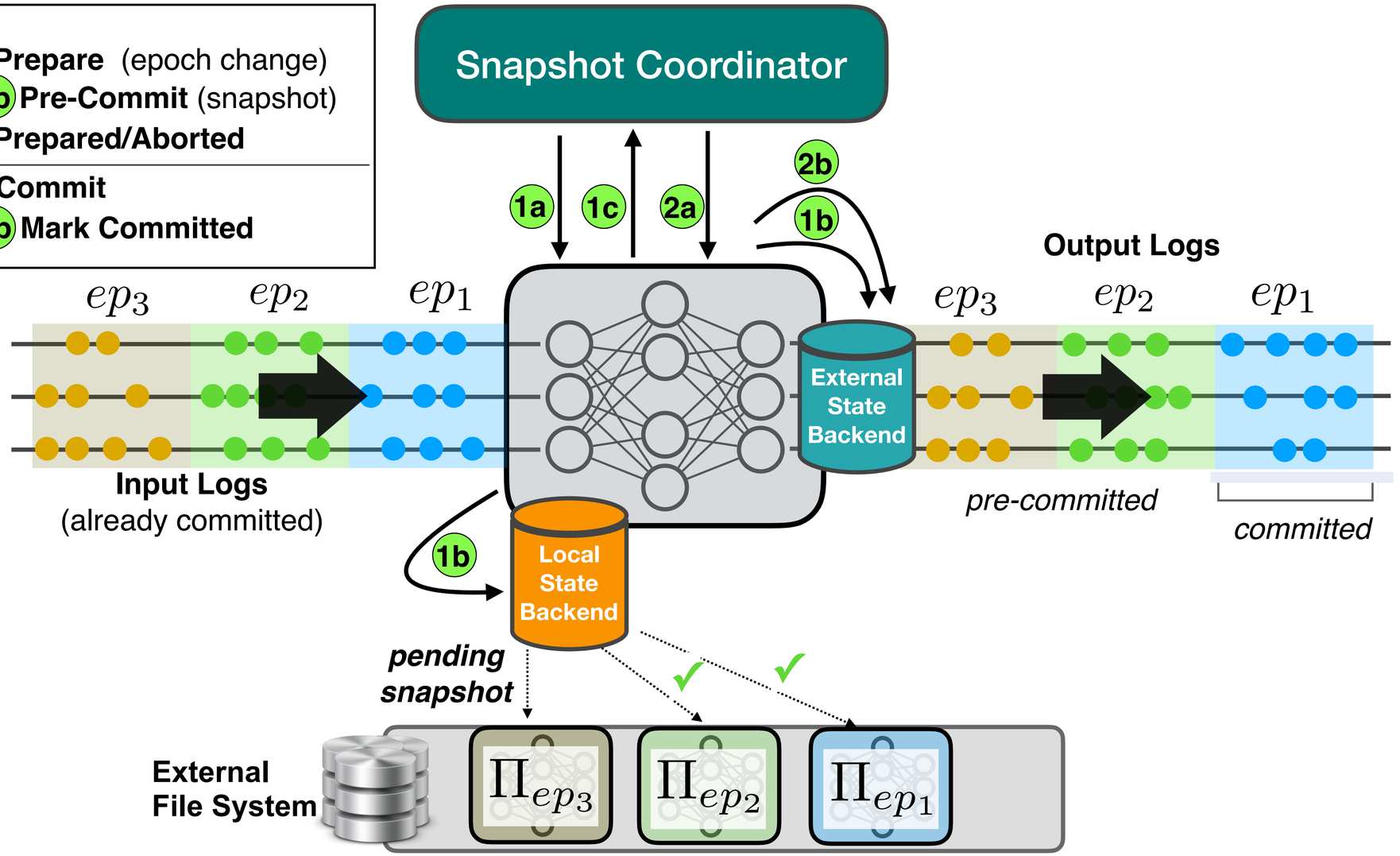
Epoch Snapshotting Algorithm



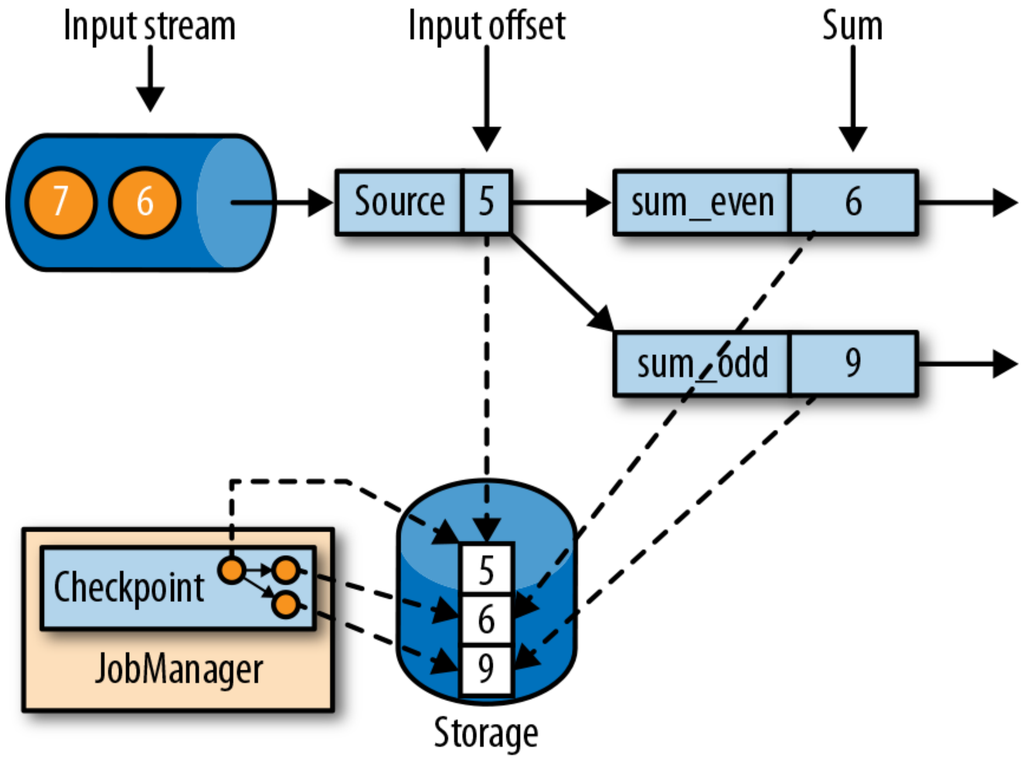
The Epoch Commit Protocol

1a **Prepare** (epoch change)
1b **Pre-Commit** (snapshot)
1c **Prepared/Aborted**

2a **Commit**
2b **Mark Committed**



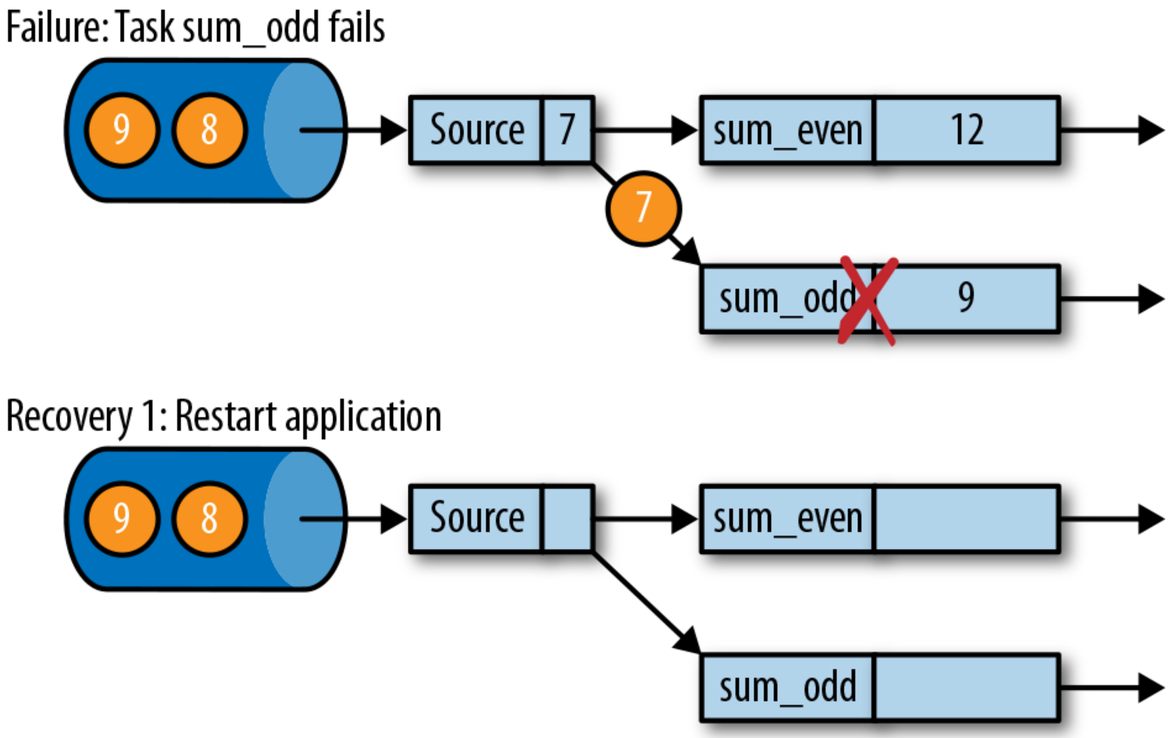
Asynchronous checkpoints in Apache Flink



- A source of increasing numbers partitioned into a stream of even and odd numbers
- Two sum operators maintaining the running sums of even and odd numbers
- The snapshot contains the source offset 5 and the sums 6 and 9

Recovery process

1. Stop and restart the application. All operators have empty state.

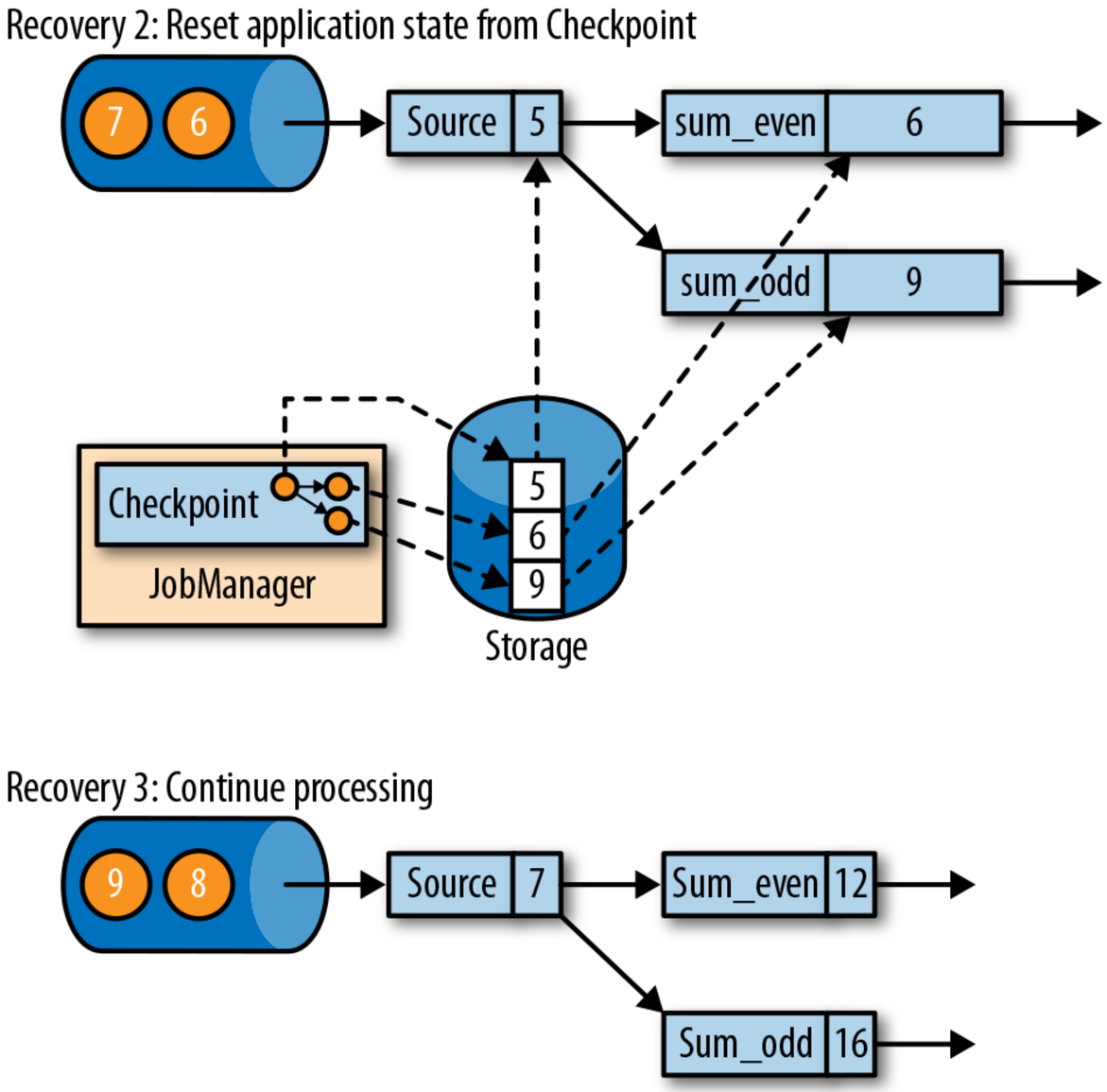


Recovery process

1. Stop and restart the application. All operators have empty state.

2. Initialize stateful operators from the latest checkpoint.

3. Resume processing.



Re-settable sources

- All input streams are reset to the position up to which they were consumed when the checkpoint was taken.
- Event logs like Apache Kafka can provide records from a previous offset of the stream.

Re-settable sources

- All input streams are reset to the position up to which they were consumed when the checkpoint was taken.
- Event logs like Apache Kafka can provide records from a previous offset of the stream.

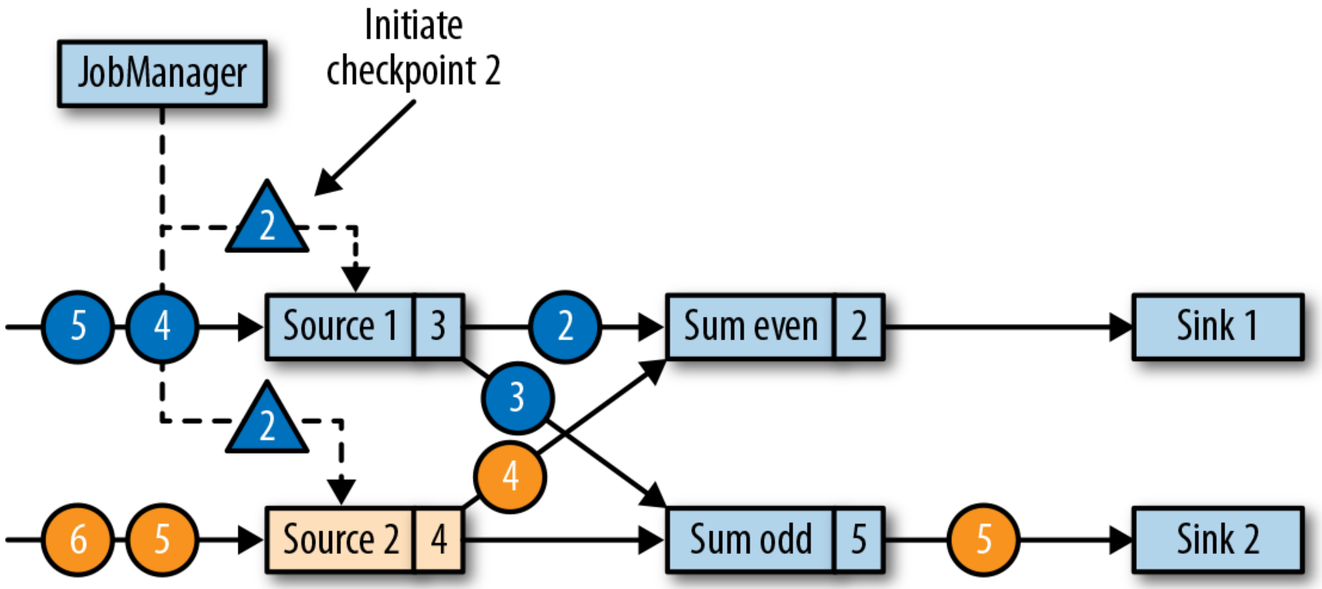
What if the input stream comes from a socket?

Re-settable sources

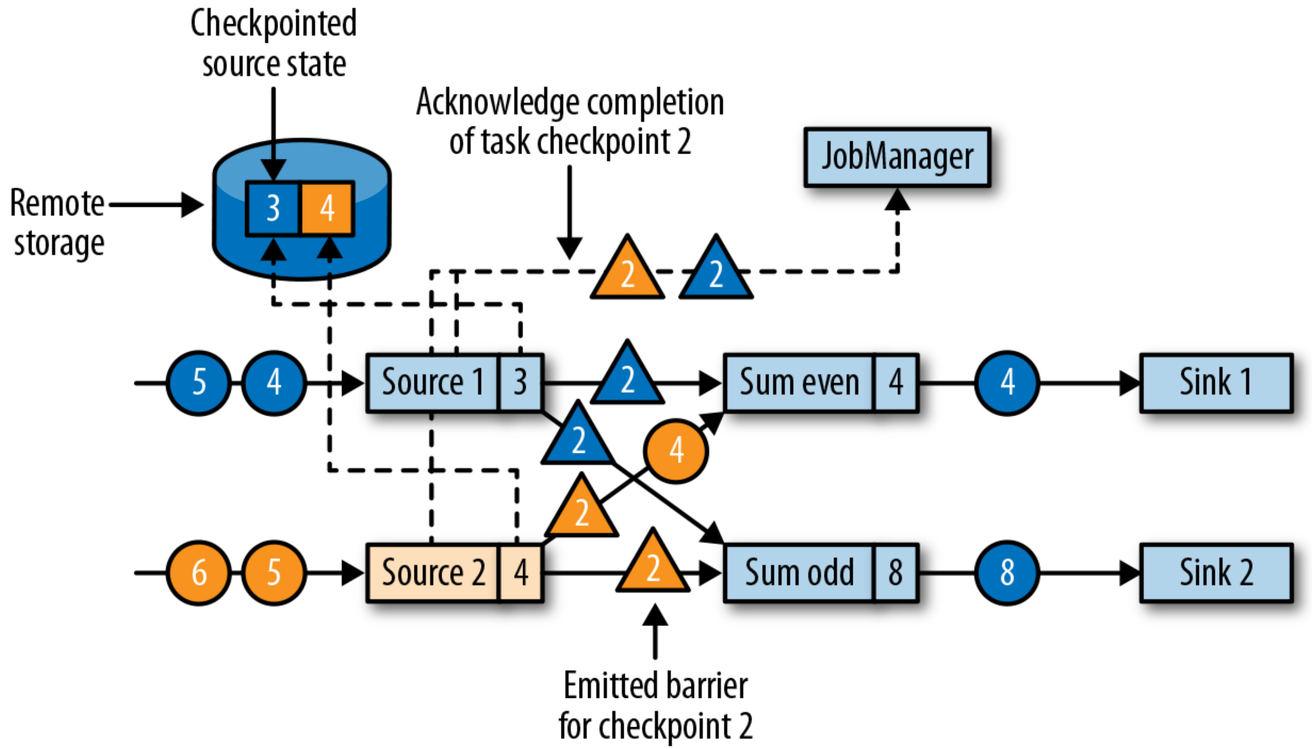
- All input streams are reset to the position up to which they were consumed when the checkpoint was taken.
- Event logs like Apache Kafka can provide records from a previous offset of the stream.

What if the input stream comes from a socket?

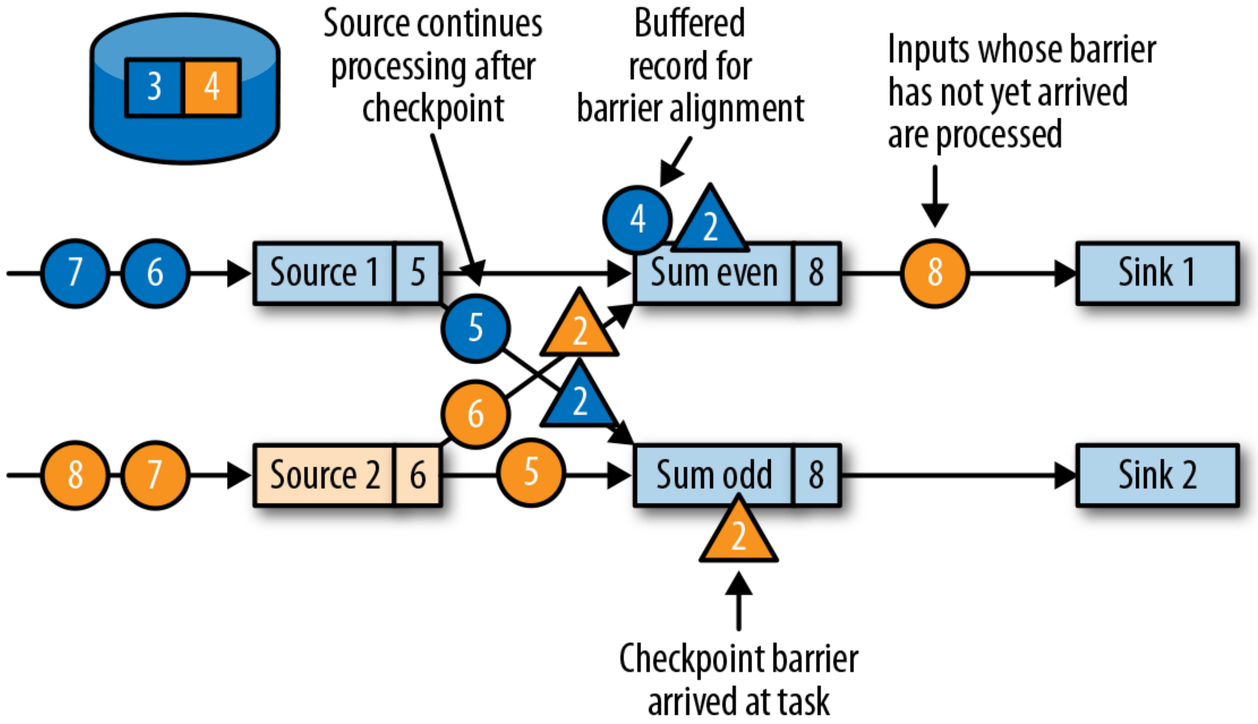
Exactly-once state consistency (in Apache Flink) can be achieved only if all streaming sources are re-settable



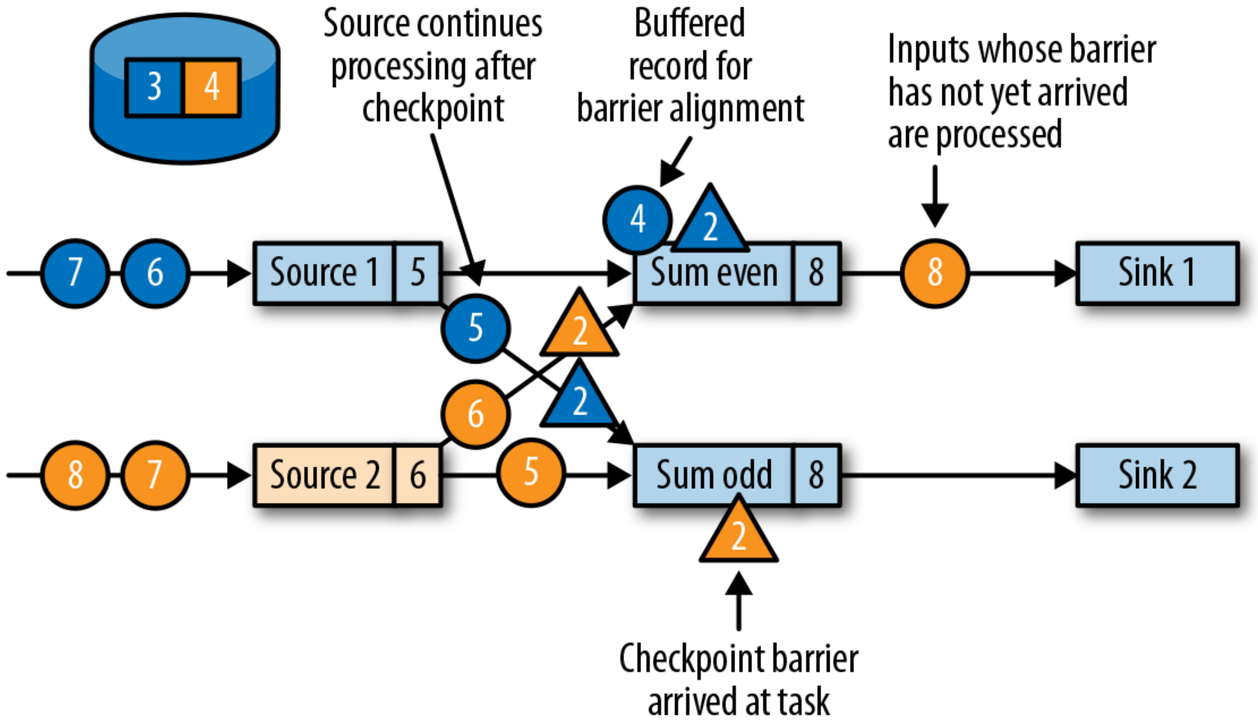
- Flink checkpoints are initiated by the JobManager and have a unique Checkpoint ID.
- The checkpoint barrier flows from the sources to the sink of the dataflow.



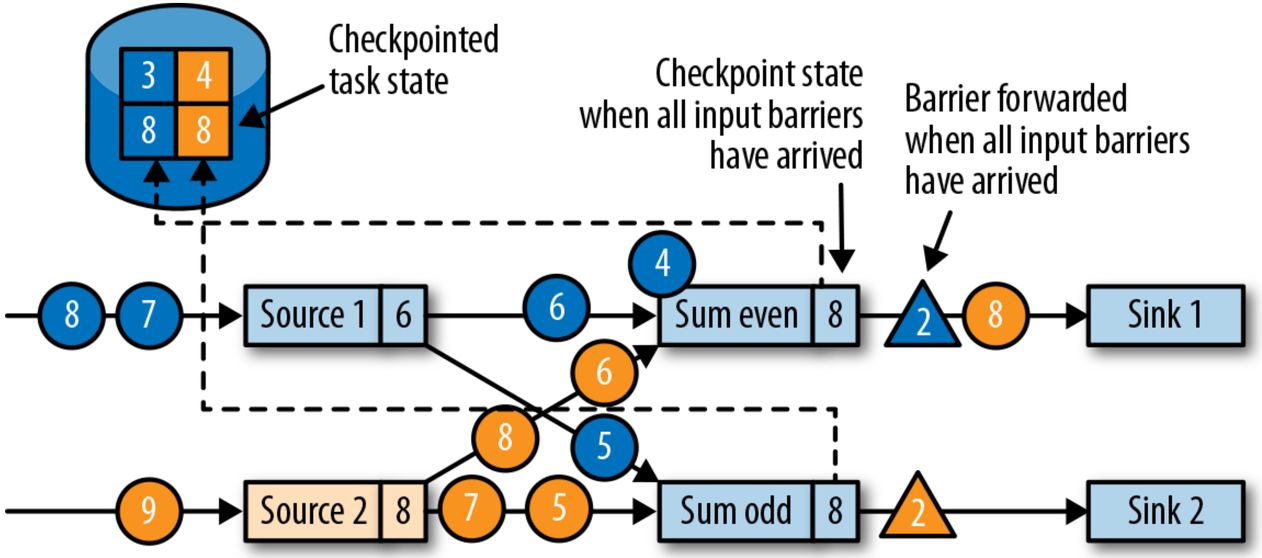
- When a source task receives a checkpoint barrier, it pauses emitting records, triggers a checkpoint of its local state at the state backend, and broadcasts barriers to all outgoing stream partitions.
- The state backend notifies the task once its state checkpoint is complete.
- The task acknowledges the checkpoint to the JobManager. After all barriers are sent out, the source continues its regular operations.

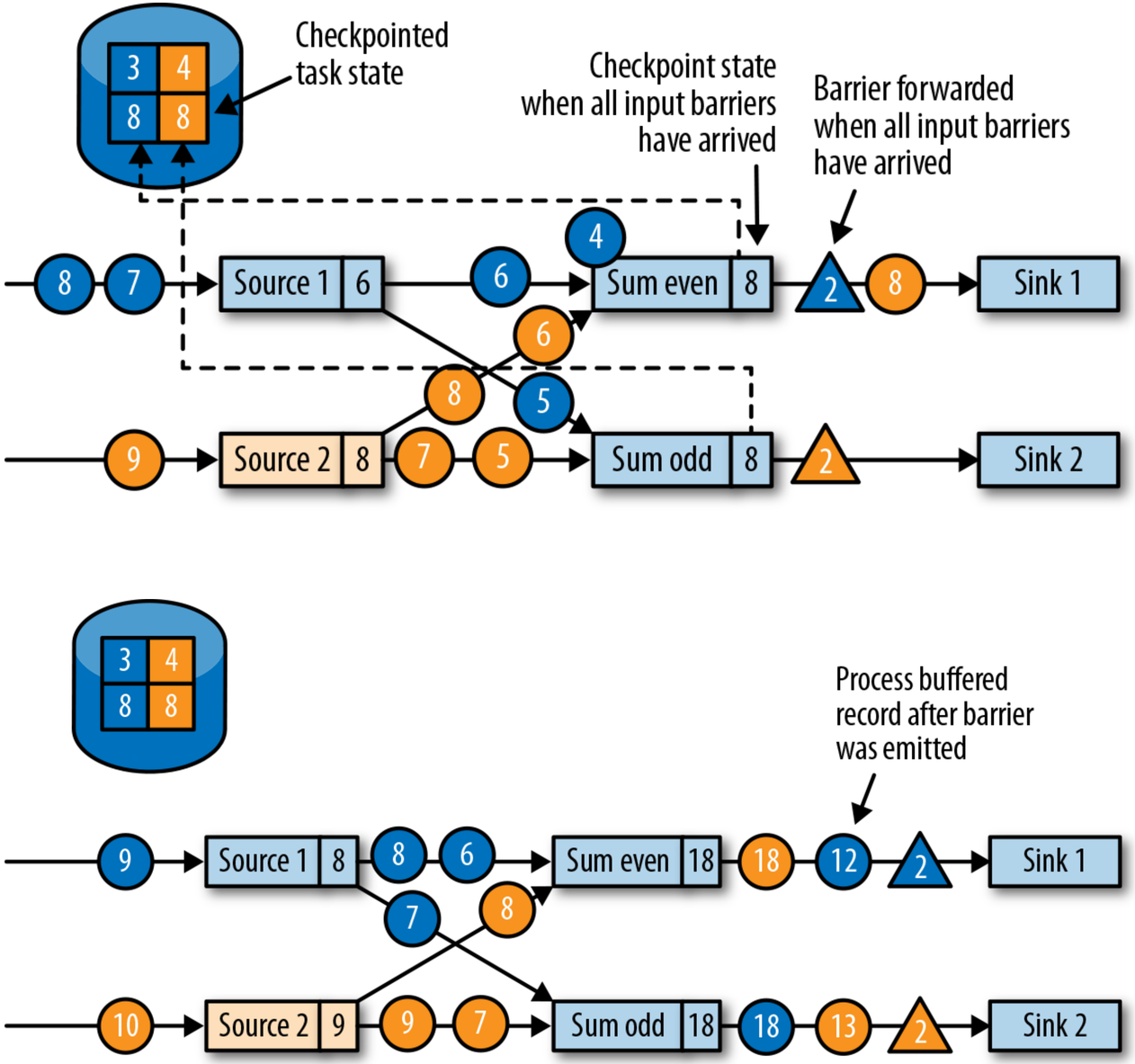


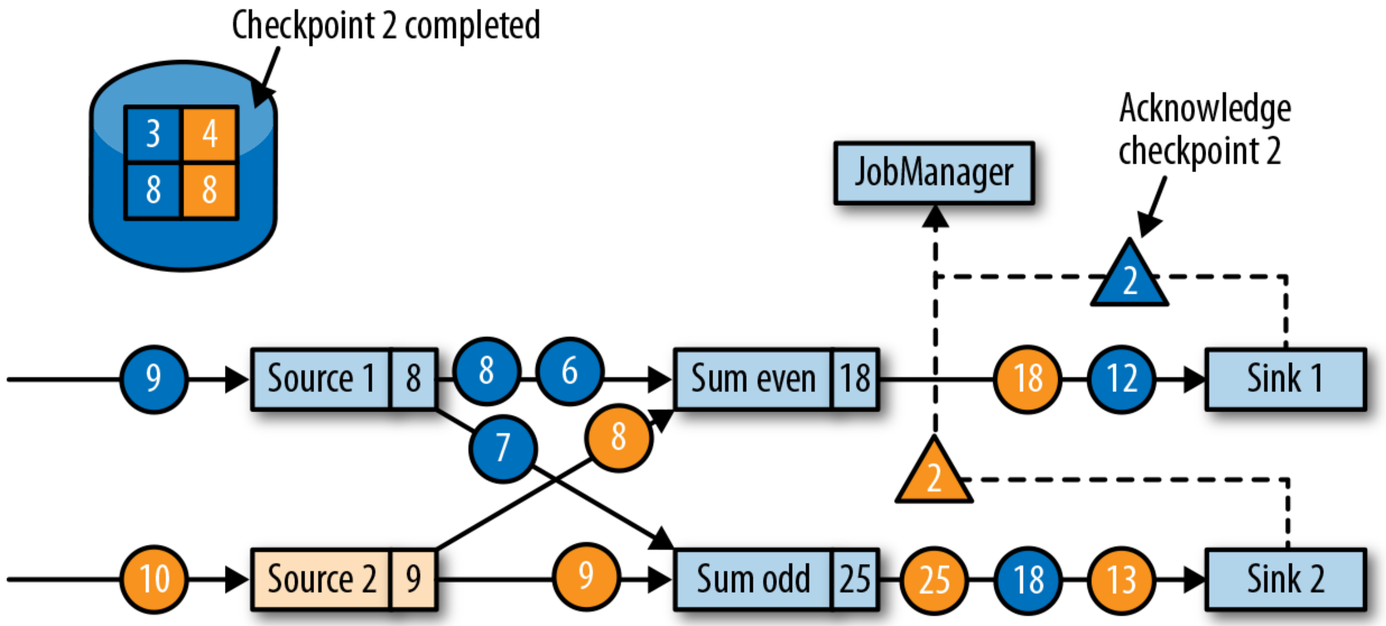
- When a task receives a checkpoint barrier, it waits for the arrival of barriers from all its input partitions for the checkpoint.
- In the meantime, it can process records from stream partitions that did not provide a barrier yet.
- Records that arrive on partitions that forwarded a barrier already cannot be processed and are buffered. This process is called **barrier alignment**.



How would state guarantees change if we skipped barrier alignment?


Checkpointed
task state
3
4
8
8
Checkpoint state
when all input barriers
have arrived
Barrier forwarded
when all input barriers
have arrived
8
7
Source 1
6
6
4
Sum even
8
2
8
Sink 1
6
8
5
9
Source 2
8
7
5
Sum odd
8
2
Sink 2


Checkpointed task state
Checkpoint state when all input barriers have arrived
Barrier forwarded when all input barriers have arrived
Source 1
Source 2
Sum even
Sum odd
Sink 1
Sink 2
Process buffered record after barrier was emitted



- When a sink task receives a barrier, it performs a barrier alignment, checkpoints its own state, and sends an acknowledgement to the JobManager.
- The checkpoint is completed once the JobManager has received acknowledgements from all tasks.

Performance implications

How may checkpointing affect application performance?

Performance implications

How may checkpointing affect application performance?

How often to checkpoint?

Performance implications

- **How may checkpointing affect application performance?**
- **How often to checkpoint?**
- **Do we need to checkpoint the complete application state in every checkpoint?**

Performance implications

How may checkpointing affect application performance?

How often to checkpoint?

Do we need to checkpoint the complete application state in every checkpoint?

- RocksDB supports both **asynchronous** and **incremental** checkpoints:
  - take a local snapshot and use a background thread to copy the state to remote storage
  - compute state deltas to reduce data transfer

End-to-end exactly once

- Flink's checkpointing and recovery mechanism only resets the internal state of a streaming application
- Some result records might be emitted multiple times to downstream systems

End-to-end exactly once

- Flink's checkpointing and recovery mechanism only resets the internal state of a streaming application
- Some result records might be emitted multiple times to downstream systems

How can we ensure exactly-once output?

Enabling and configuring checkpoints

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// set checkpointing interval to 10 seconds
env.enableCheckpointing(10000L)
```

```
// get the CheckpointConfig from the StreamExecutionEnvironment
val cpConfig: CheckpointConfig = env.getCheckpointConfig

// set mode to at-least-once
cpConfig.setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE);

// make sure we process at least 30s without checkpointing
cpConfig.setMinPauseBetweenCheckpoints(30000);

// allow three checkpoints to be in progress at the same time
cpConfig.setMaxConcurrentCheckpoints(3);

// checkpoints have to complete within five minutes, or are discarded
cpConfig.setCheckpointTimeout(300000);

// do not fail the job on a checkpointing error
cpConfig.setFailOnCheckpointingErrors(false);
```

Lecture references

- Paris Carbone et al. **State management in Apache Flink®: consistent stateful distributed stream processing**. (PVLDB 2017).

- Fabian Hueske, and Vasiliki Kalavri. **Stream Processing with Apache Flink**. (O'Reilly Media '19).

- A nice blog post and example: http://composition.al/blog/2019/04/26/an-example-run-of-the-chandy-lamport-snapshot-algorithm/

- A video lecture on global snapshots: https://www.coursera.org/lecture/cloud-computing/1-2-global-snapshot-algorithm-hndGi