ID2203

KTH-2021

Advanced Course Distributed Systems

Consistent Snapshotting

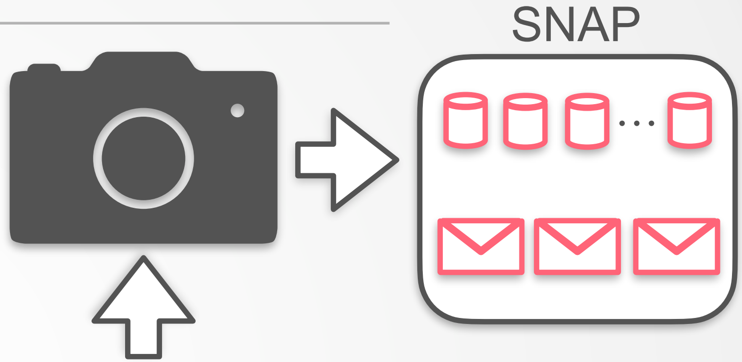
Paris Carbone

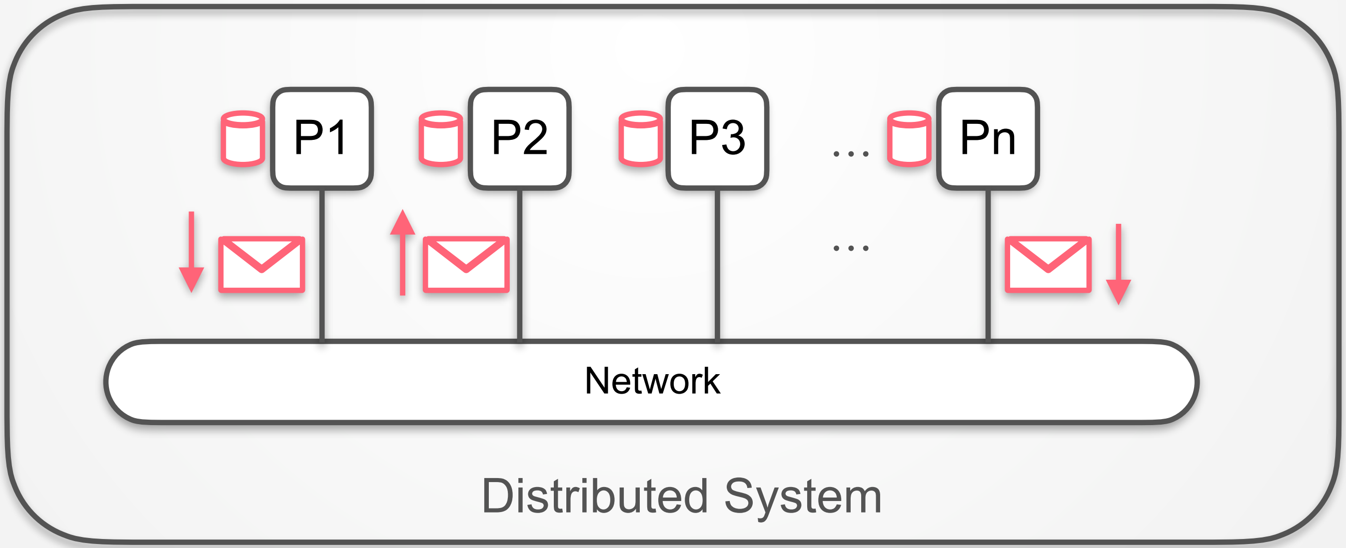
Course Topics

- Intro to Distributed Systems
- Basic Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Shared Memory
- Consensus (Single-Value / Sequence / Byzantine)
- Dynamic Reconfiguration
- Time Abstractions and Interval Clocks
- **Consistent Snapshotting**

DISTRIBUTED SNAPSHOTS

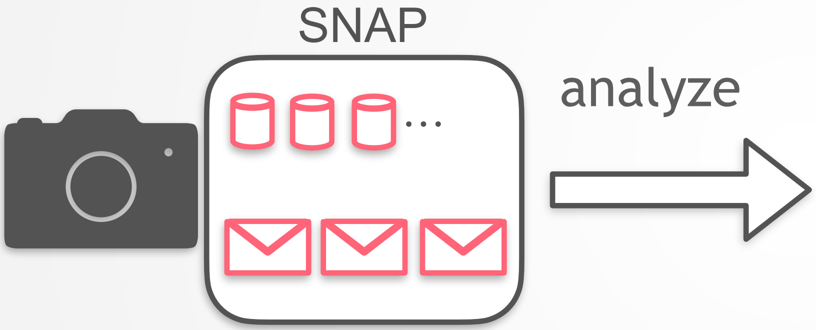
- Distributed algorithms that capture the **global state** of a distributed system.





Snapshot Usages

1. Stable Property Detection



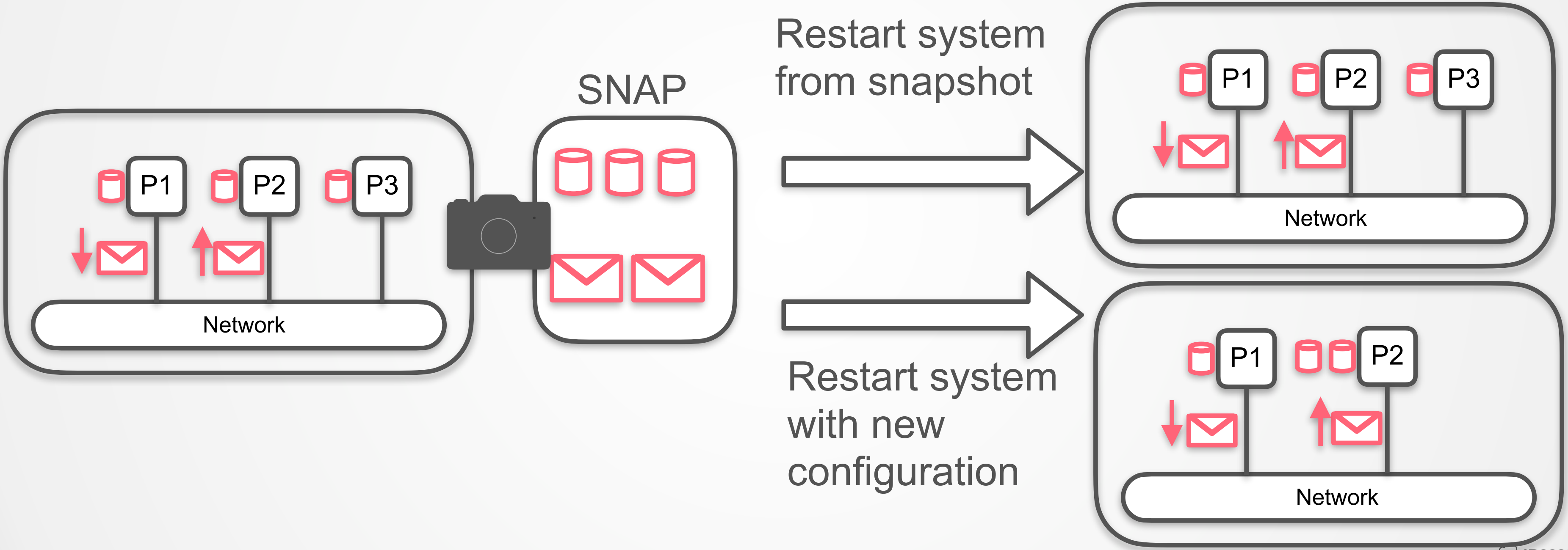
- Deadlocked execution
- Computation Terminated
- No tokens in transit

"A stable property is one that persists: once a stable property becomes true it remains true thereafter"

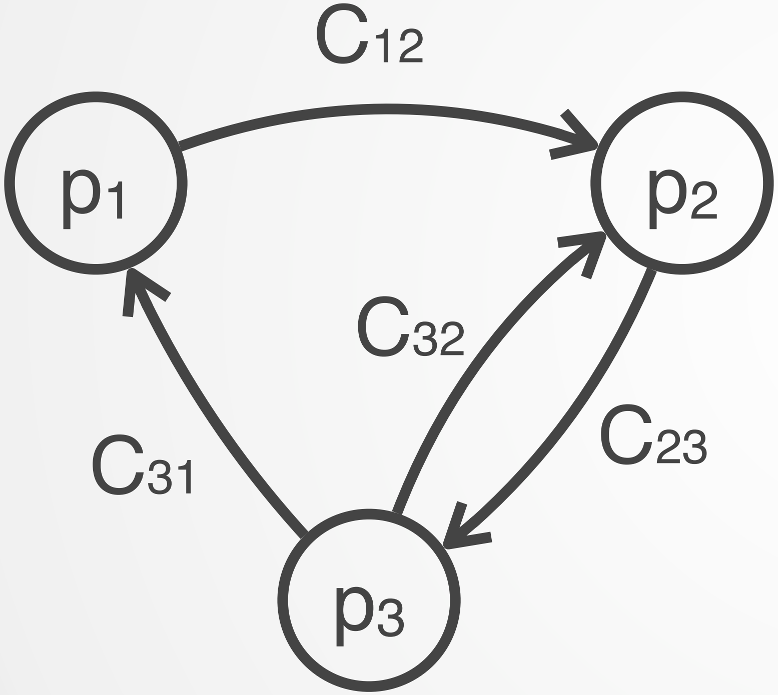
- Chandy, Lamport 88

SNAPSHOT USAGES

2. Failure Recovery and Reconfiguration



Process Model



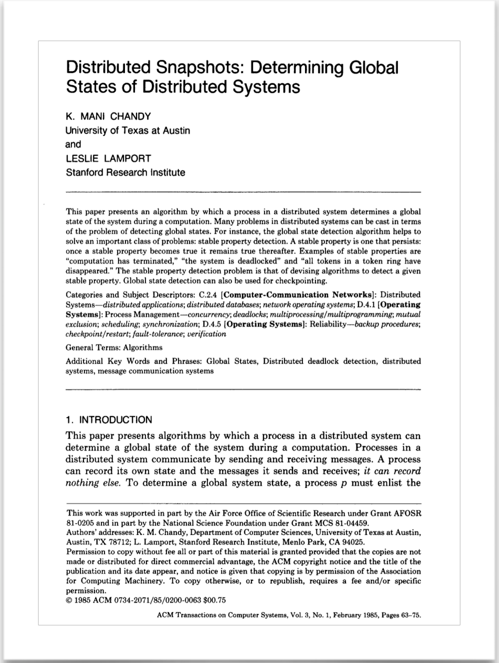
Process Graph

- Processes are connected by Input (I_p)/ Output channels (O_p)
- For each message m in I_p:
  - S'_p = process(m, S_p, O_p)
  - Updates local state $S_p = S'_p$
  - Adds output messages in O_p

CONSISTENT SNAPSHOTTING

- **Observation**: Impossible to get a direct snapshot without "freezing" all processes and channels
- **Goal**: Acquire a **consistent snapshot** instead

- **Consistent Snapshot**: Reflects a "valid" configuration of the running system (states and in-transit messages)
  - Valid Configuration ~ "**consistent cut**"



Distributed Snapshots: Determining Global States of Distributed Systems

K. MANI CHANDY
University of Texas at Austin
and
LESLIE LAMPORT
Stanford Research Institute

This paper presents an algorithm by which a process in a distributed system determines a global state of the system during a computation. Many problems in distributed systems can be cast in terms of the problem of detecting global states. For instance, the global state detection algorithm helps to solve an important class of problems: stable property detection. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared." The stable property detection problem is that of devising algorithms to detect a given stable property. Global state detection can also be used for checkpointing.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications; distributed databases; network operating systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency; deadlocks; multiprocessing/multiprogramming; mutual exclusion; scheduling; synchronization*; D.4.5 [**Operating Systems**]: Reliability—*backup procedures; checkpoint/restart; fault-tolerance; verification*

General Terms: Algorithms

Additional Key Words and Phrases: Global States, Distributed deadlock detection, distributed systems, message communication systems

1. INTRODUCTION

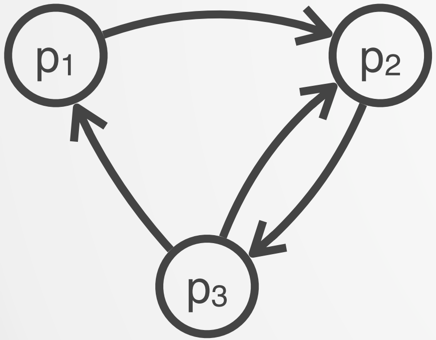
This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; *it can record nothing else*. To determine a global system state, a process p must enlist the

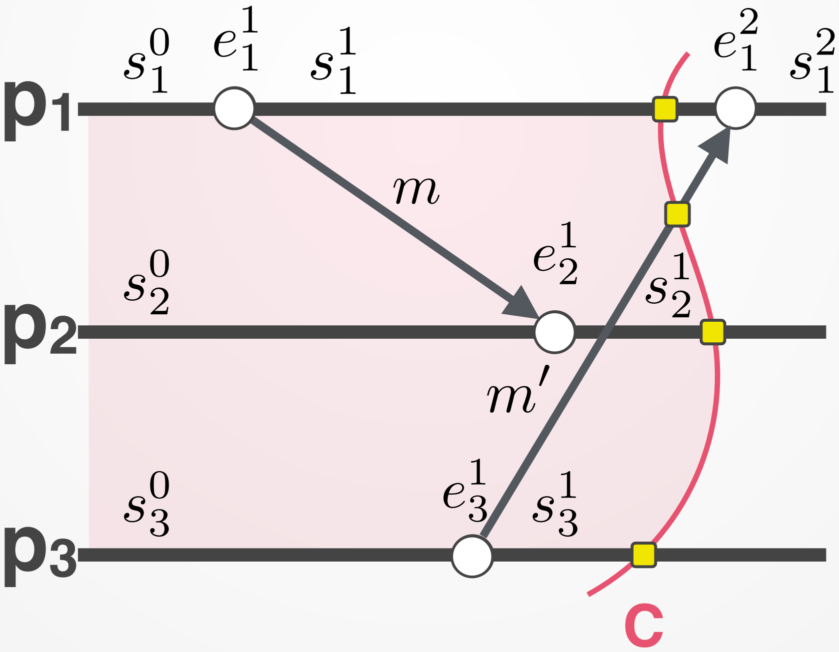
This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR 81-0205 and in part by the National Science Foundation under Grant MCS 81-04459.
Authors' addresses: K. M. Chandy, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. Lamport, Stanford Research Institute, Menlo Park, CA 94025.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1985 ACM 0734-2071/85/0200-0063 $00.75

ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985, Pages 63–75.

DISTRIBUTED CUTS

- A snapshot implements a cut **C** of an execution (prefix) and returns the system's corresponding states/configuration.



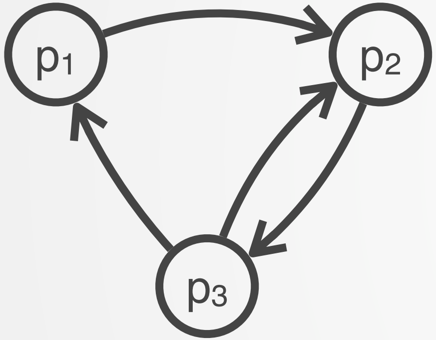


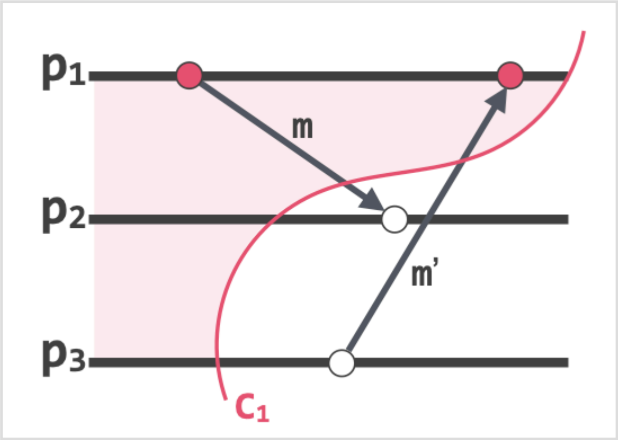
Snapshot of C

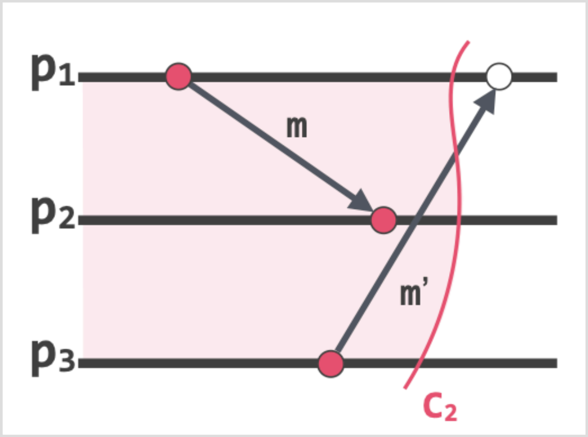
$\{s_1^1, s_2^1, s_3^1\}$
$\{m'\}$

CONSISTENT CUTS

- We are interested in consistent cuts - those that preserve **causality**







Inconsistent : Message m' was received but never sent in C_1

C_2 is Consistent

Consistent Snapshotting Specification

S_p: state of p
M_p: messages in I_p

Events

Request: ⟨snapshot⟩

Indication: ⟨record | p, [S_p,M_p]⟩

Properties:

S1: Termination, S2: Validity

Consistent Snapshotting Specification

S1: Termination: *Eventually every process records its state.*

S2: Validity: *All recorded states correspond to a consistent cut of the execution.*

THE CHANDY LAMPORT ALGORITHM

Assumptions:

- **FIFO Reliable Channels**
- **Single Initiating Process p_i**
- **Strong Connectivity**: There is a (channel) path from p_i to every other process in the system (always satisfied in strongly connected process graphs)

The Chandy Lamport Algorithm

Design Goal:

- **Obstruction-freedom**: The global-state-detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, this underlying computation. - Lamport, Chandy

Idea Intuition:

- Disseminate a special message ⊙ to mark events before and after the consistent cut.

THE ALGORITHM

Chandy-Lamport Consistent Snapshots

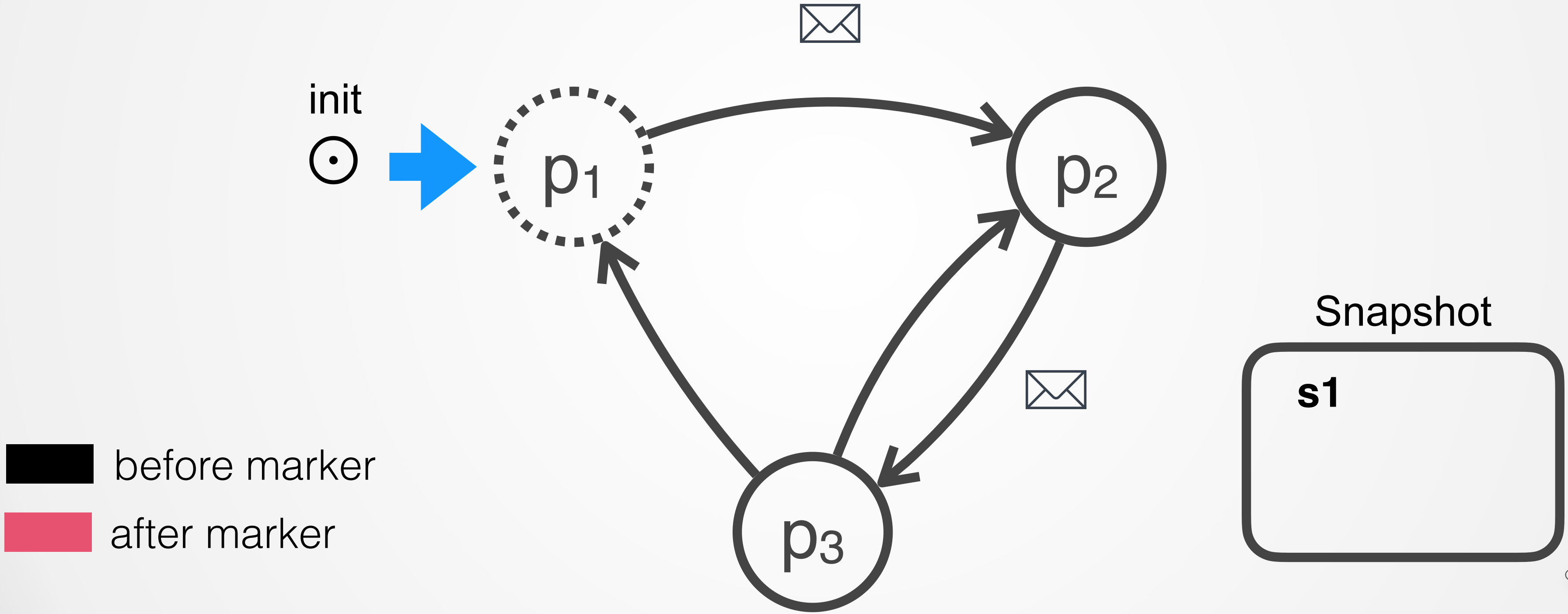
Implements: csnap, **Requires**: fiforc $(\mathbb{I}_p, \mathbb{O}_p)$

$(\mathbb{I}_p, \mathbb{O}_p) \leftarrow$ configured_channels;
$s_p \leftarrow \varnothing$; ▷ volatile local state
Recorded $\leftarrow \emptyset$; ▷ channels under logging
$s_p^* \leftarrow \emptyset$; $M_p \leftarrow \emptyset$; ▷ state in snapshot

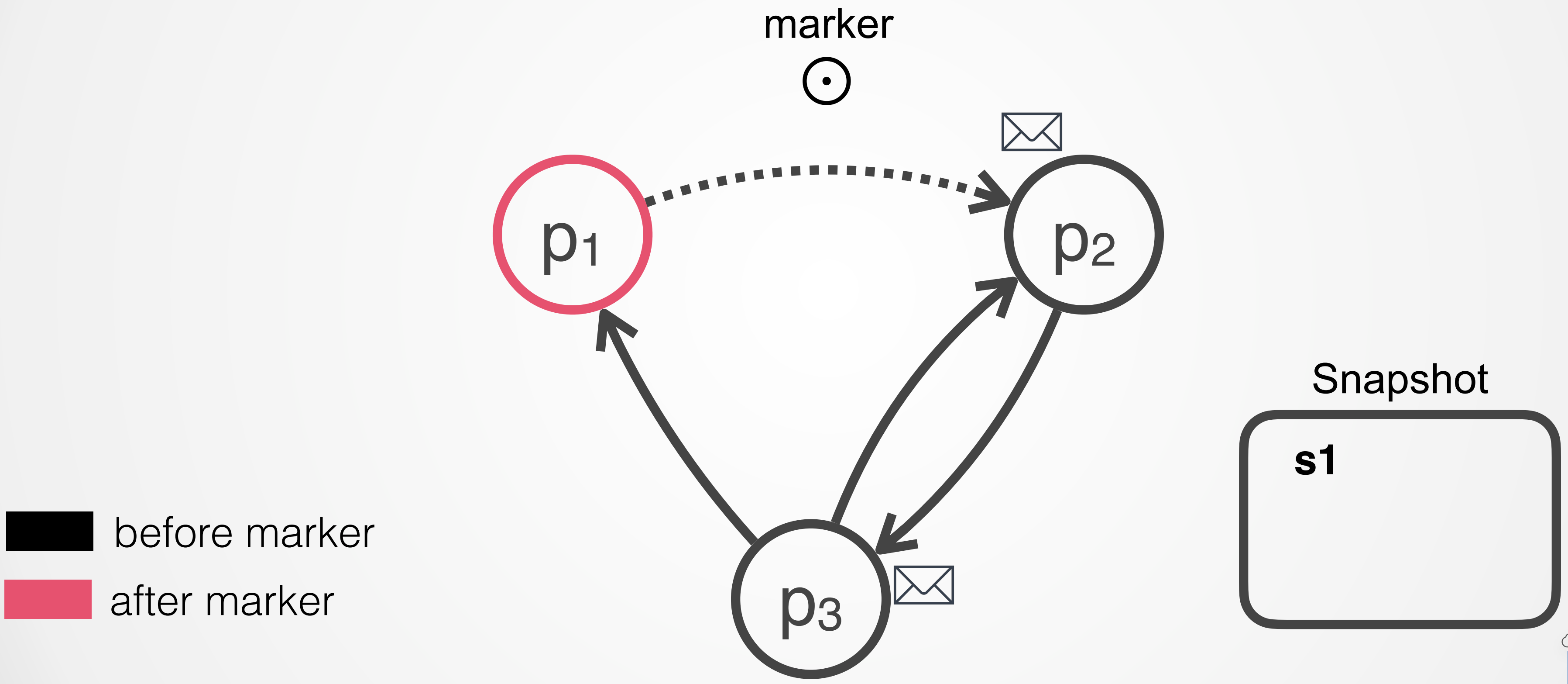
---

Upon $\langle rcvd, m\rangle$ *on* $c_{qp} \notin$ Recorded, $m \neq \odot$
  $s_p \leftarrow process(m, s_p, \mathbb{O}_p)$; ▷ regular process logic
Upon $\langle rcvd, m\rangle$ *on* $c_{qp} \in$ Recorded, $m \neq \odot$
  $M_p \leftarrow M_p \cup \{m\}$; ▷ record in-transit message
  $s_p \leftarrow process(m, s_p, \mathbb{O}_p)$;
Upon $\langle rcvd, \odot\rangle$ *on* $c_{qp} \in \mathbb{I}_p$
  if $s_p^* = empty$ **then**
    $startRecording()$;
  Recorded = Recorded $-\{c_{qp}\}$;
  if Recorded $= \emptyset$ **then**
    $csnap \rightarrow \langle record|self, s_p^*, M_p\rangle$;
Upon $\langle snapshot\rangle$ *on* $csnap$
  $startRecording()$;
  if Recorded = $\emptyset$ **then**
    $csnap \rightarrow \langle record|self, s_p, \emptyset\rangle$;
Fun *startRecording()*
  $s_p^* \leftarrow s_p$; ▷ record local state
  foreach out $\in \mathbb{O}_p$ **do**
    $out \rightarrow \langle send, \odot\rangle$;
  Recorded $\leftarrow \mathbb{I}_p$

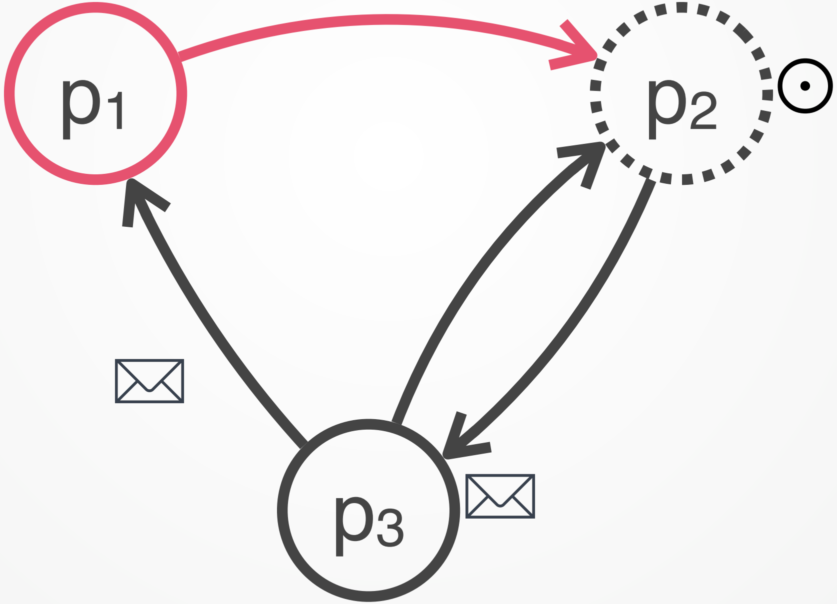
EXAMPLE EXECUTION



EXAMPLE EXECUTION



EXAMPLE EXECUTION



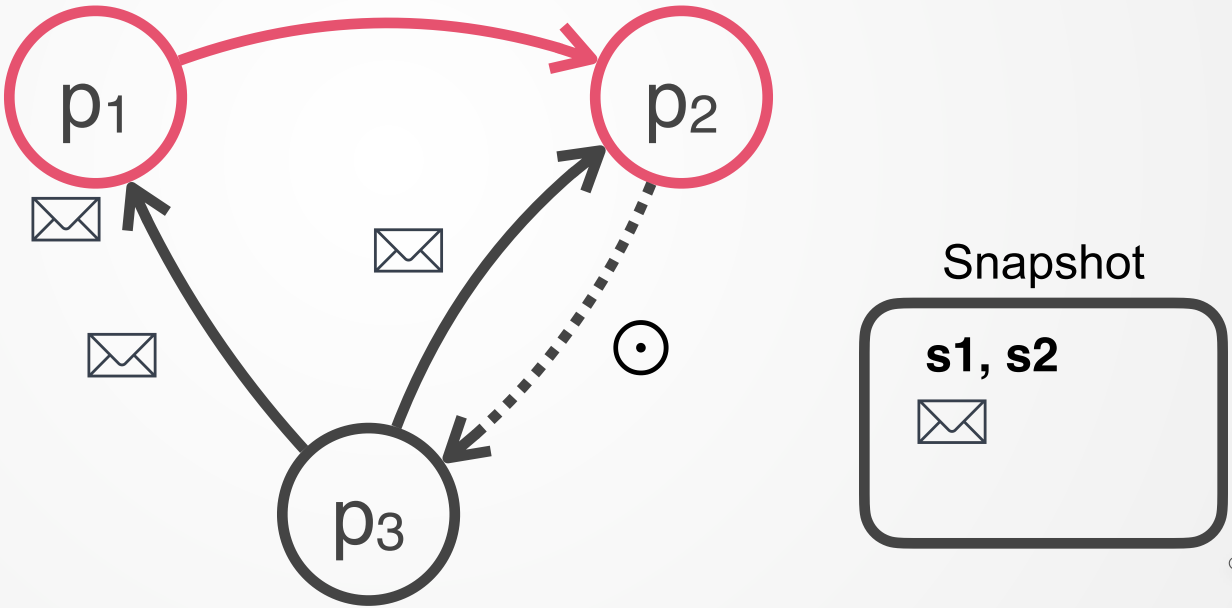
before marker

after marker

Snapshot

s1, s2

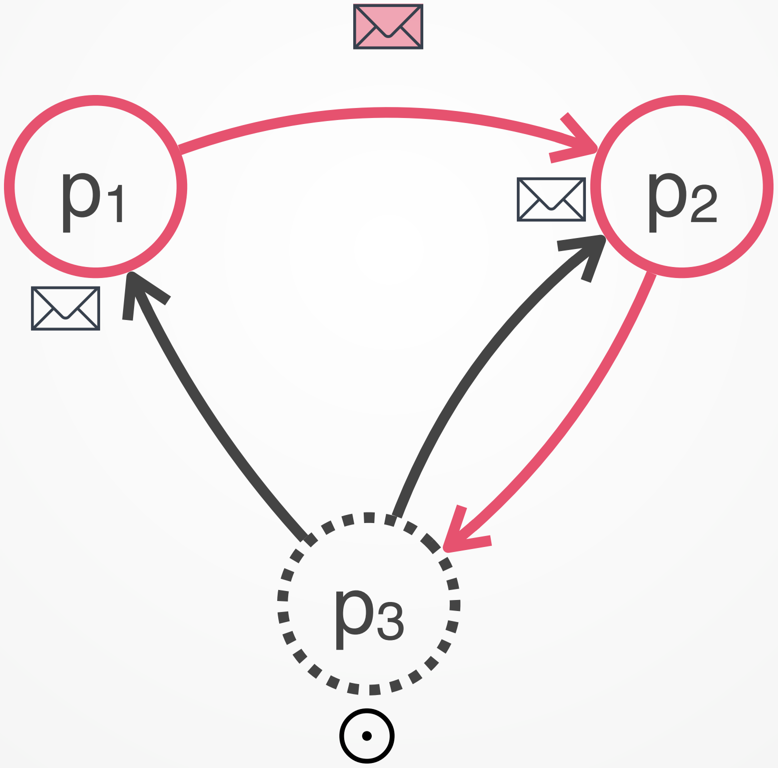
Example Execution

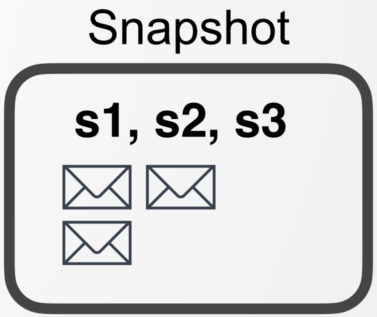


before marker

after marker

Example Execution

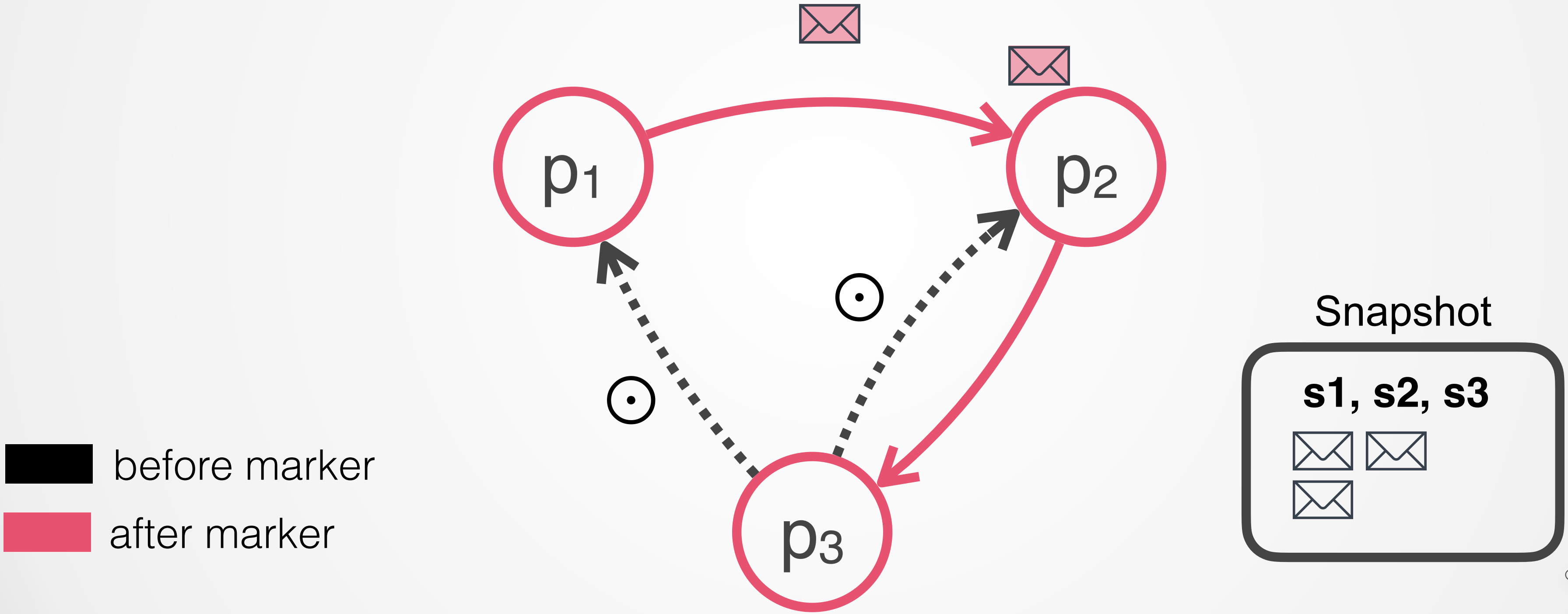




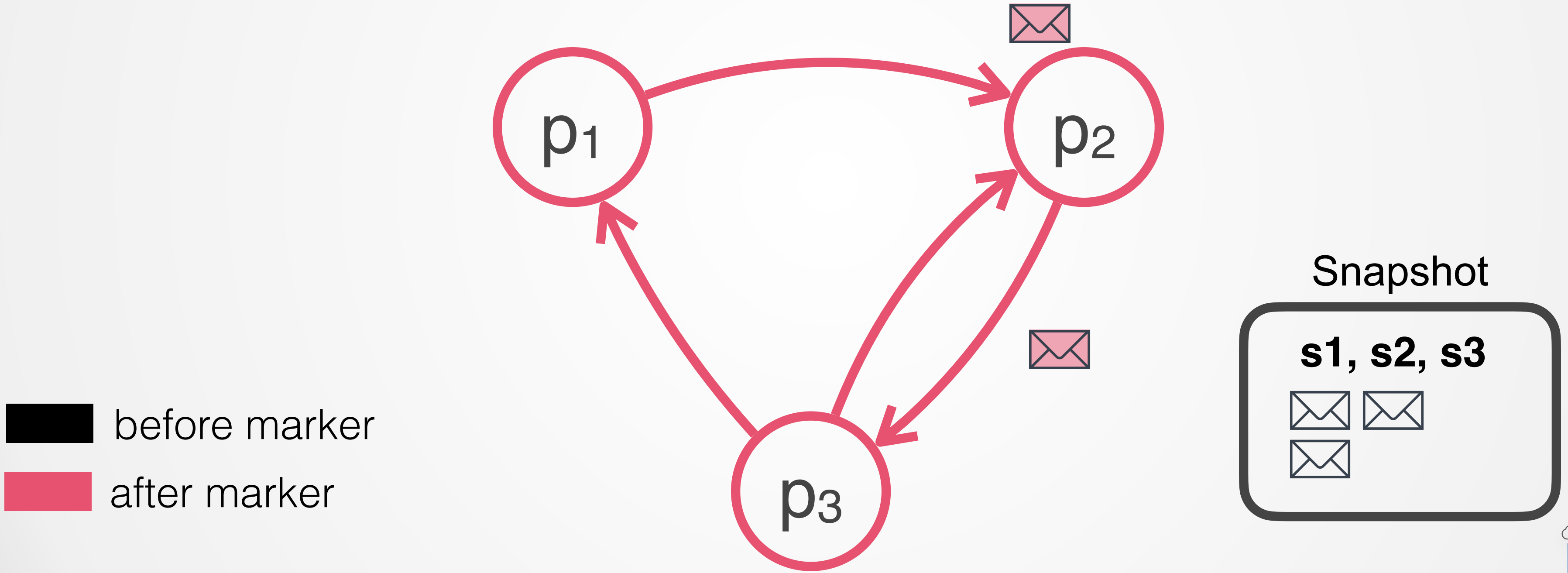
before marker

after marker

EXAMPLE EXECUTION



EXAMPLE EXECUTION



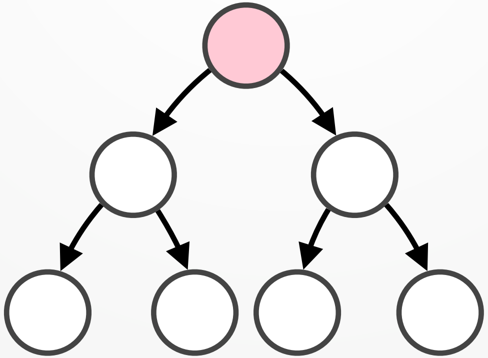
PROOF SKETCH

- **Validity**

    - **Marker** sent between pi and pj separates pre- and post-snapshot events (through FIFO channel delivery)

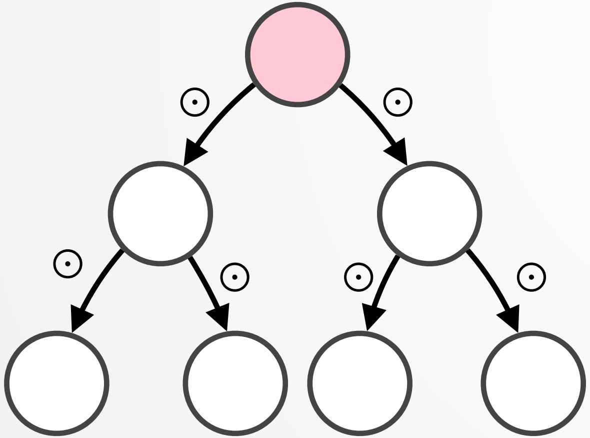
    - Validity applies to the transitive closure of reachable processes (through induction)

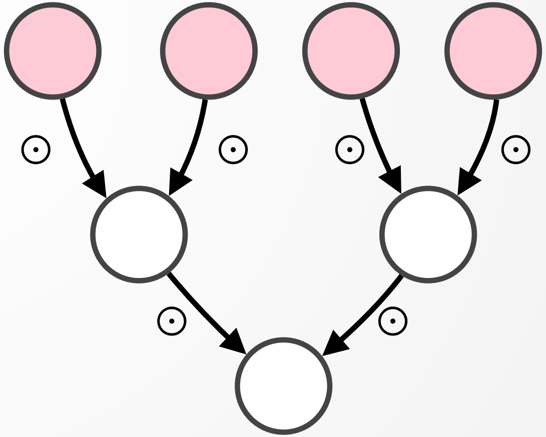
- **Termination** is satisfied **if** initiator can **reach** all tasks.



GENERALIZATION

- **Termination** is still satisfied **if** the protocol is initiated by a **set** of processes that can reach all tasks. (No modifications)



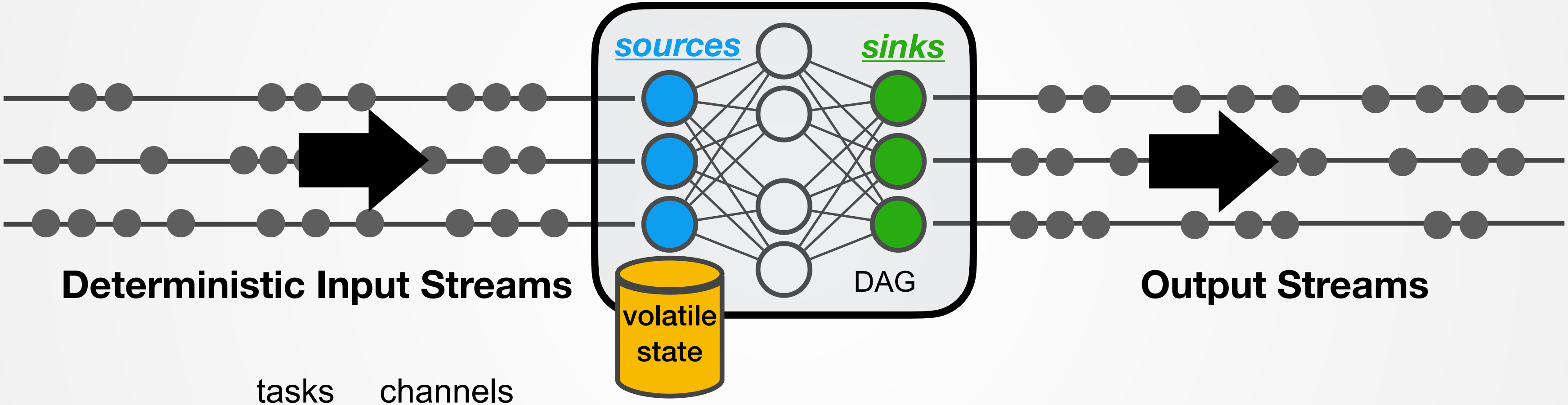


Epoch Snapshotting

DATA PROCESSING SNAPSHOTS

- **Snapshotting** protocols can be used to make production-grade data processing systems reliable.
- Examples: Google Dataflow, Flink, Tensorflow, Spark, IBM Streams, Storm, Apex etc.
- **Use Case:** The Apache Flink data processing system

STREAM PROCESS GRAPHS

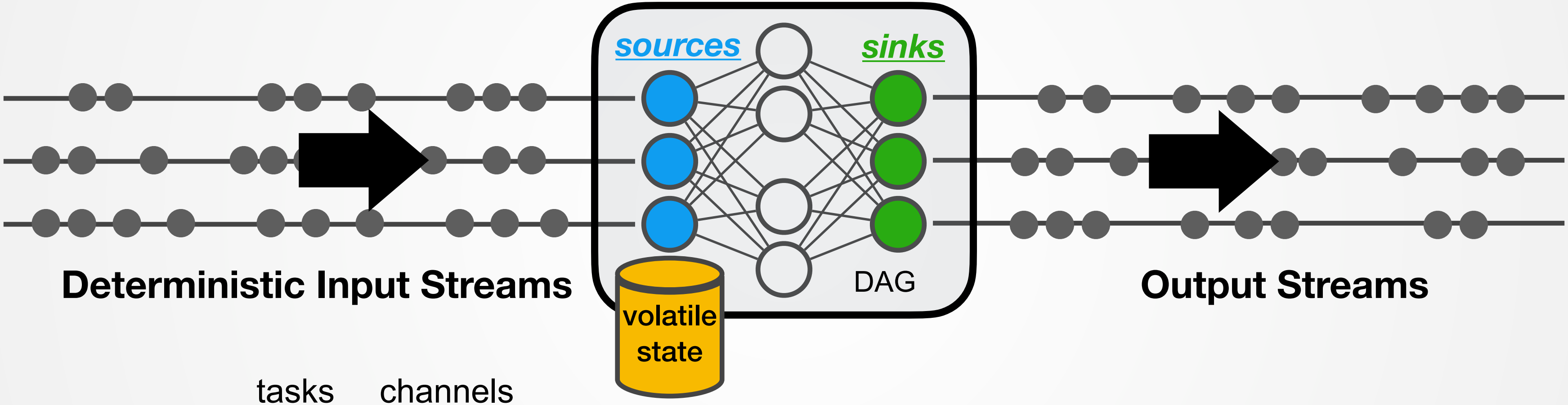


tasks channels

System : $\{\Pi, \mathbb{E}\}$

System Execution : $\ldots \rightarrow \{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\} \rightarrow \ldots$

Stream Process Graphs



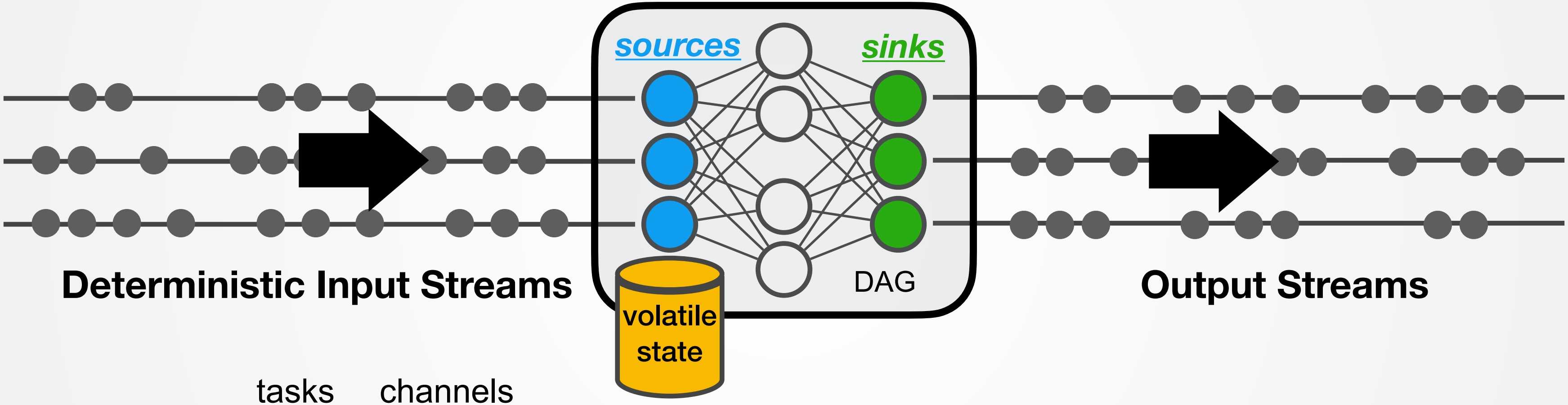
tasks channels

System : $\{\Pi, \mathbb{E}\}$

Task Actions

System Execution : $\ldots \boxed{\rightarrow} \{\Pi_*, M\} \boxed{\rightarrow} \{\Pi'_*, M'\} \boxed{\rightarrow} \ldots$

Stream Process Graphs



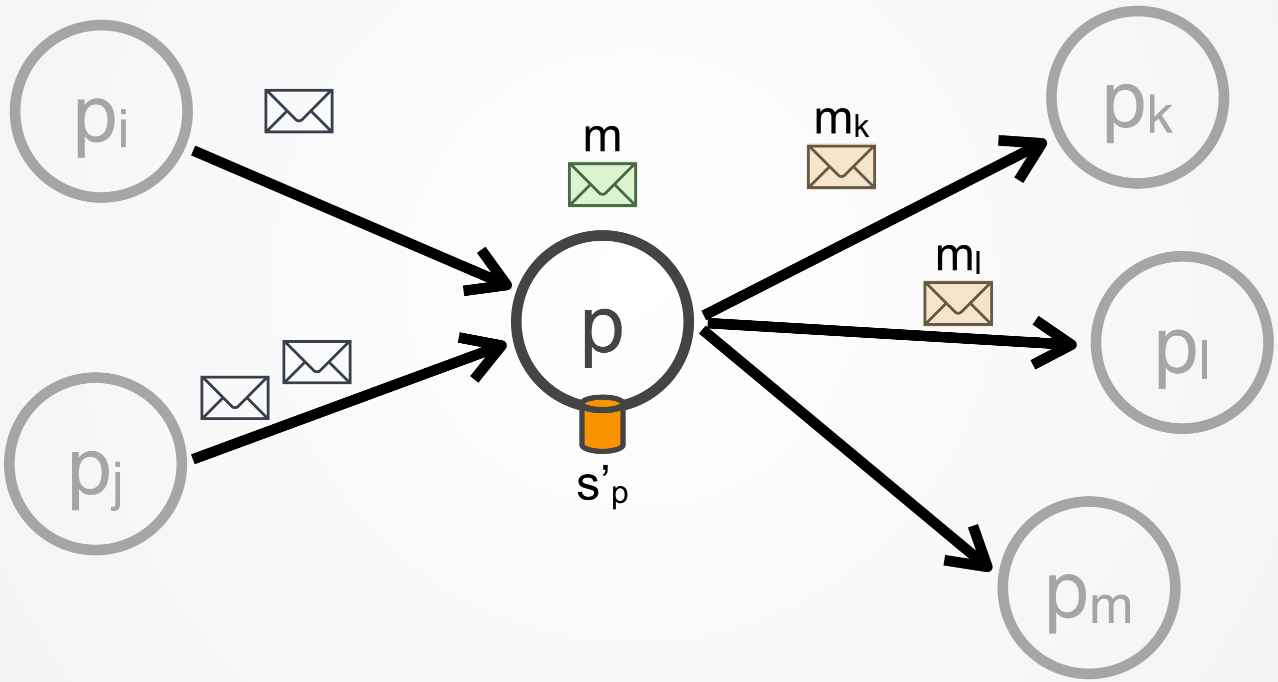
tasks, channels

System : $\{\Pi, \mathbb{E}\}$

System Configurations (states, messages in-transit)

System Execution : $\ldots \rightarrow \{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\} \rightarrow \ldots$

Fault Tolerance

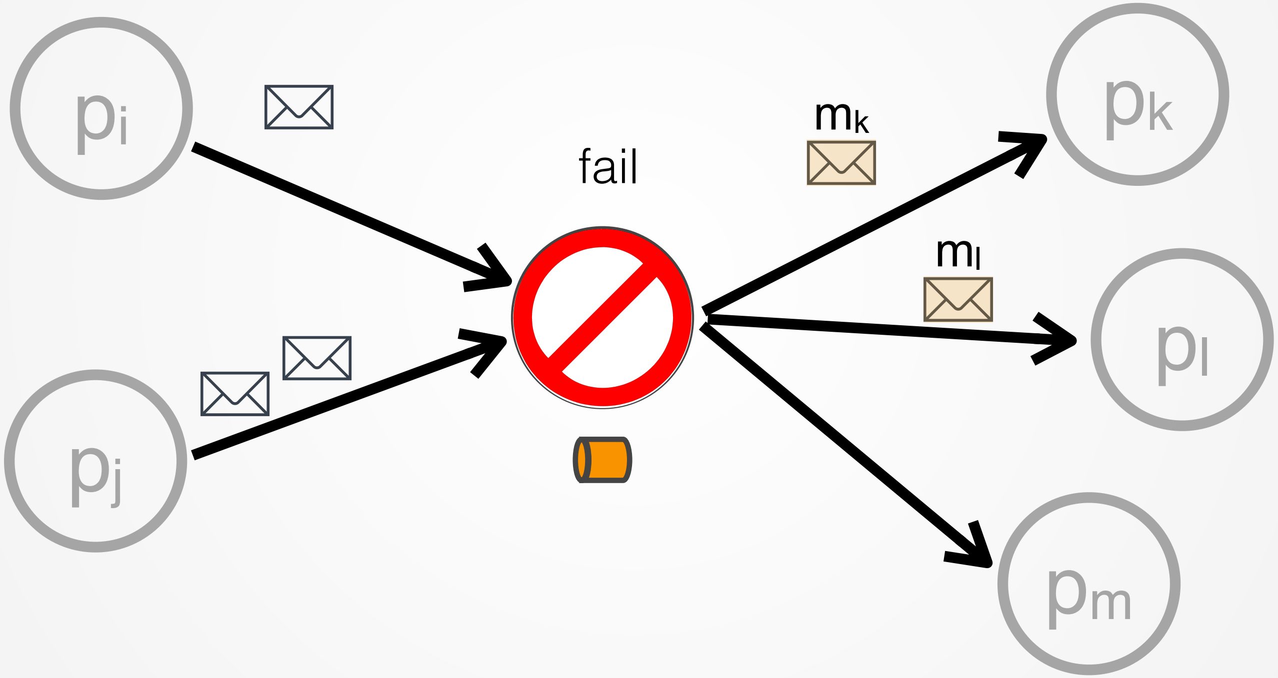


ID2203

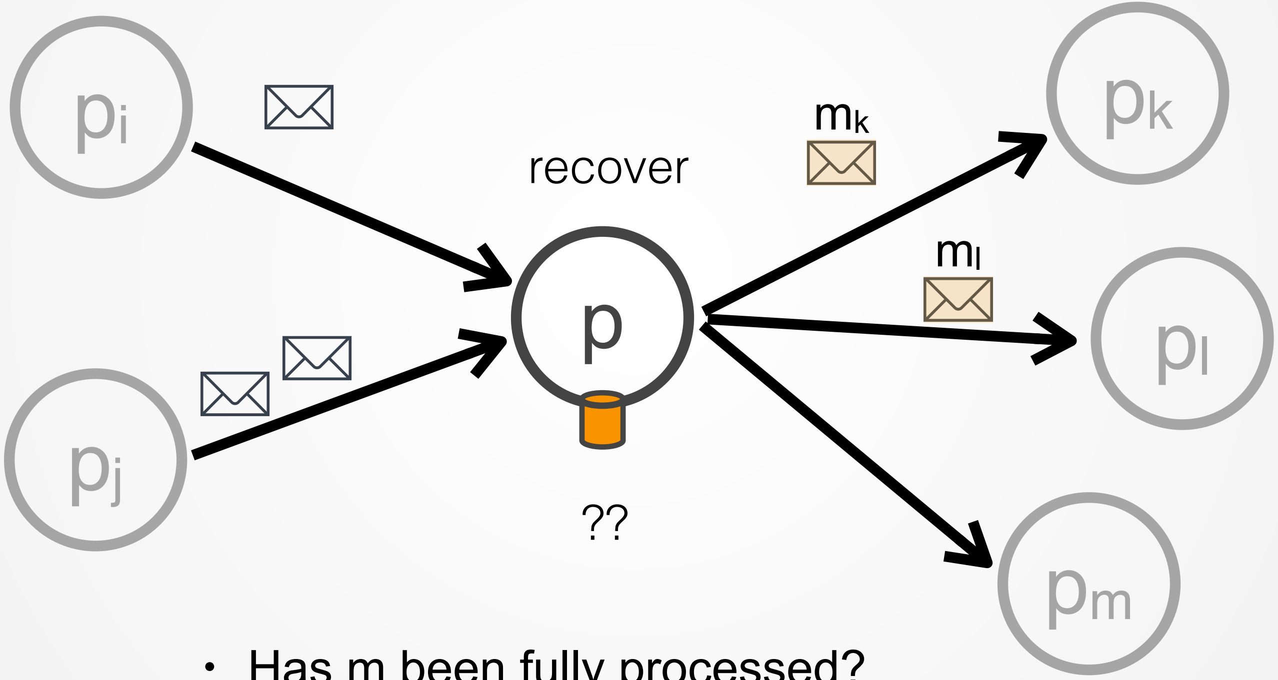
KTH VETENSKAP OCH KONST

KTH-2021

Fault Tolerance



Fault Tolerance



- Has m been fully processed?
- Have m_k and m_l been delivered?

Reliable Stream Processing

- Previous approaches* typically adopt a fail recovery model to amend individual task execution and reproduce computations that were possibly lost
  - Complex Workarounds (e.g., duplicate elimination, input logging, acks)
  - Strong Assumptions (idempotent operations, key vs task level causal order)
  - External State Management (transactional external commits per action)

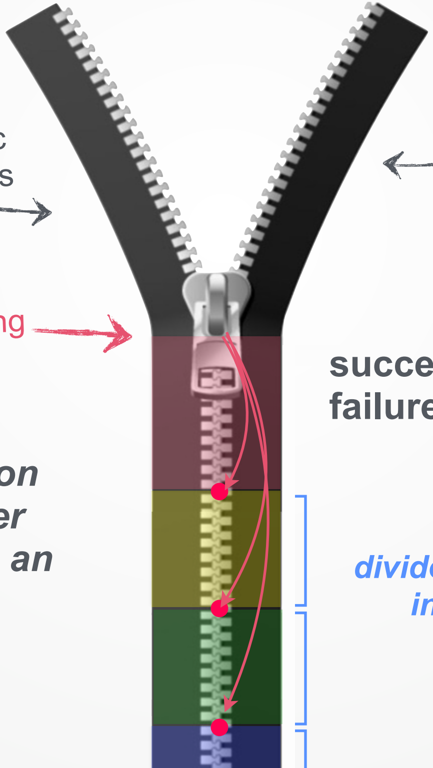
*MillWheel: Fault- tolerant stream processing at internet scale," in VLDB, 2013.
Integrating scale out and fault tolerance in stream processing using operator state management. in SIGMOD 2013
Fault-tolerance and high availability in data stream management systems. in Encyclopedia of Database Systems 2009
Fault-tolerance in the Borealis distributed stream processing system, in SIGMOD 2005

Fault Tolerance is not Enough

- Are output and states always correct?
- Can we reconfigure the system without losing computation?
- Can applications migrate without loss?
- Is external state access isolation possible?

We need a system-wide coarse-grained commit mechanism.

EPOCH-BASED STREAM EXECUTION
THE INTUITION

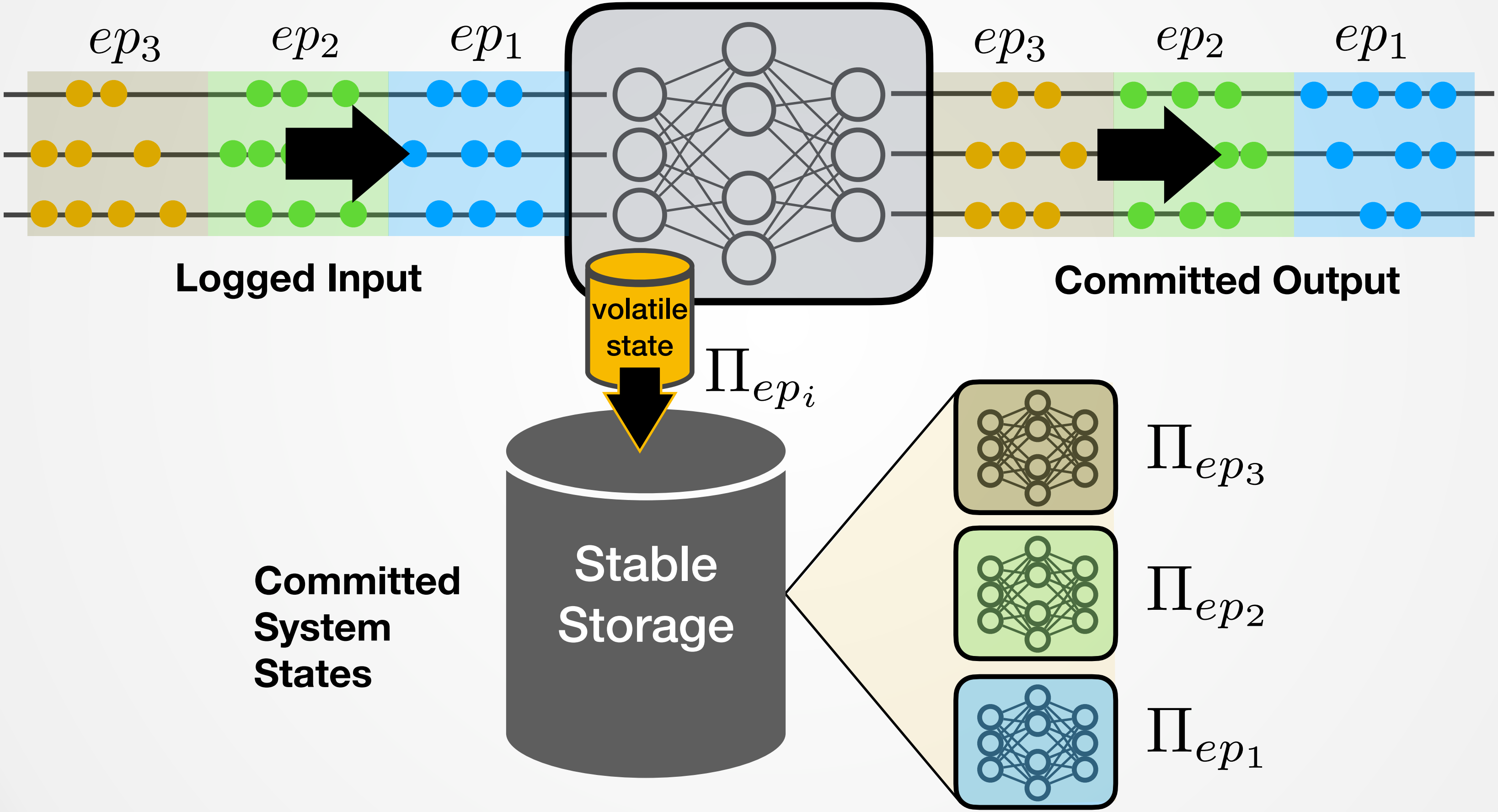


success: commit system configuration
failure: abort and start from previous epoch

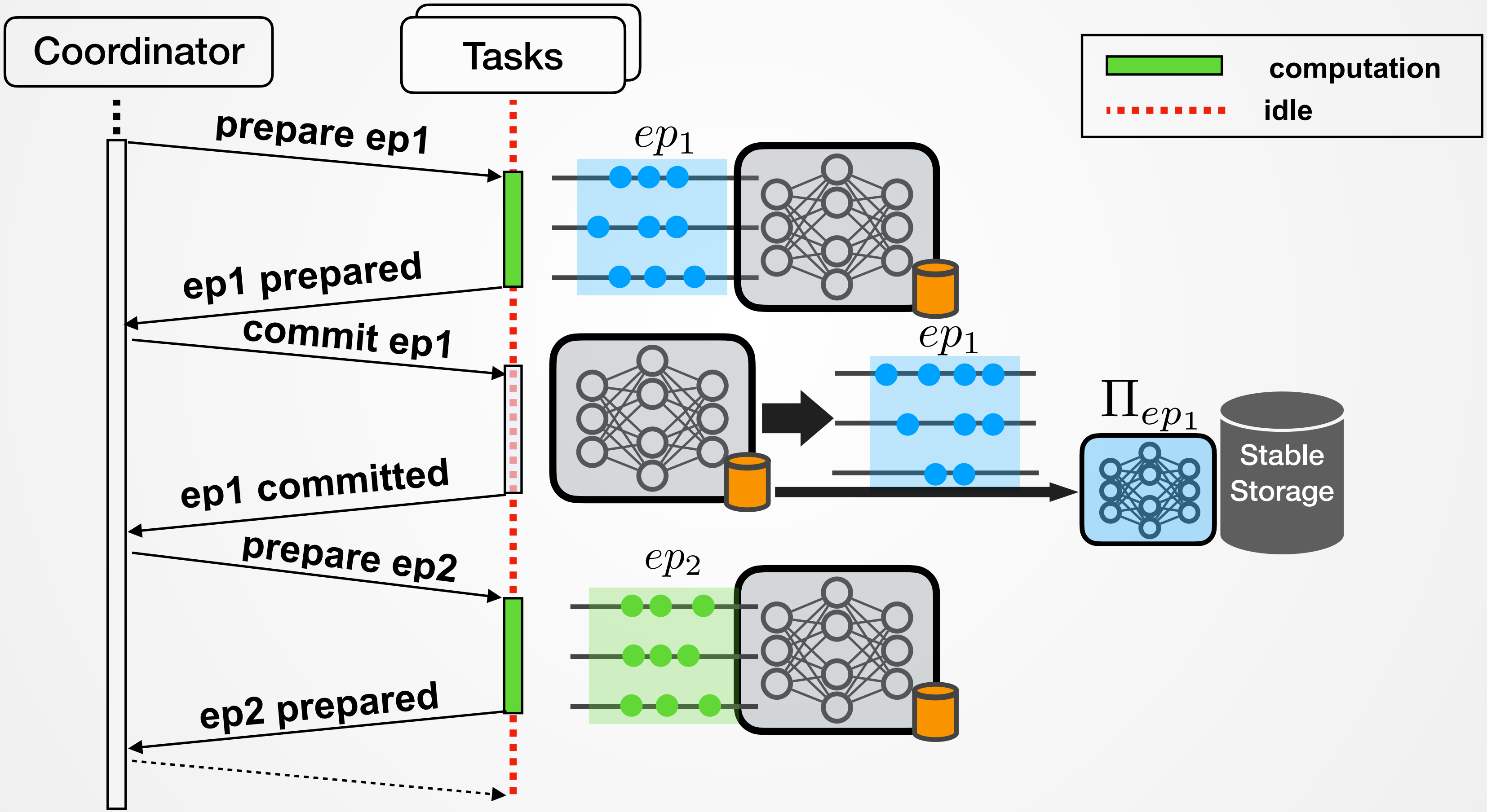
- ***system configuration (states) after completing an epoch***

divide computation into epochs

EPOCH-BASED STREAM EXECUTION



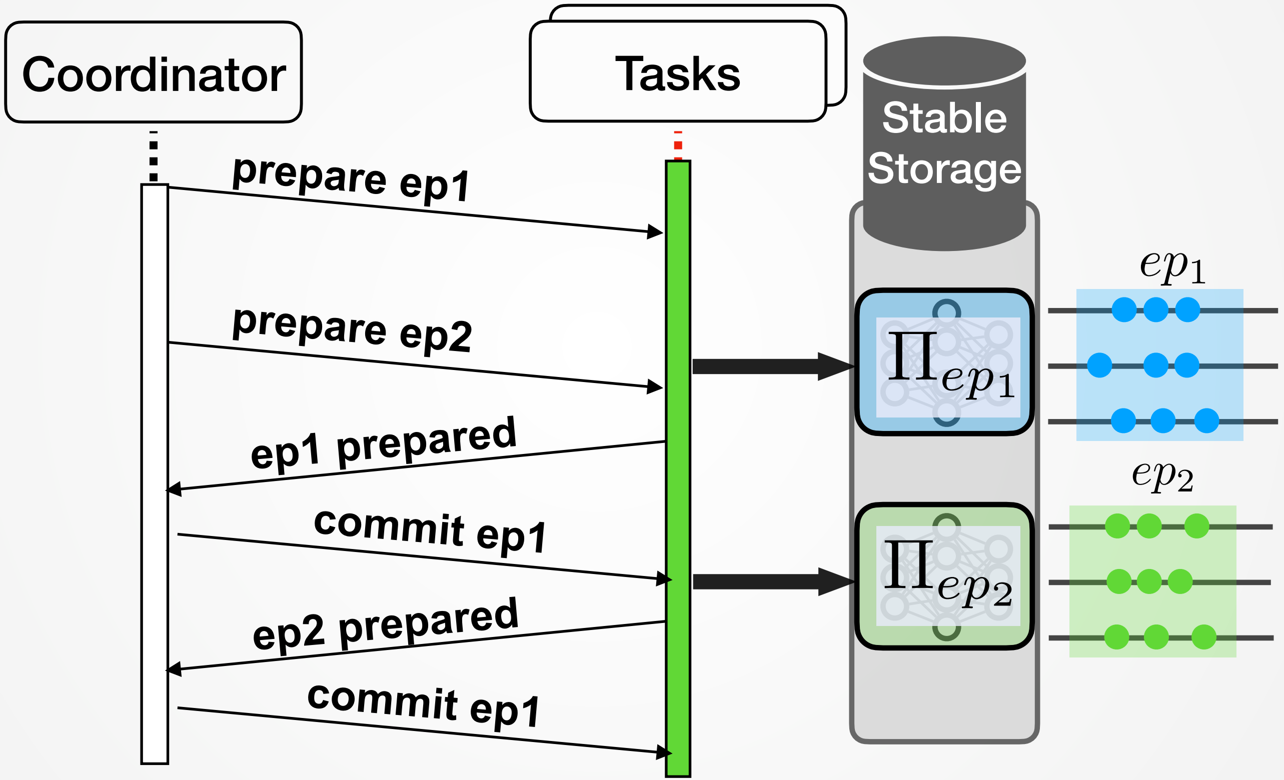
Synchronous Epoch Commits



Synchronous Epoch Commit

- Suitable for short-lived, stateless task execution
- **Problem:** Unnecessary high **latency** in long-running task execution
- **Cause: Blocking synchronisation (idle time)** - coordination & epoch scheduling.

Asynchronous Epoch Commits

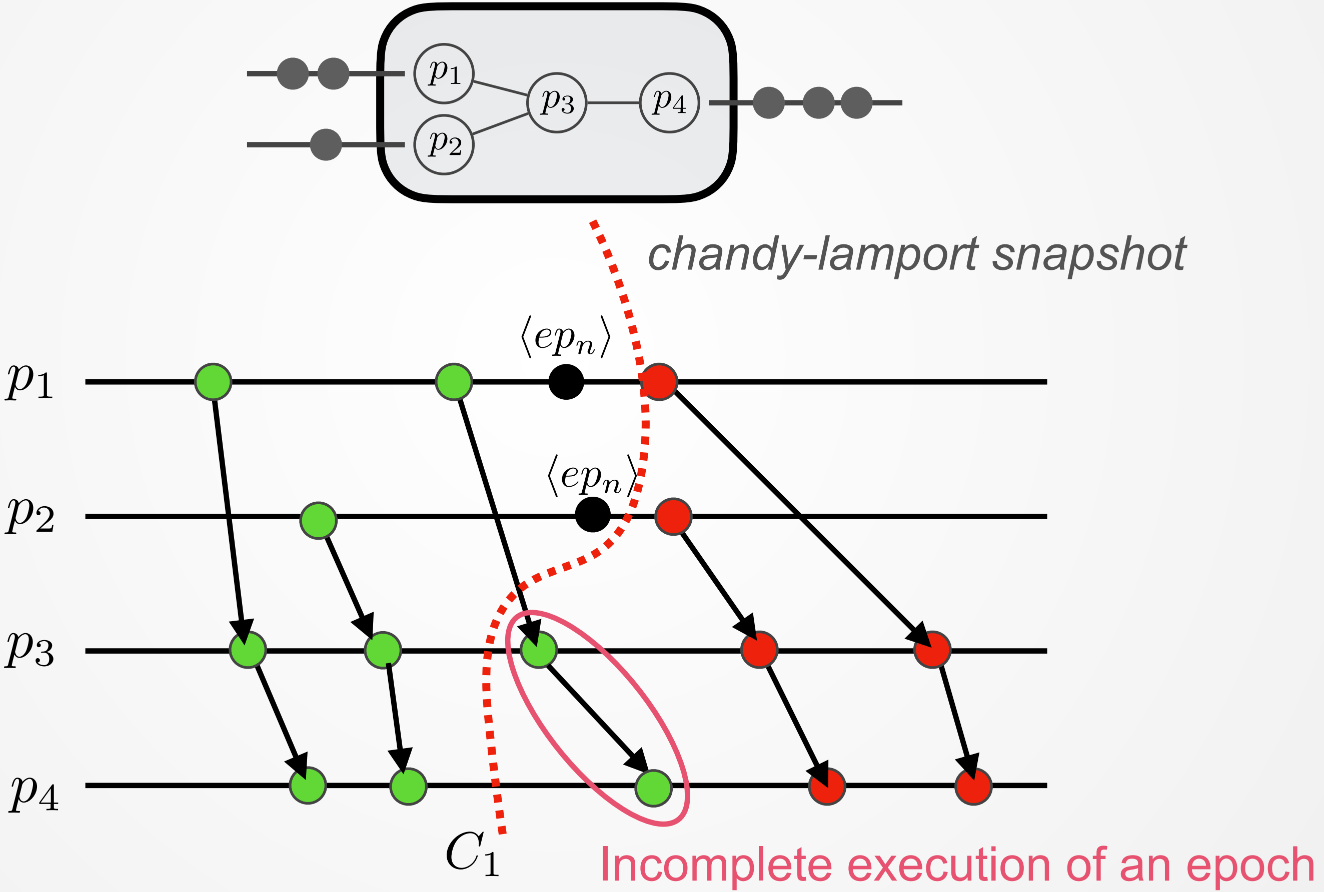


How? Using Snapshots

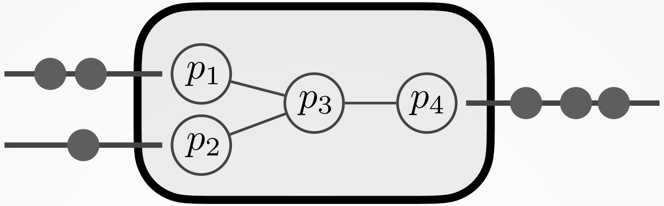
Epoch Snapshotting

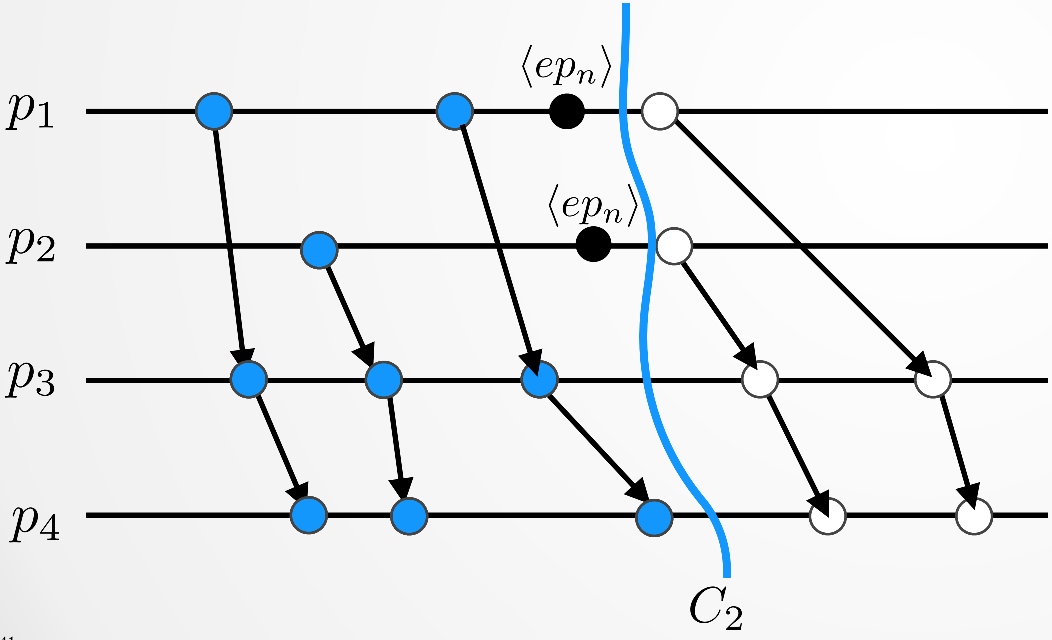
- Assumptions:
  - DAG of tasks
  - **Epoch change** events triggered on each **source** task (⟨ep1⟩,⟨ep2⟩,…)
    - Issued by master or generated periodically
- We want to snapshot stream process graphs after the **complete computation** of an epoch.

VALIDITY IS NOT ENOUGH



Epoch Cuts





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 SNAPSHOTTING PROPERTIES

Termination (liveness):

A **full** system configuration is eventually captured per epoch

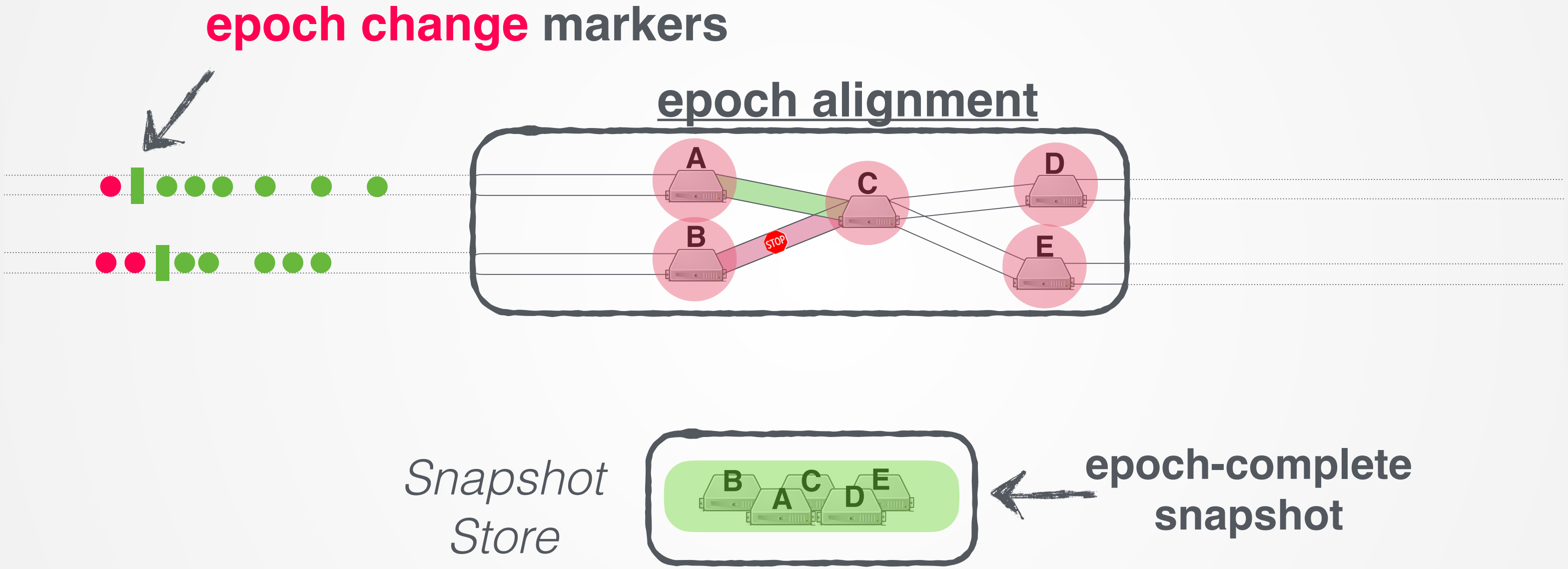
Validity (safety):

Obtain a **valid** system configuration (consistent cut)

Epoch-Completeness (safety):

Obtain an **epoch-complete** system configuration

THE ALGORITHM



The Epoch Snapshotting Algorithm

Epoch-Based Snapshots (Sources)

Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($\mathbb{I}_p, \mathbb{O}_p$)
Algorithm:

$\mathbb{O}_p \leftarrow$ configured_channels;
$s_p \leftarrow \varnothing$;

```
/* Source Task Logic
```

Upon $\langle rcvd, m \rangle$
  $(s_p) \leftarrow process(s_p, m, \mathbb{O}_p)$;

Upon $\langle ep|n \rangle$
  $esnap \rightarrow \langle record|self, n, s_p \rangle$;
  foreach $out \in \mathbb{O}_p$ **do**
    $out \rightarrow \langle send, \odot_n \rangle$;

Epoch-Based Snapshots (Regular Tasks)

Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($\mathbb{I}_p, \mathbb{O}_p$)
Algorithm:

$(\mathbb{I}_p, \mathbb{O}_p) \leftarrow$ configured_channels;
$Enabled \leftarrow \mathbb{I}_p$;
$s_p \leftarrow \varnothing$;

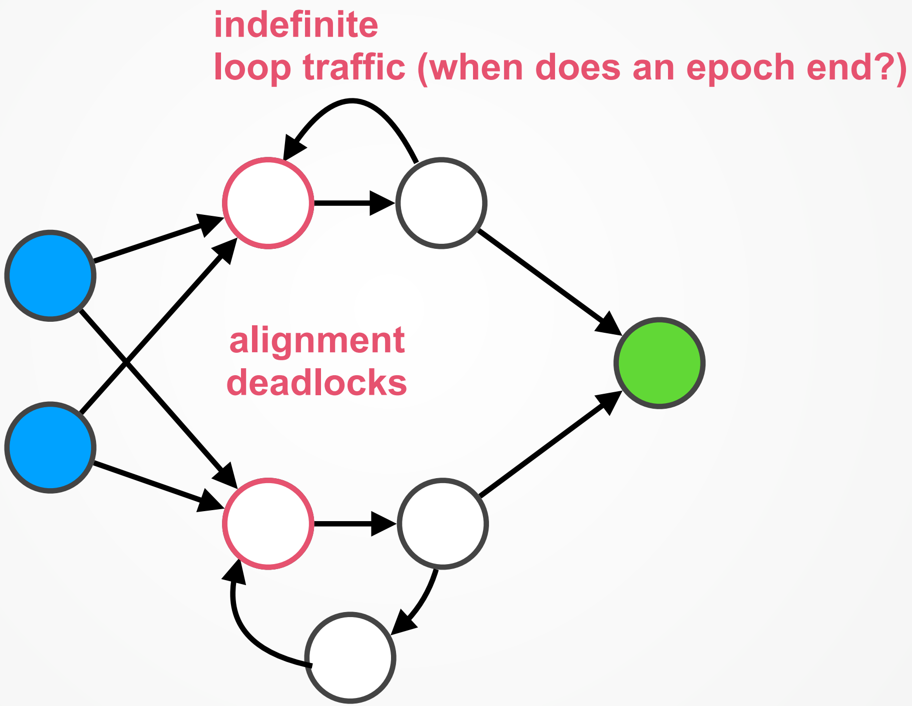
```
/* Common Task Logic
```

Upon $\langle rcvd, m \rangle$ *on* $c \in Enabled$
  $s_p \leftarrow process(s_p, m, \mathbb{O}_p)$;

Upon $\langle rcvd, \odot_n \rangle$ *on* $c \in Enabled$
  $esnap \rightarrow \langle record|self, n, s_p \rangle$;
  $Enabled \leftarrow Enabled/\{c\}$;
  if $Enabled = \emptyset$ **then**
    foreach $out \in \mathbb{O}_p$ **do**
      $out \rightarrow \langle send, \odot_n \rangle$;
    $Enabled \leftarrow \mathbb{I}_p$;

Carbone, Haridi, Ewen et al 2015

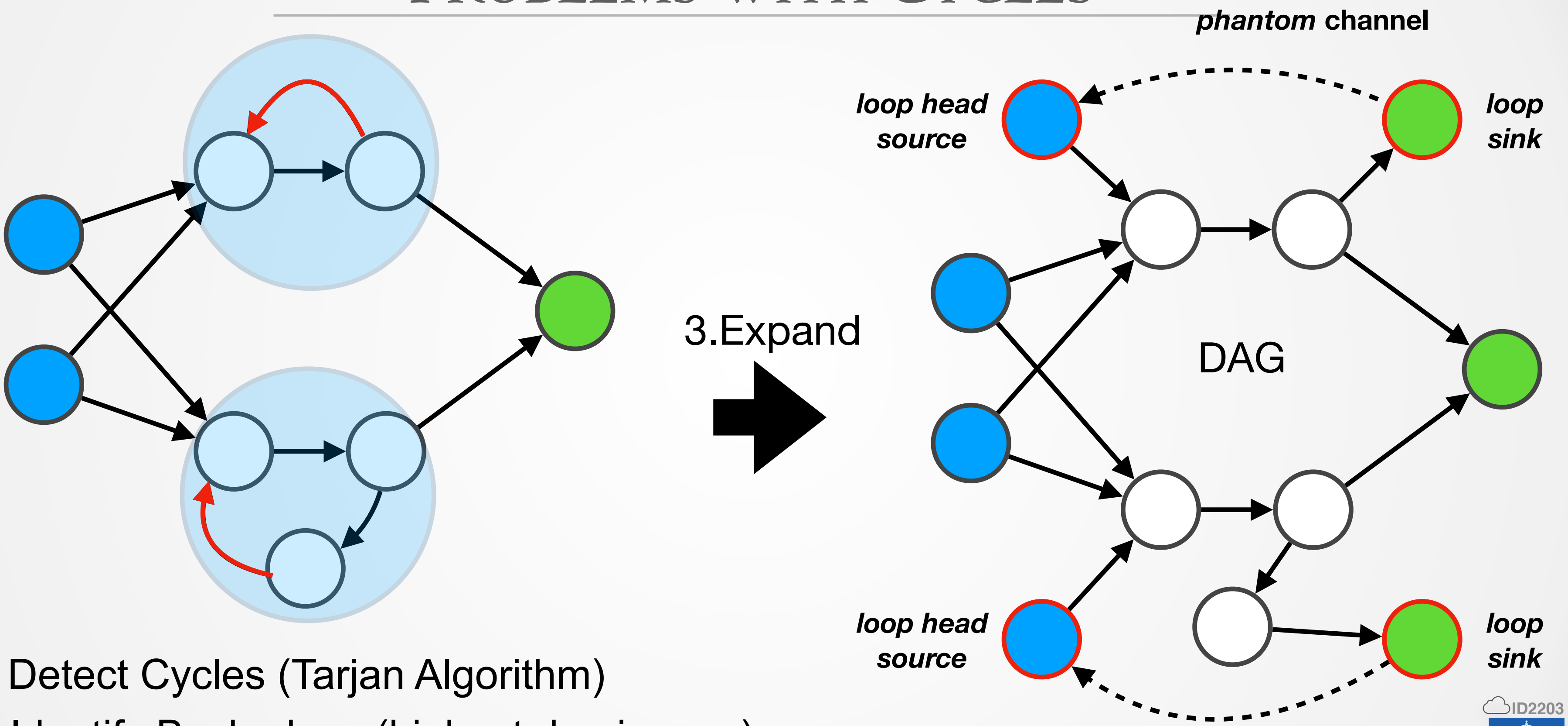
PROBLEMS WITH CYCLES



ID2203

KTH-2021

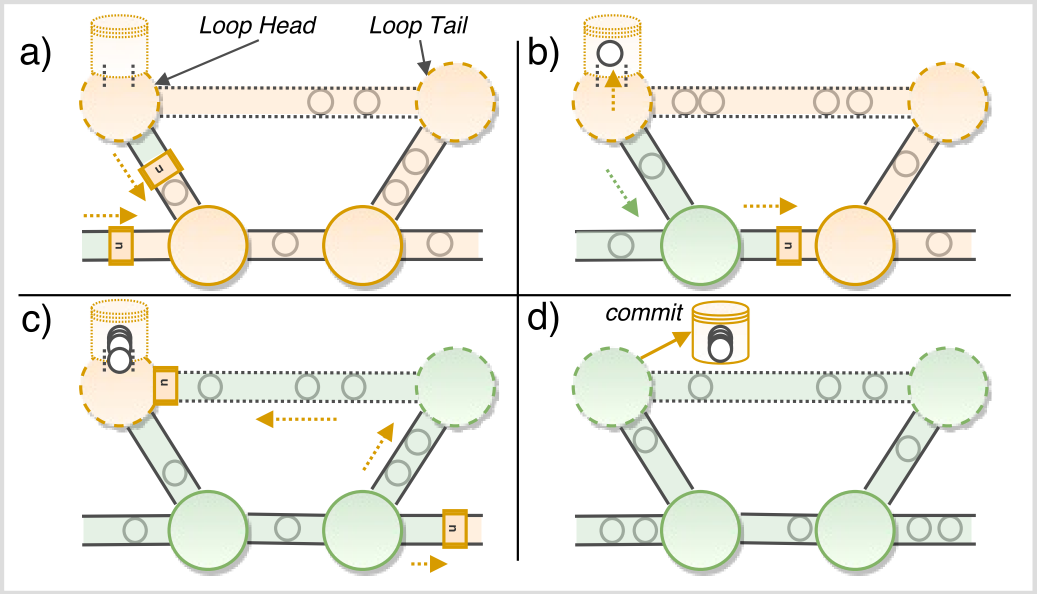
PROBLEMS WITH CYCLES



1. Detect Cycles (Tarjan Algorithm)
2. Identify Backedges (highest dominance)

PROBLEMS WITH CYCLES

Loop Sources receive epoch change events (like regular sources).



Snapshot Variant on loop heads

Log in-transit records per loop until marker arrives back.
(~Chandy-Lamport)

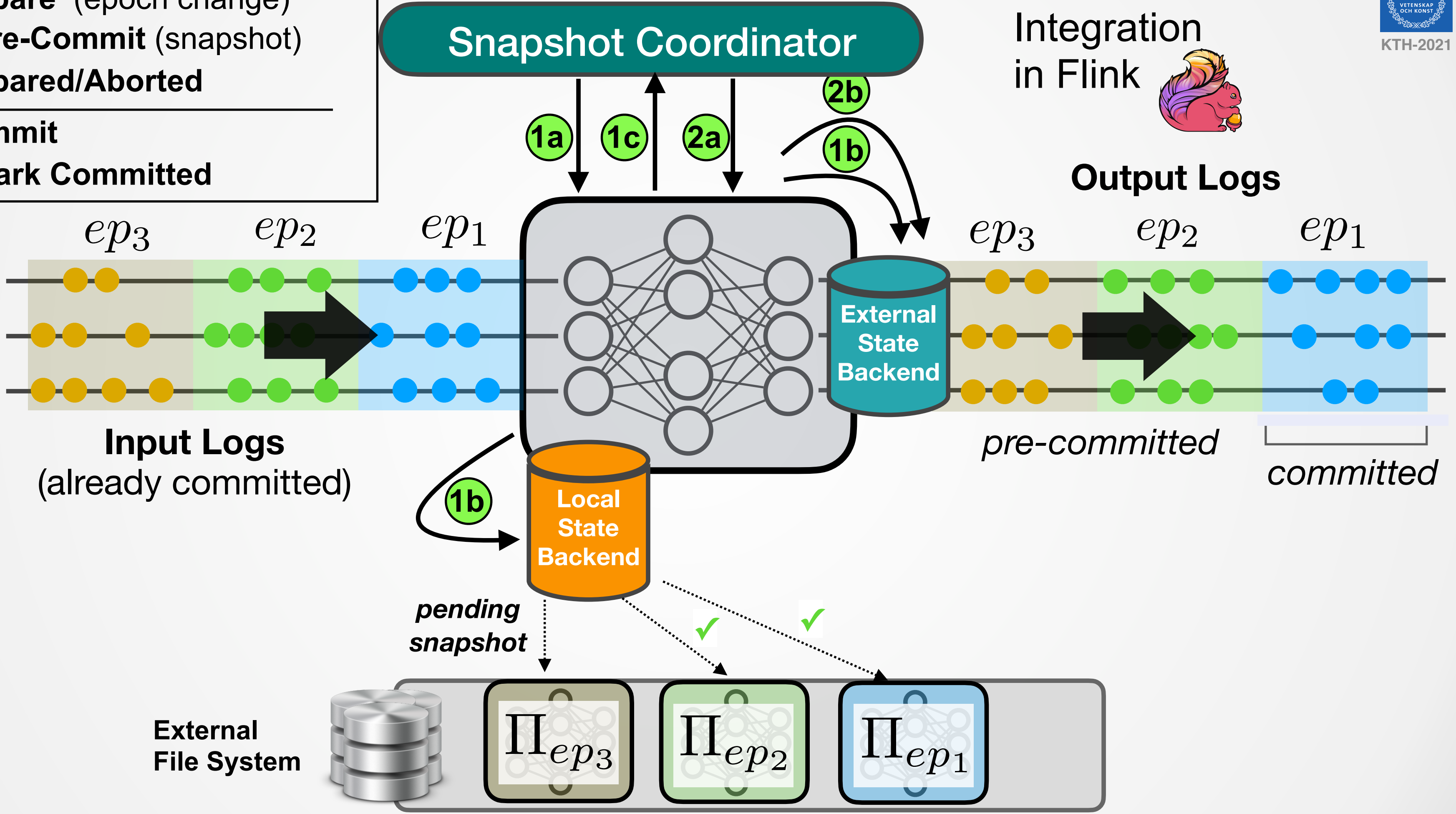
ID2203
KTH-2021

End to End Integration in Flink

The 2-Phase Commit Protocol

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

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



BEYOND ID2203

- Our Distributed Systems Research Group
  - https://dcatkth.github.io/
- The Continuous Deep Analytics Team
  - https://cda-group.github.io/
- Contact us for MSc topics and internships (RISE, KTH) in
  - Distributed Algorithms
  - Distributed Data Management (Graphs, ML, Relational)
  - Data Storage Optimisation for Data Analytics