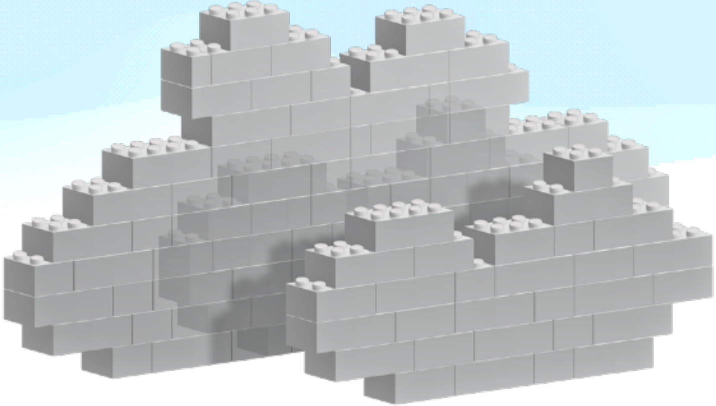
ADVANCED COURSE

Distributed Systems



Distributed Data Management

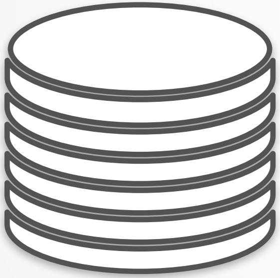
Paris Carbone

Course Topics

- Intro to Distributed Systems
- Fundamental Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Distributed Shared Memory-CRDTs
- Consensus (Paxos)
- Replicated State Machines (OmniPaxos, Raft, Zab etc.)
- Time Abstractions and Interval Clocks (Spanner etc.)
- Consistent Snapshotting (Stream Data Management)
- Distributed ACID Transactions (Cloud DBs)

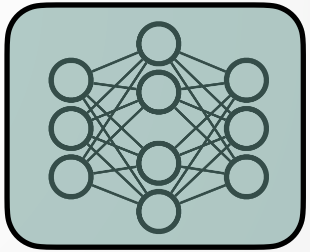
WHY DO WE NEED DISTRIBUTED SYSTEMS AGAIN?

- The majority of applications and problems come from the domain of scalable data management
- Goals: Make data systems more **scalable** and **reliable**



DBs and Data Storage Systems

\+



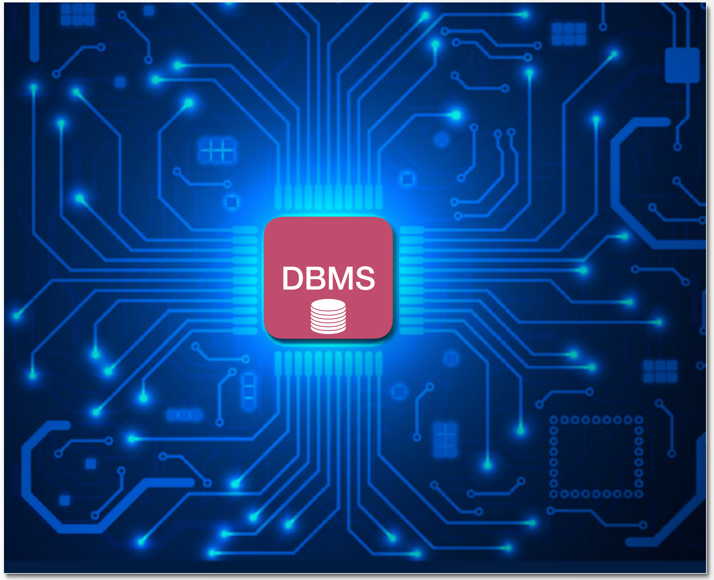
Data Processing Systems

Distributed ACID Databases

The Concurrent Power of Databases

Why DBMSs are so trusted:

- Concurrent Accessibility / scalability
  - >100k-million transactions per second per dbms process.
- Consistent recovery from failures.
- Isolation Guarantees



Also…your **bank accounts** (active, savings, investments) , **ATM interactions**, **online banking**, your **medical data records etc.** are handled by the same databases that handle other million users.

ANATOMY OF A TRANSACTION

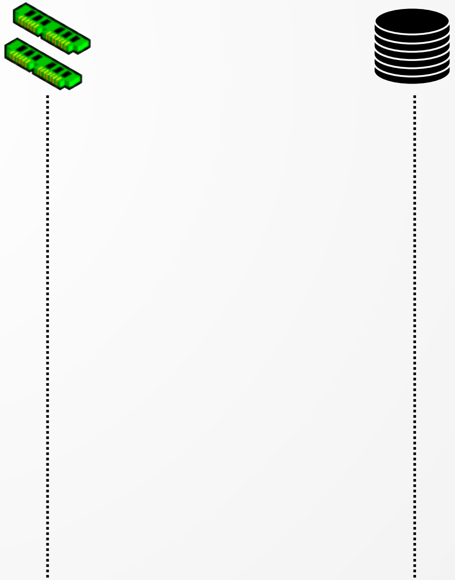
Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

	Balance
X	1000
Q	100



ANATOMY OF A TRANSACTION

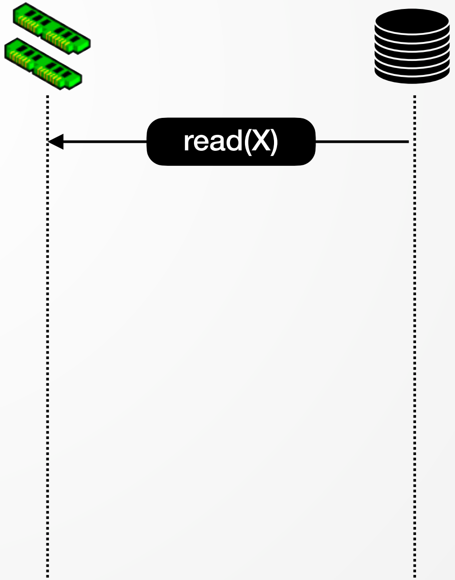
Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

	Balance
X	1000
Q	100



ANATOMY OF A TRANSACTION

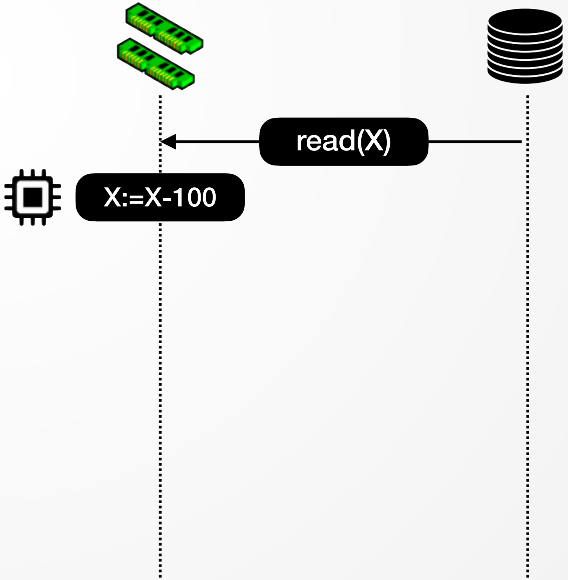
Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

	Balance
X	1000
Q	100



Anatomy of a Transaction

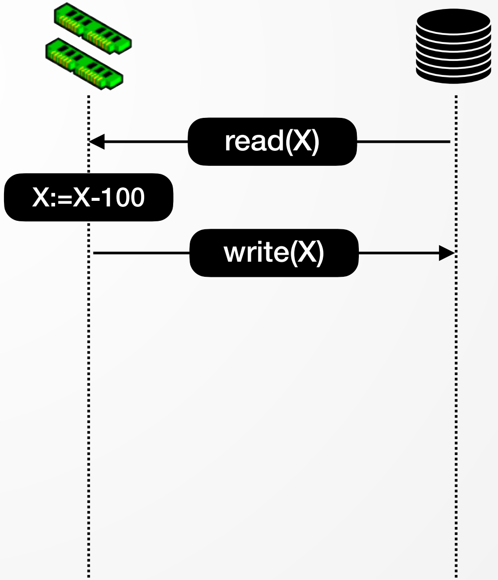
Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

	Balance
X	900
Q	100



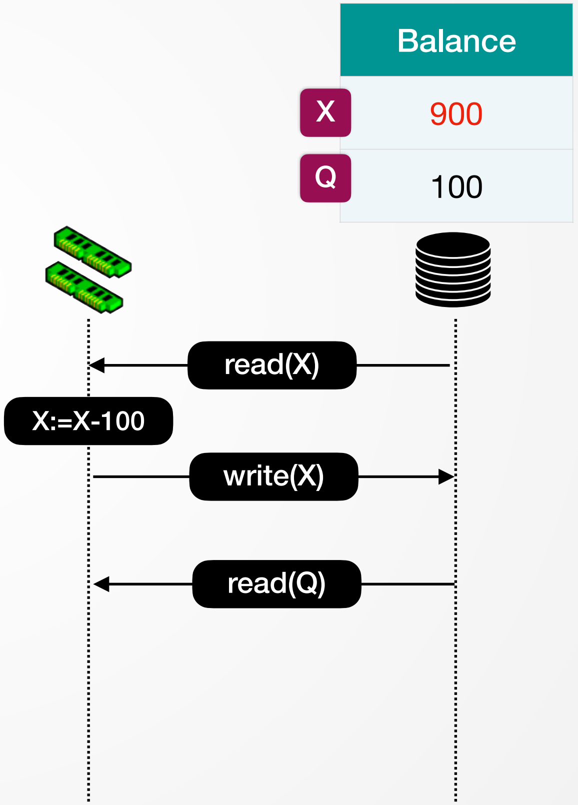
ANATOMY OF A TRANSACTION

Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)



Anatomy of a Transaction

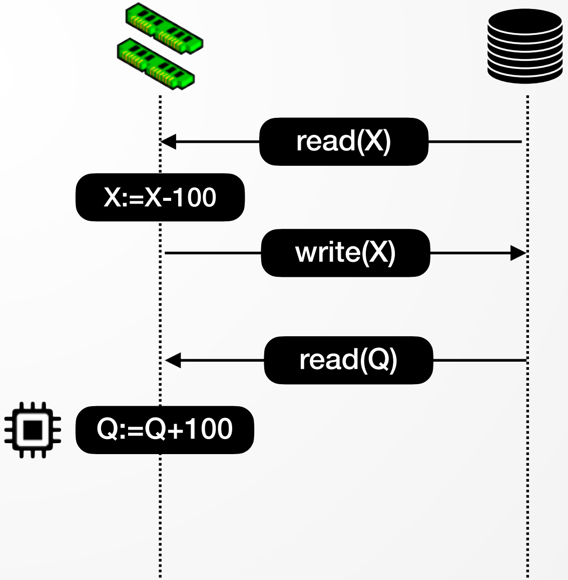
Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

	Balance
X	900
Q	100



ANATOMY OF A TRANSACTION

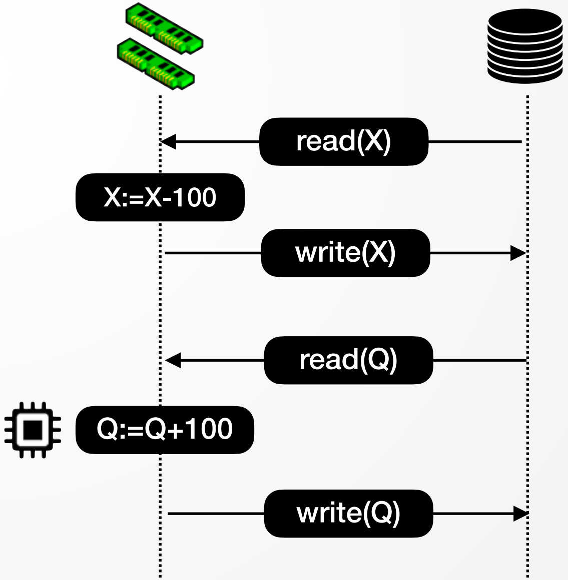
Classic Example

T1: We want to transfer **100sek** from **X** to **Q**.

That involves the following operations:

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

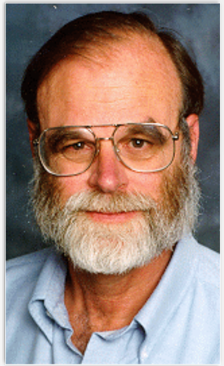
	Balance
X	900
Q	200



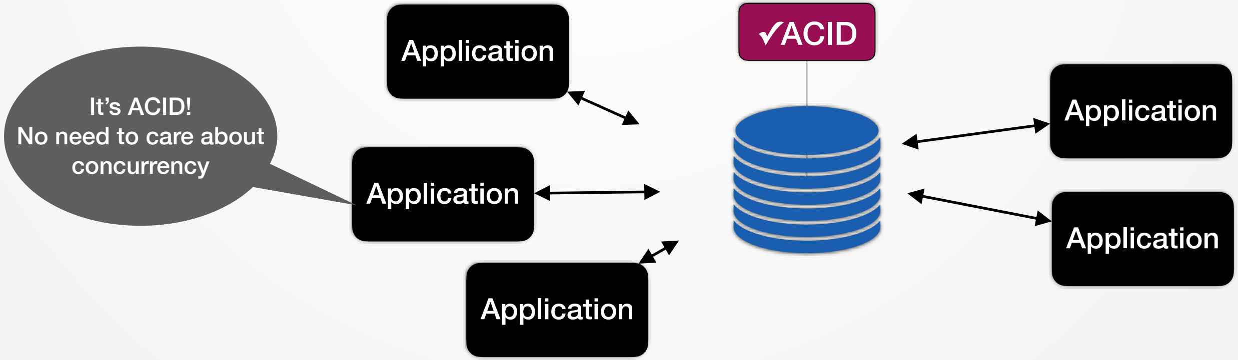
ACID

The core 4 properties for Transactions

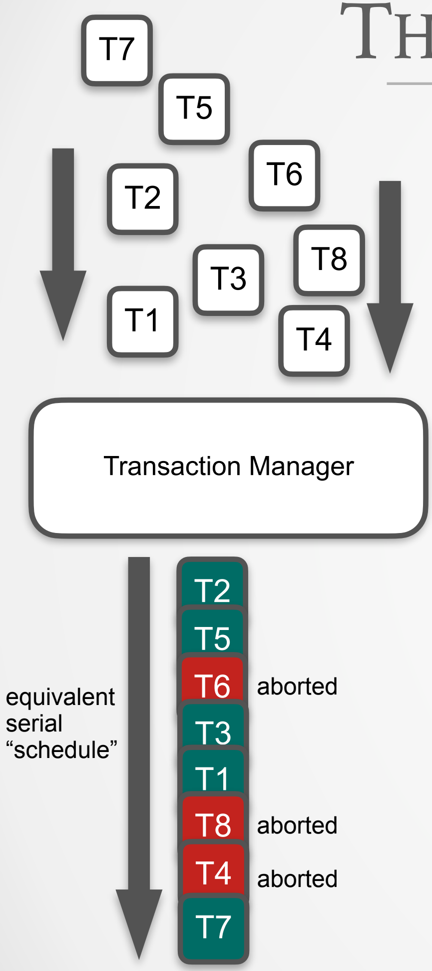
- **Atomicity** : “all transaction commands are committed or none”
- **Durability**: “all transaction object updates are persisted”
- **Isolation**: “transactions do not ‘compete’ but are isolated”
- **Consistency**: “no relational model/constraint violations”

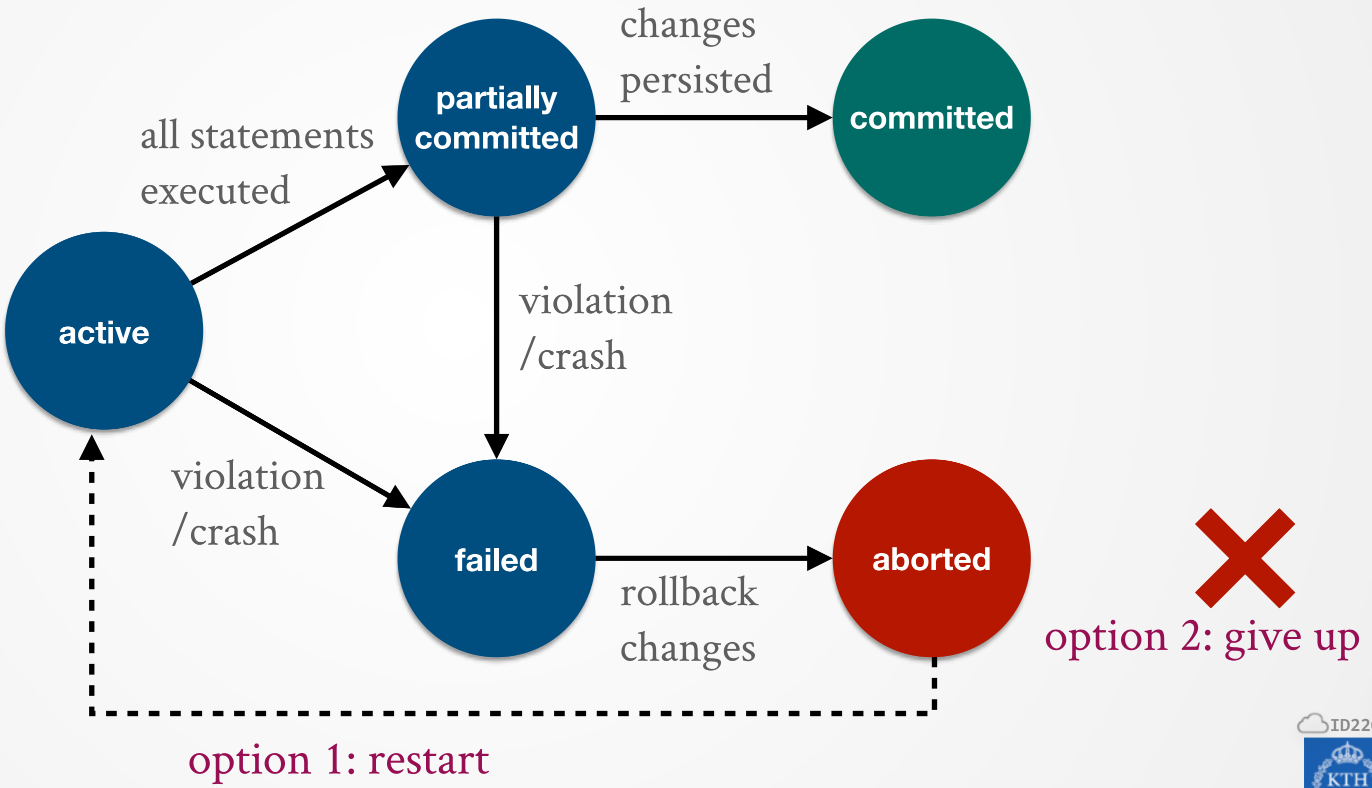


Jim Gray
Turing Award Winner
1944-2012



The Two Outcomes of Atomicity





ACID Challenges

Single-DB Transactions

	Balance
X	1000
Q	100

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

Distributed Transactions

shard #1

	Balance
X	1000

shard #2

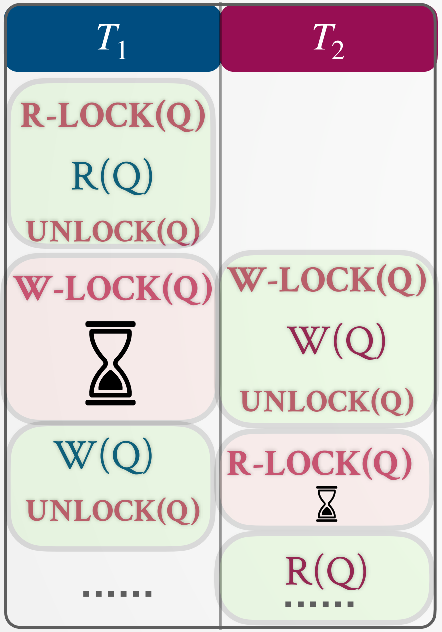
	Balance
Q	100

Atomicity: write ahead log + rollback | +Atomic Commit Protocol

Durability: persistent storage | +Replication (i.e., SMR)

Isolation: concurrency control

ISOLATION THROUGH LOCKING



A standard (pessimistic) concurrency control mechanism to isolate transaction is to grant read and write locks.

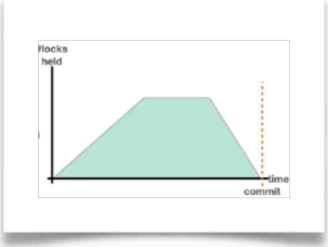
However, naive locking does not enforce isolation

Two Phase Locking (2PL)

Each transaction should acquire all necessary locks first and then release them.

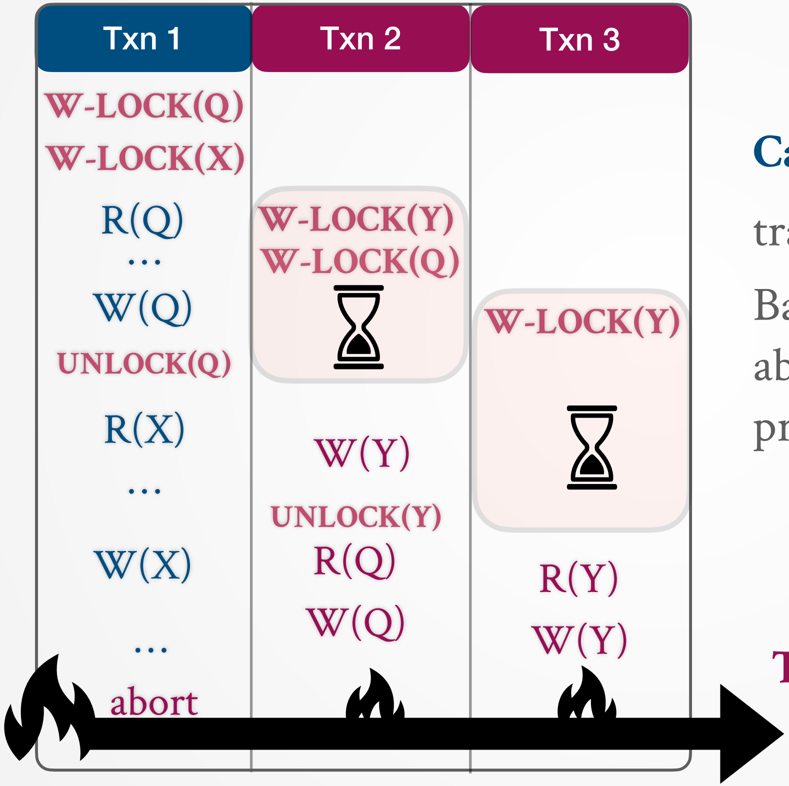
Growing Phase: Locks are acquired/upgraded and no locks are released.

Shrinking Phase: Locks are released/downgraded but no locks are acquired.



Core invariant: never acquire any lock after a lock has been released.

2PL + Cascading Aborts



Cascading Aborts are common in multi-transactional workloads.

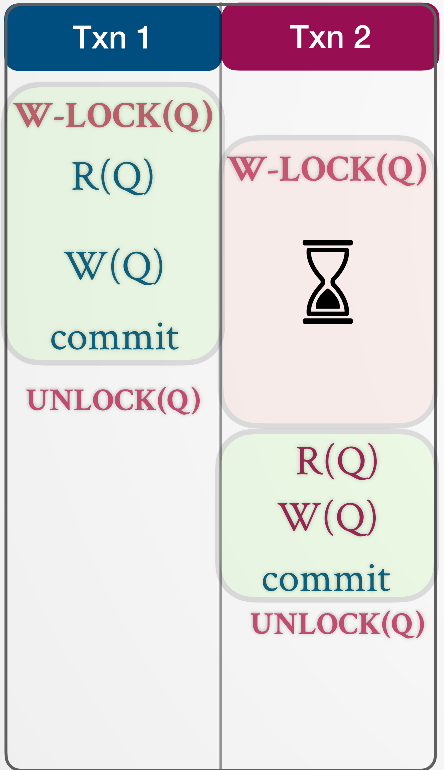
Basic 2PL does not prevent cascading aborts and as a result…we can lose progress across many transactions.

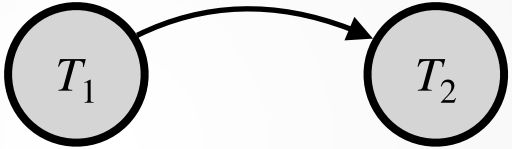
Txn2 and Txn3 also need to abort

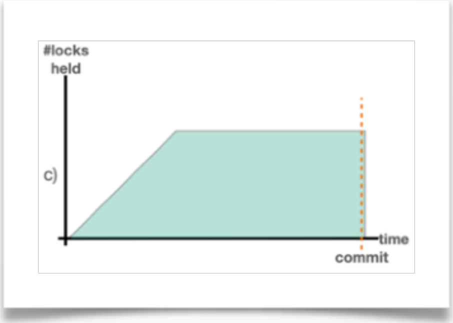
STRONG STRICT 2PL EXAMPLE

Strong-Strict 2PL (SS2PL) or Rigorous 2PL adds the following constraint to PL:

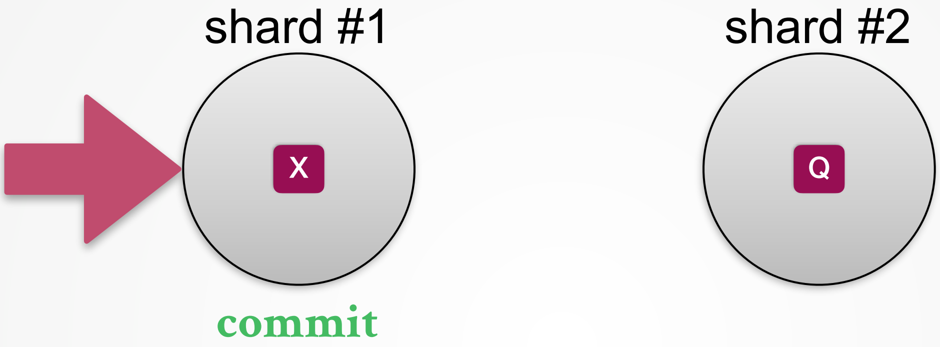
- **All locks are released only after the transaction has completed (abort/commit)**







Distributed ACID



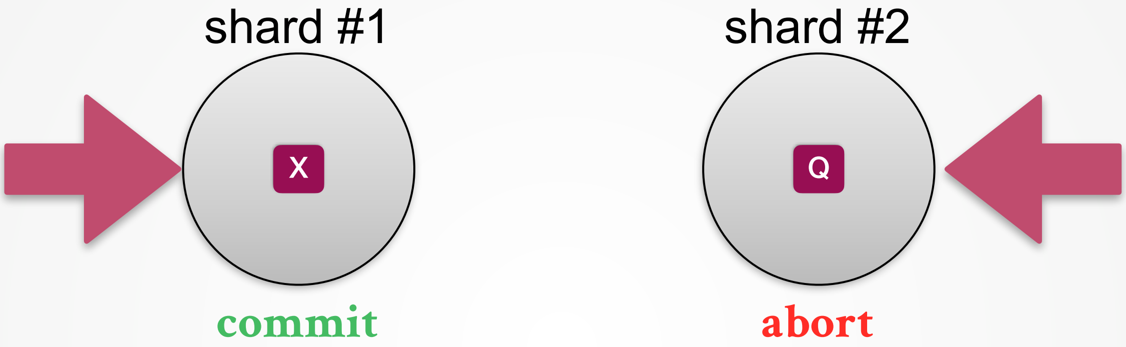
Transaction T_1

1. read(X)
2. X:=X-100
3. write(X)

- Contact shard #1
- Coordinator of shard#1 acquires X lock and commits T_1

ID2203
KTH-2023

DISTRIBUTED ACID



Transaction T_2

1. read(X)
2. X:=X-100
3. write(X)
4. read(Q)
5. Q:=Q+100
6. write(Q)

- We need to commit/abort transaction across shards.
- Either all partitions/shards should commit transaction or none!
- How do we achieve that?

- Using **Atomic Commitment**

ATOMIC COMMIT

- Transaction Coordinator (leader)
- Cohorts (followers)
- Given a proposed transaction T
  - Commit if **all** followers agree to commit
  - Abort if **at least one** follower aborts or fails

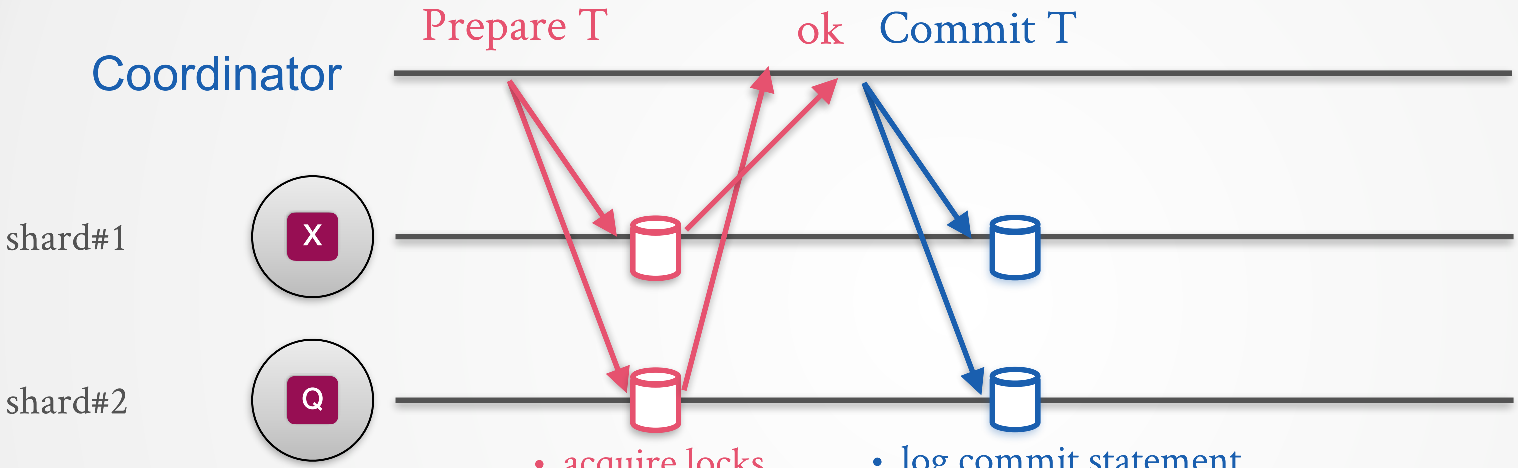
- **Request**: Transaction T
- **Indication**: Commit | Abort

Atomic Commit VS Consensus(Paxos)

Validity	Decide Commit or Abort	Decide any Proposed Value
Fault Tolerance	$f = 0$ (but can be improved)	$f < N/2$
Leader	Single Coordinator Process	Any process can propose
Agreement	Unanimous	Quorum-based

Two Phase Commit (2PC) is the defacto Atomic Commitment Protocol

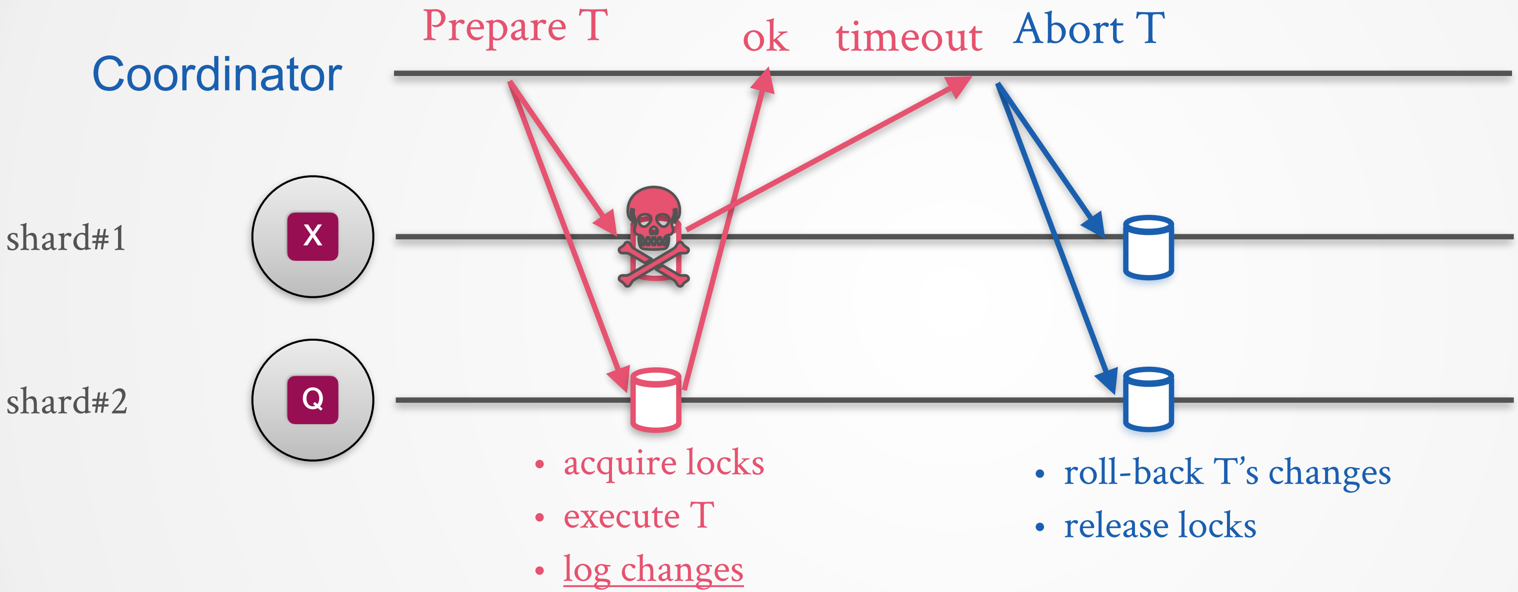
2PC (Two Phase Commit)



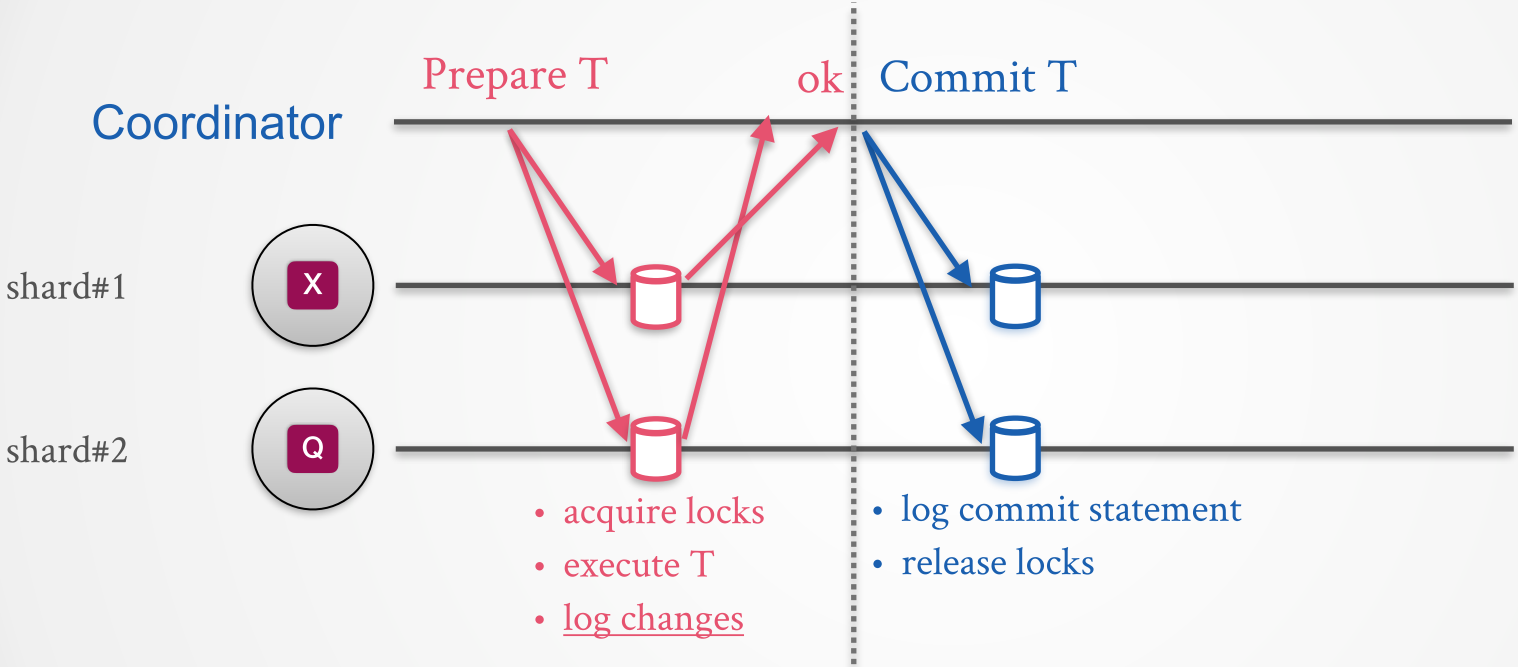
- acquire locks
- execute T
- <u>log changes</u>

- log commit statement
- release locks

2PC (Two Phase Commit)



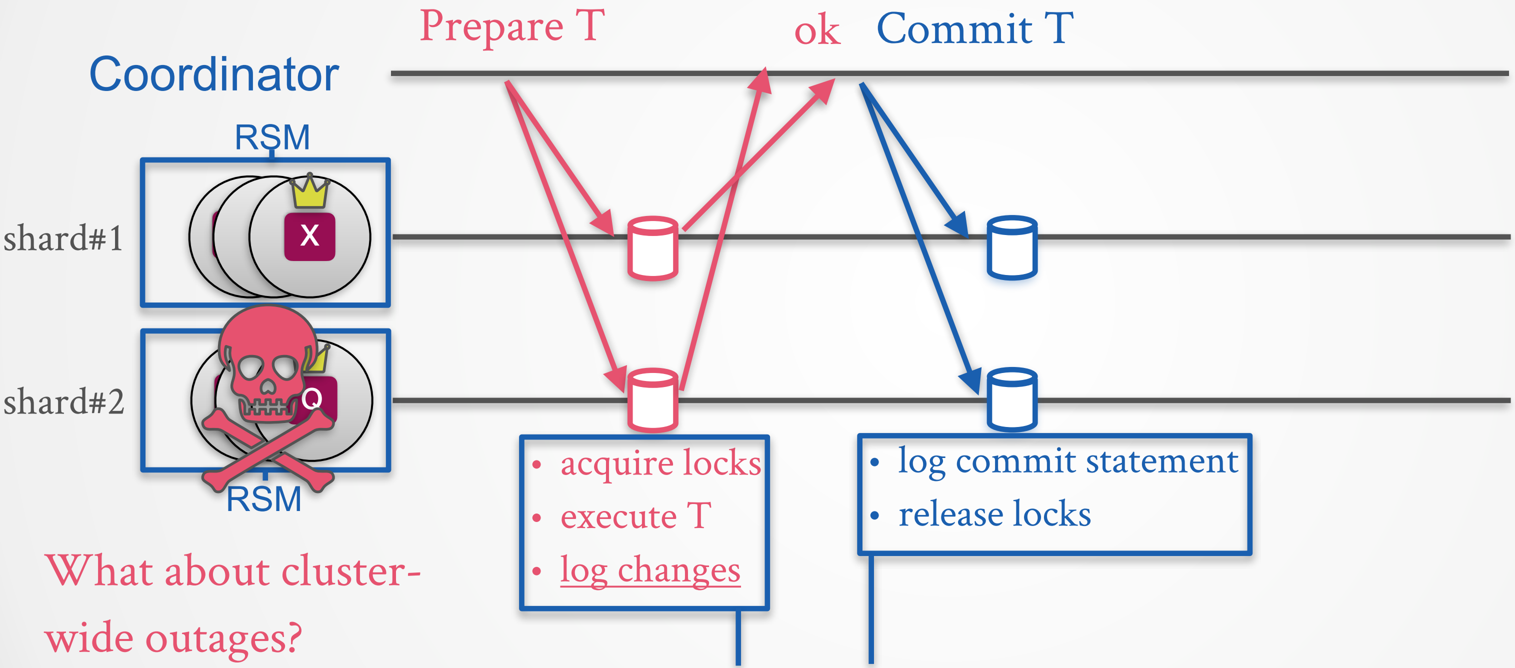
2PC (Two Phase Commit)



If any process fails here the 2PC times out and aborts

If any process fails here the decision has already be made but is the transaction durably persisted?

2PC (Two Phase Commit)

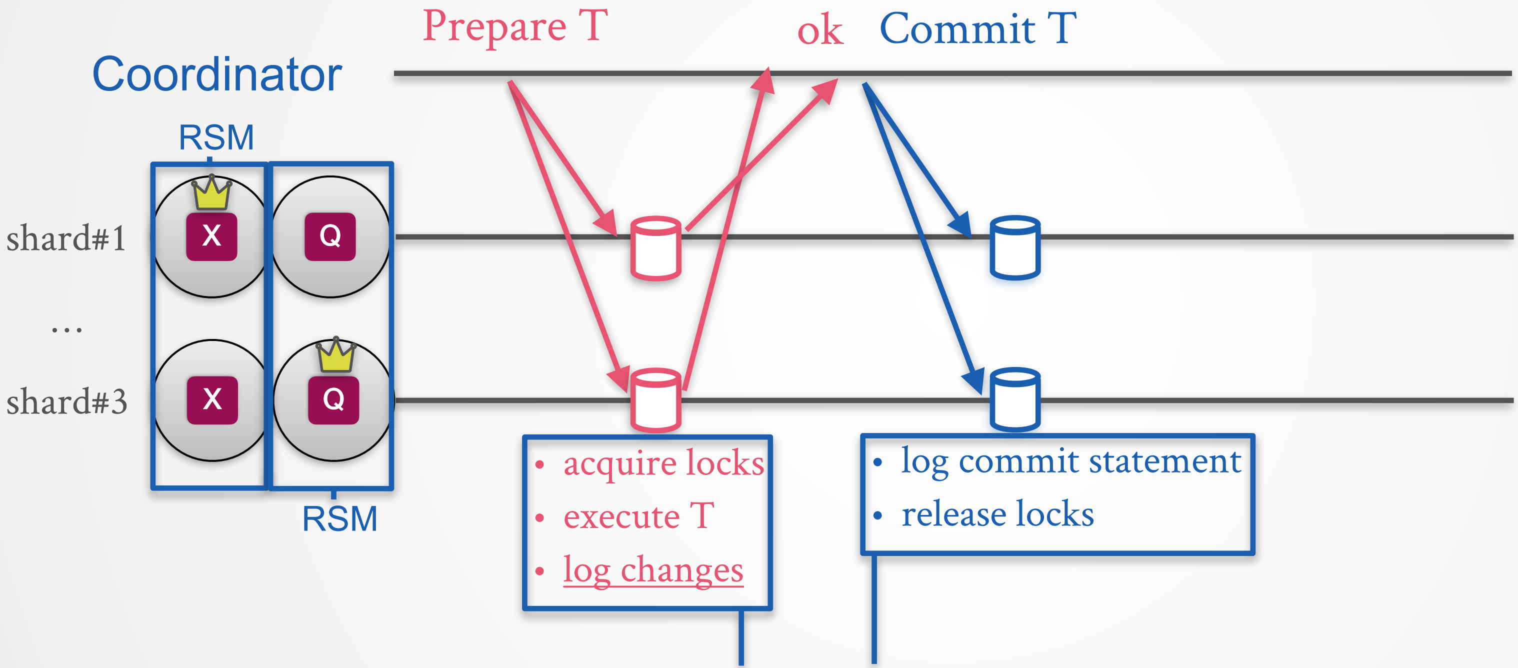


What about cluster-wide outages?

Can we durably persist that?

- All 2PC actions are decided commands in each shard's RSM
- Protocol is executed by each respected leader in a shard

2PC (Two Phase Commit)

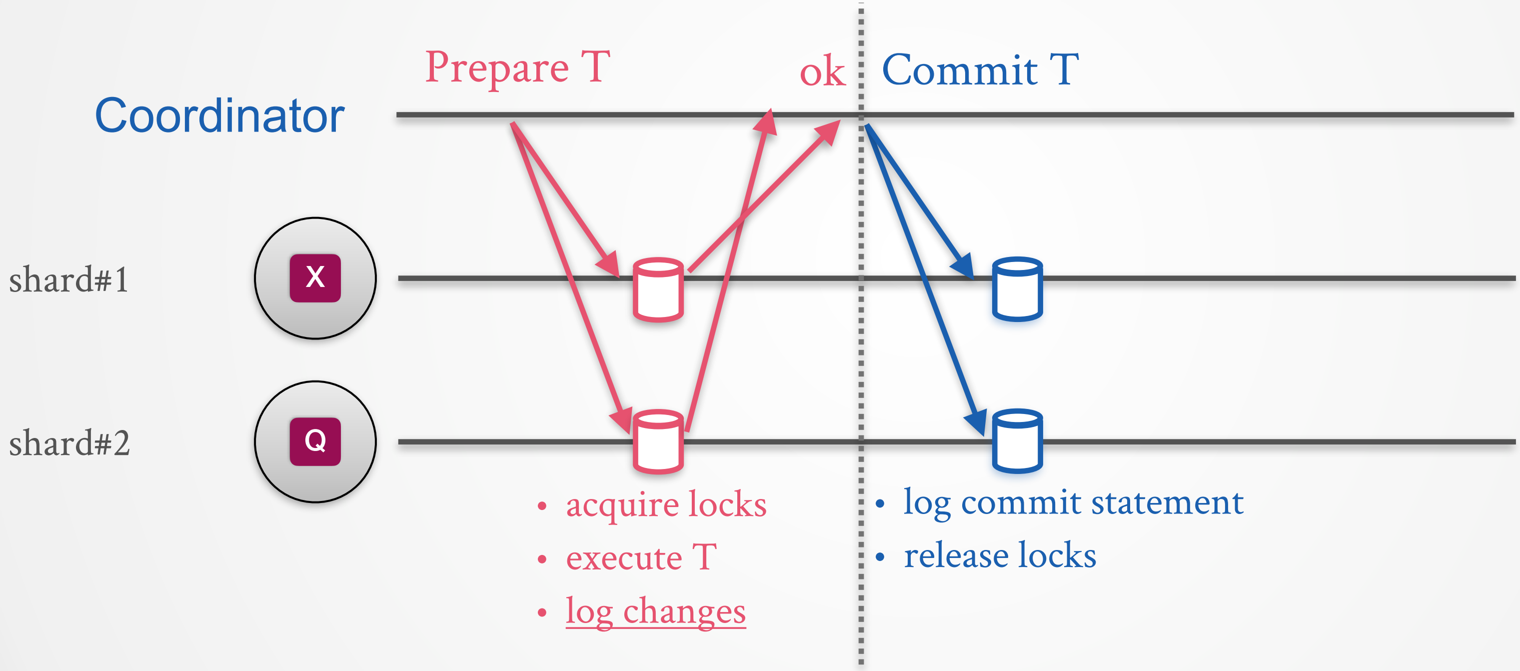


- i.e., Google Spanner geo-replication

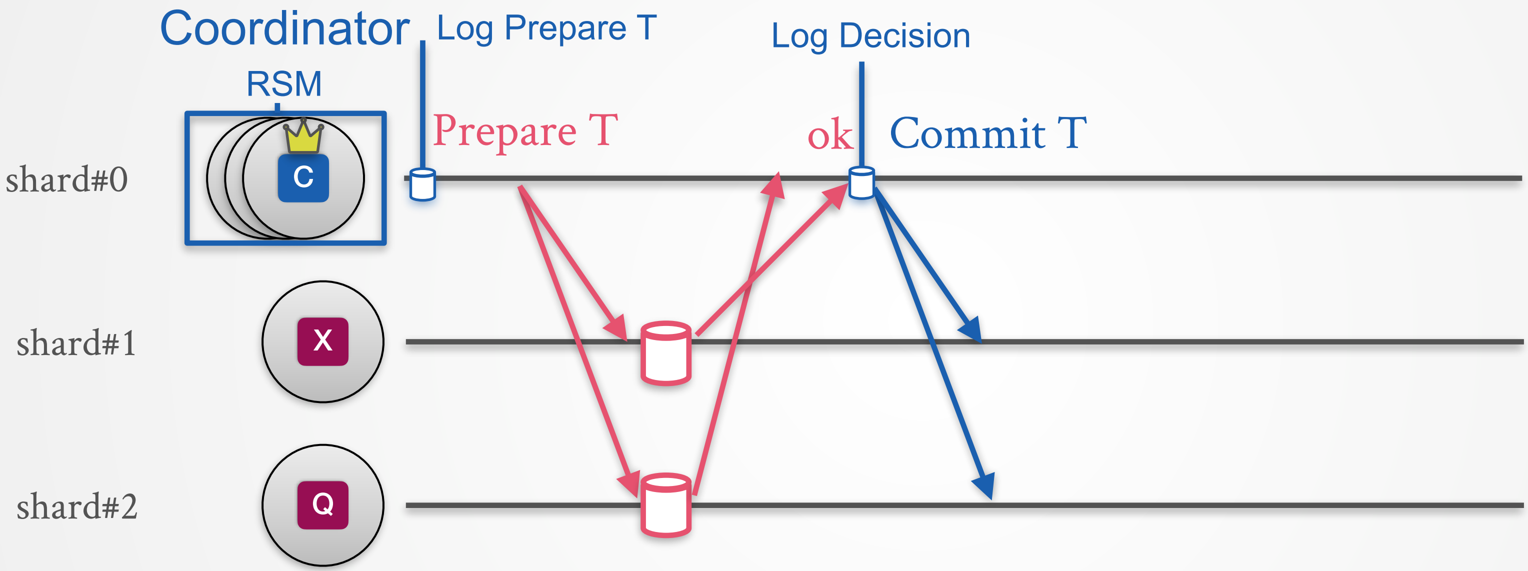
- All 2PC actions are decided commands in each shard's RSM
- Protocol is executed by each respected leader shard and replicated to other shards

2PC Coordinator Crashes

What if the coordinator crashes here? The protocol **might block in an undecided state**

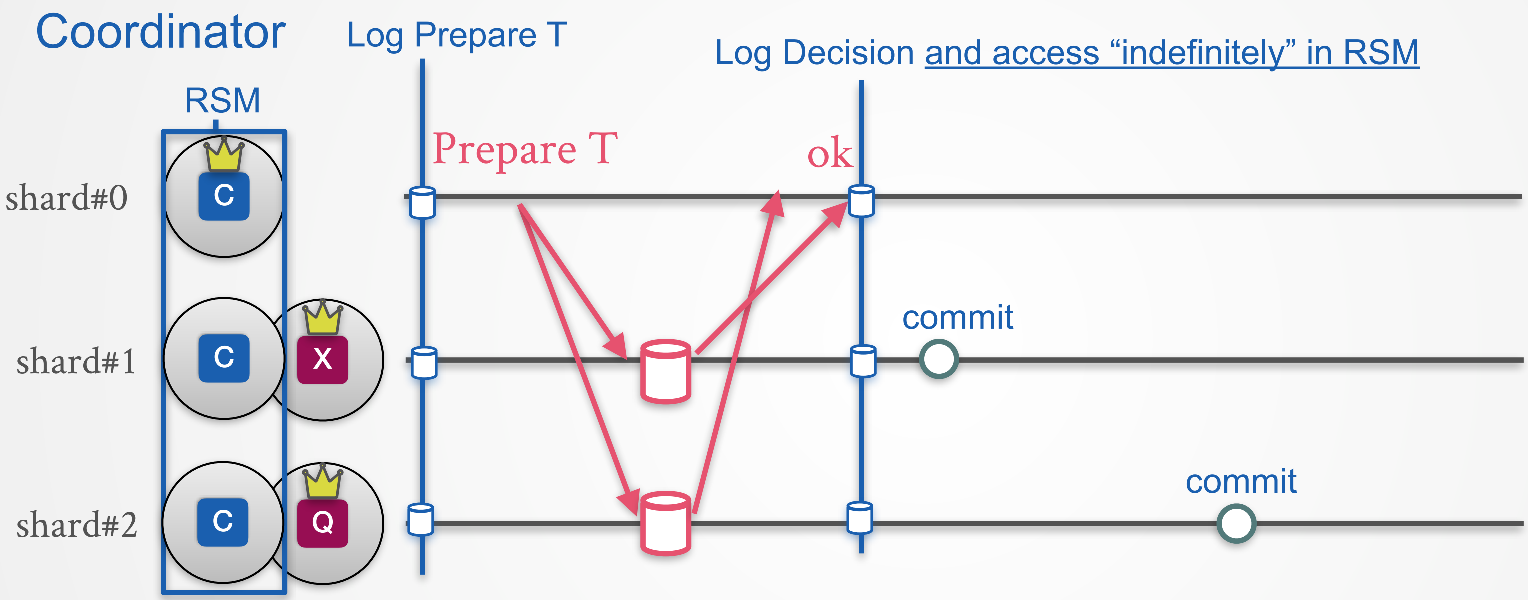


RELIABLE 2PC V.1



- This approach ensures that Transaction Decisions are reliably decided on a log.
- (New) Coordinator can access status from RSM (Zookeeper, Raft, OmniPaxos) and finalize phase 2 of the protocol or restart it if stuck in prepare phase.

RELIABLE 2PC V.2 (STATE OF THE ART)

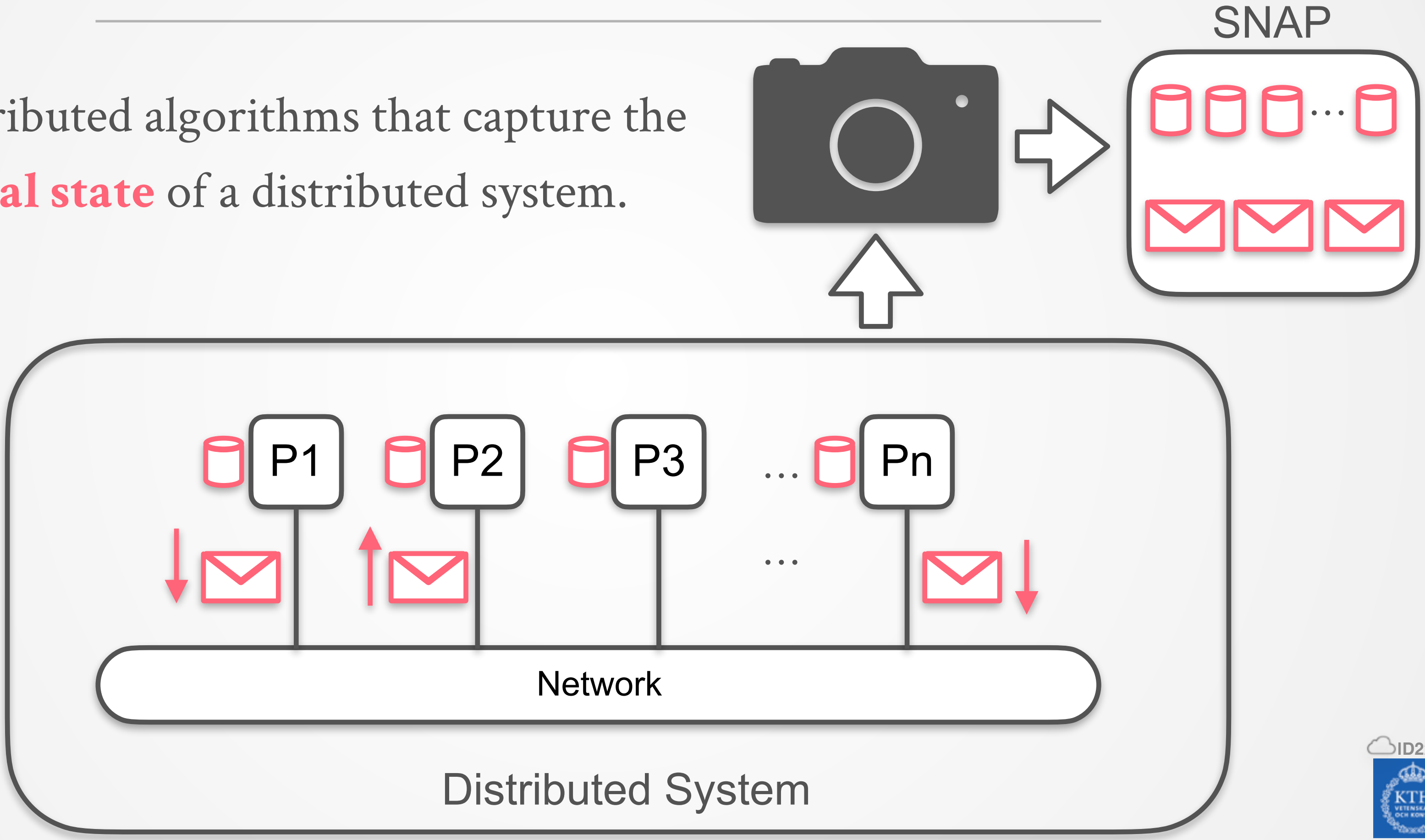


- This approach ensures that Transaction Decisions are reliably replicated across shards.
- All servers can apply finalize (rollbacks/commits) based on transaction status read from local RSM replica (Zookeeper, Raft, OmniPaxos)

Distributed Data Processing and Snapshots

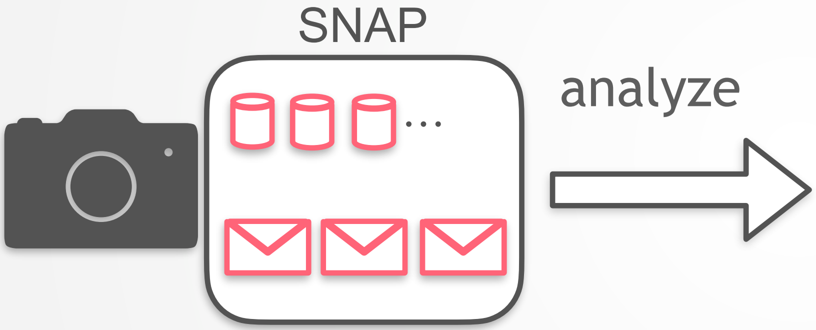
Distributed Snapshots

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



SNAPSHOT USAGES

1. Stable Property Detection



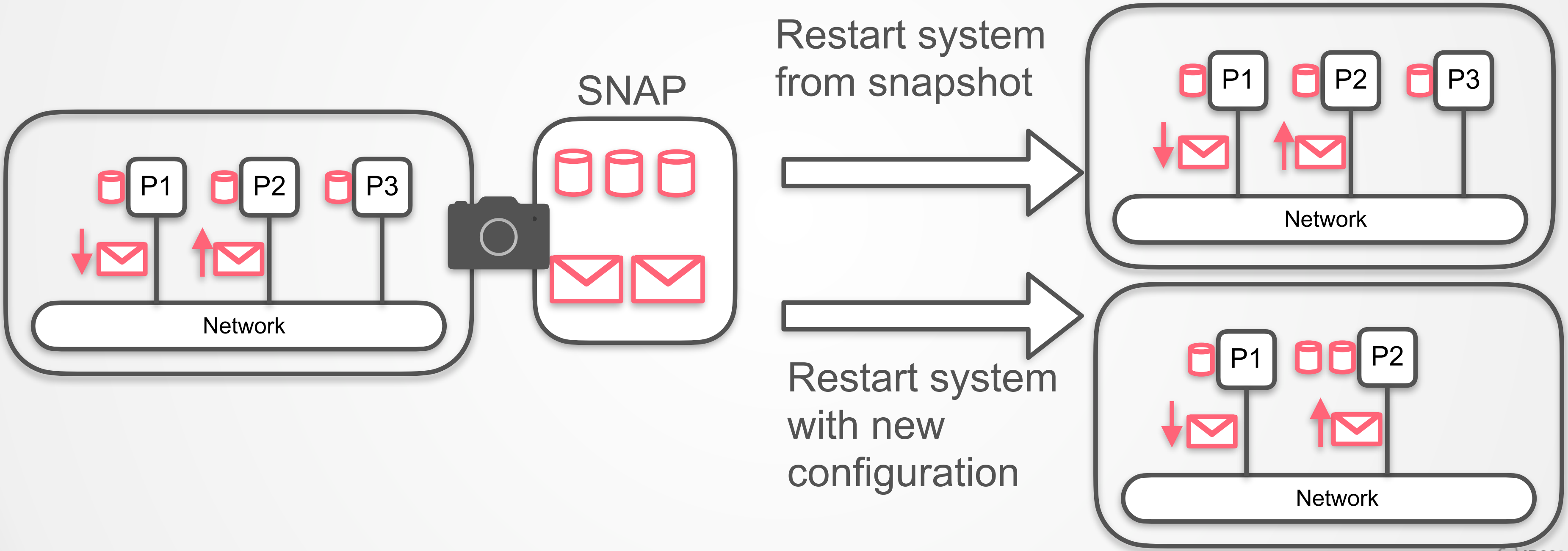
- Deadlocked execution
- Computation Terminated
- No tokens in transit

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

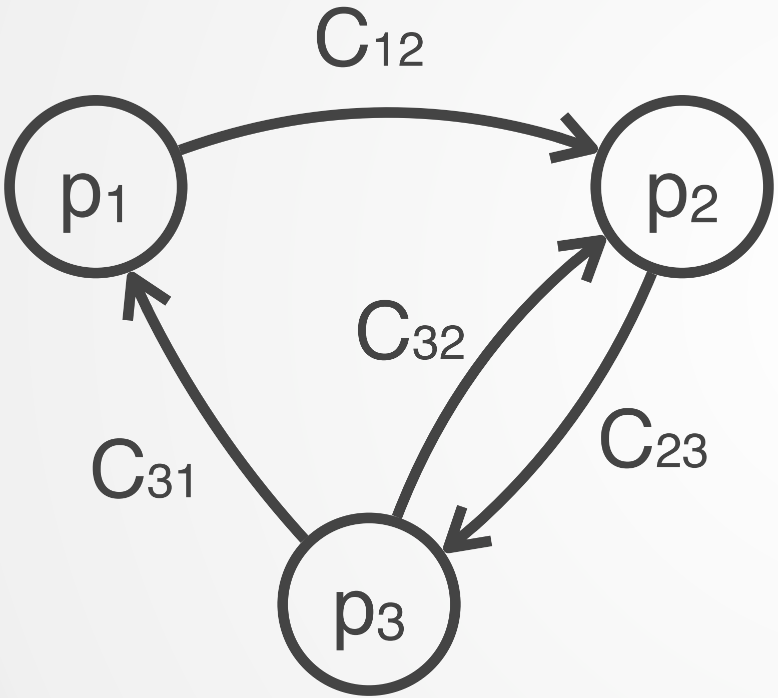
- Chandy, Lamport 88

SNAPSHOT USAGES

2. Failure Recovery and Reconfiguration



Process Model



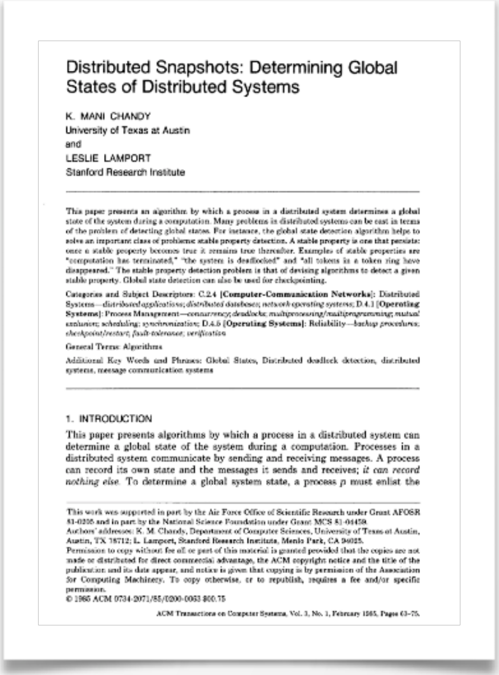
Process Graph

- 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

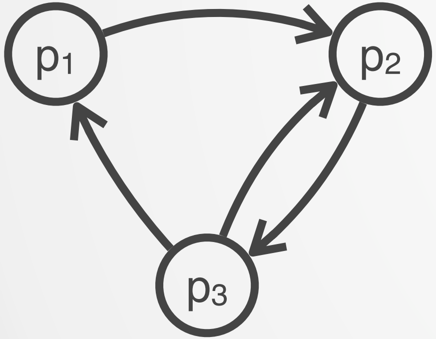
This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; *it can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

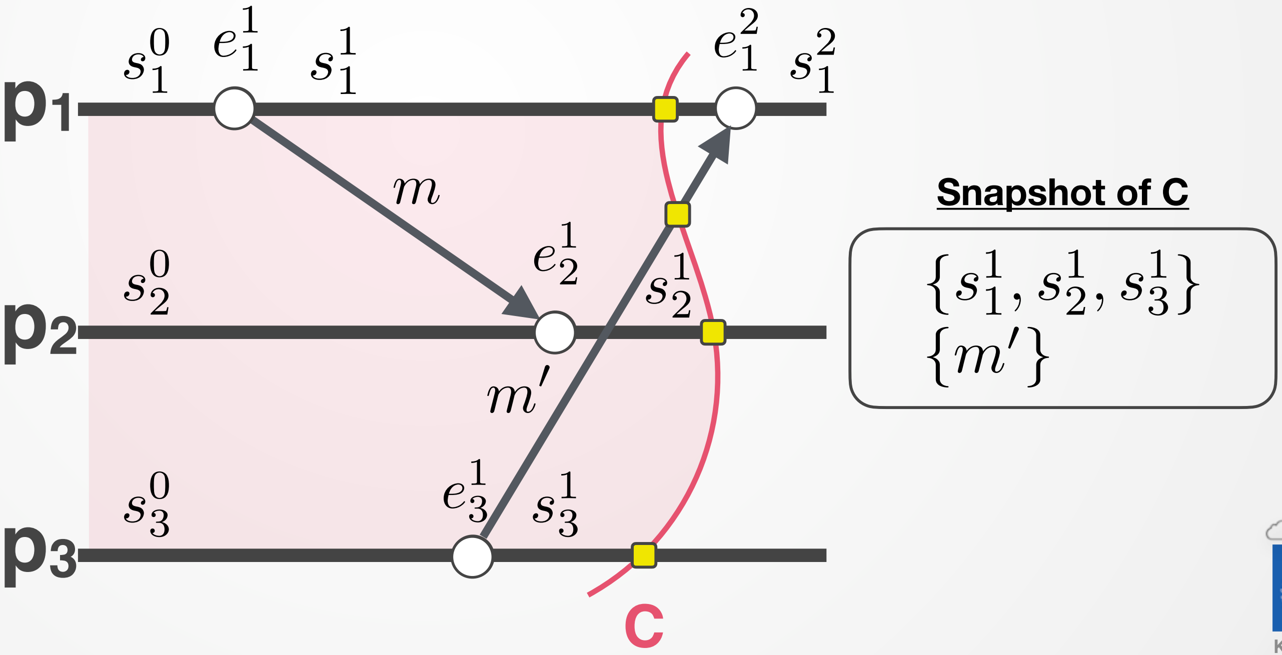
This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR 81-0205 and in part by the National Science Foundation under Grant MCS 81-04459.
Authors' addresses: K. M. Chandy, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. Lamport, 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.



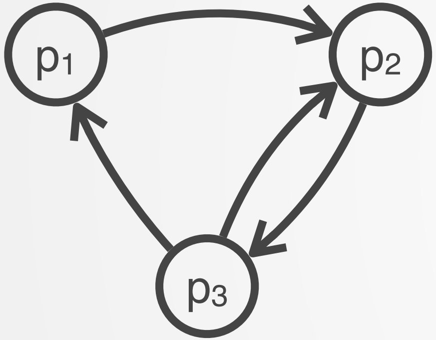


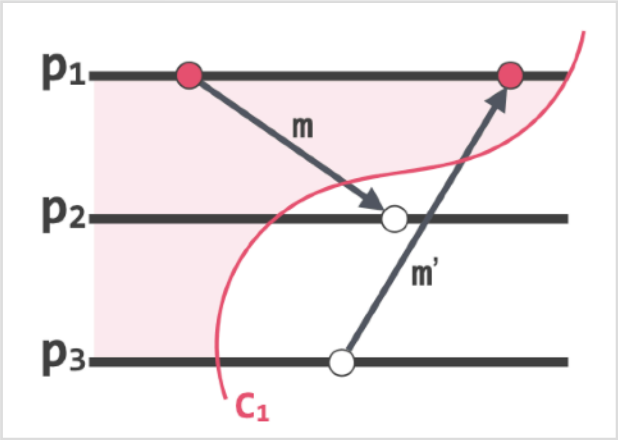
ID2203

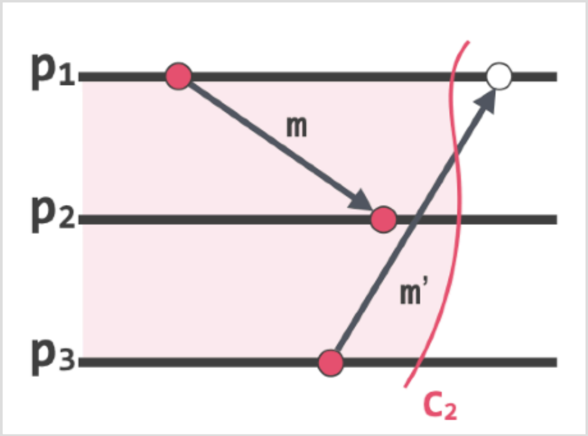
KTH-2023

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

ID2203

KTH-2023

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

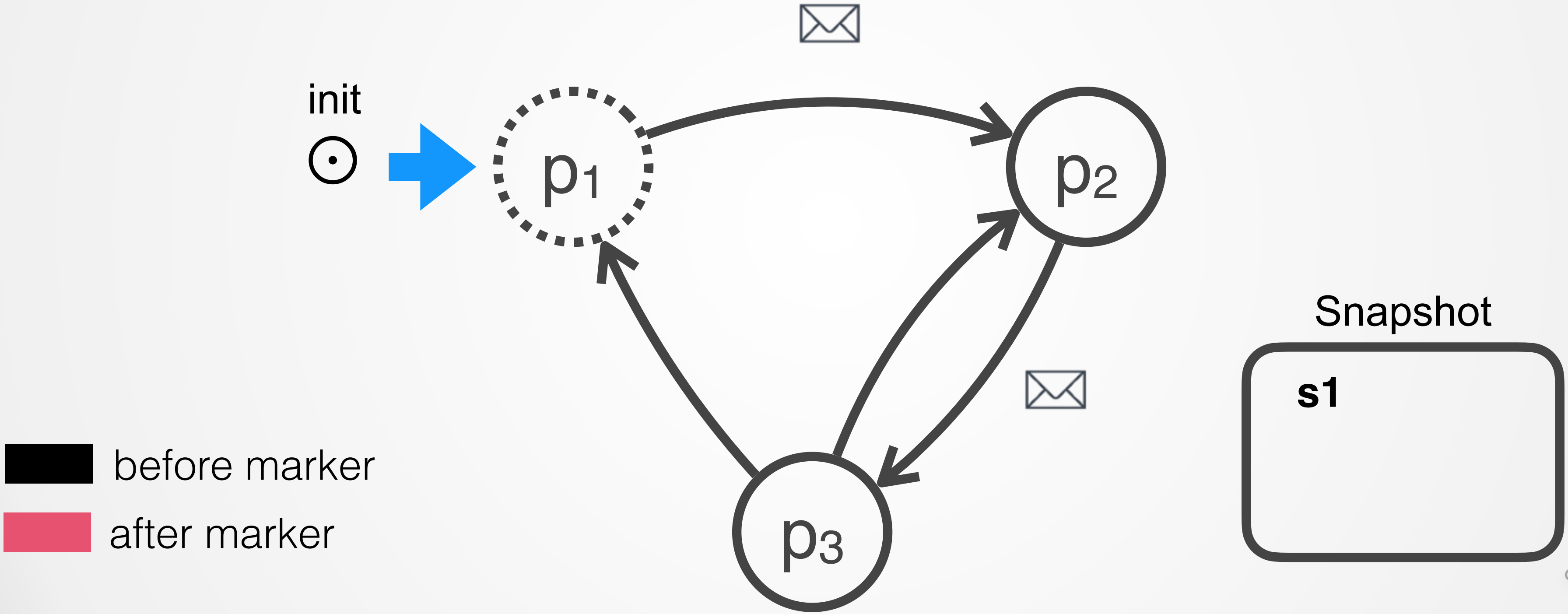
Implements: csnap, **Requires**: fiforc $(\mathbb{I}_p, \mathbb{O}_p)$

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

---

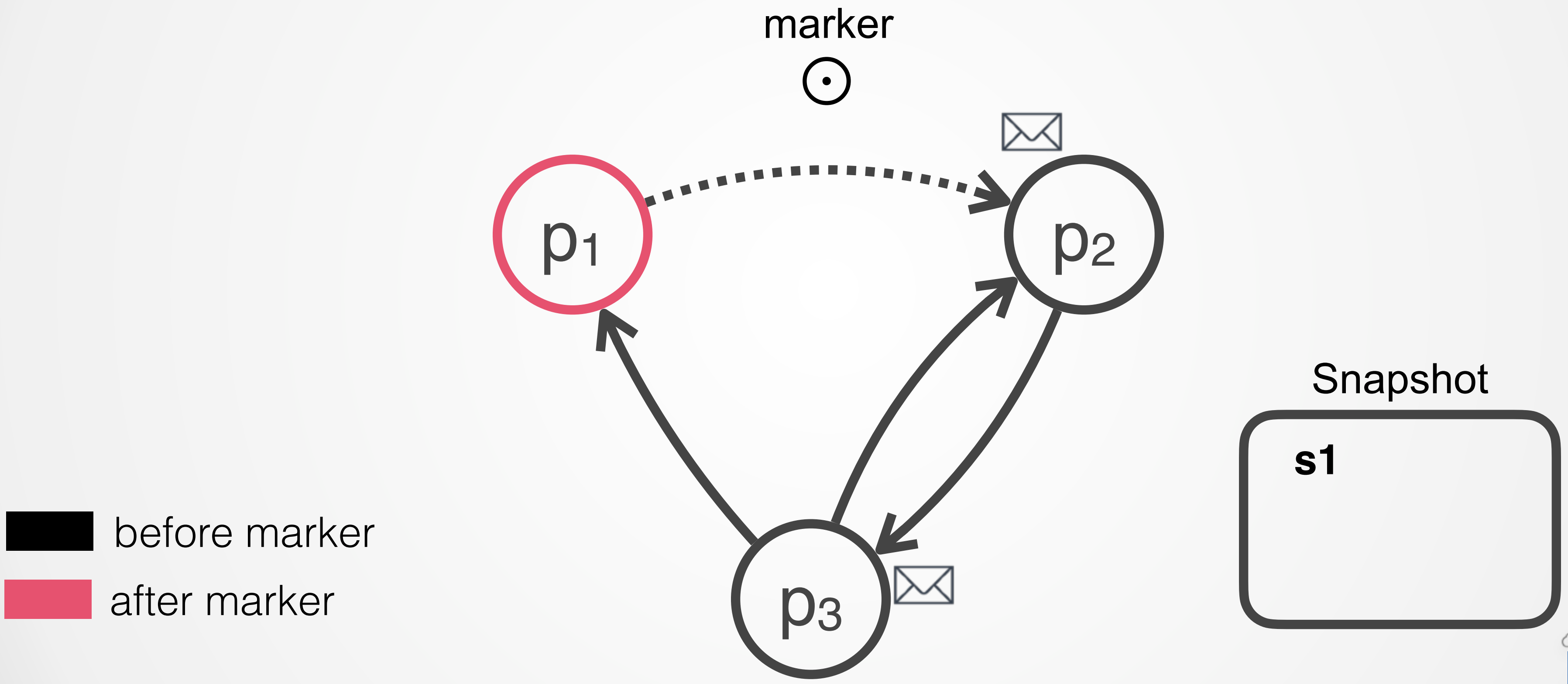
Upon $\langle \mathsf{rcvd}, m \rangle$ *on* $c_{qp} \notin \mathsf{Recorded}, m \neq \odot$
    $s_p \leftarrow \mathsf{process}(m, s_p, \mathbb{O}_p)$; ▷ regular process logic
Upon $\langle \mathsf{rcvd}, m \rangle$ *on* $c_{qp} \in \mathsf{Recorded}, m \neq \odot$
    $M_p \leftarrow M_p \cup \{m\}$;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

EXAMPLE EXECUTION

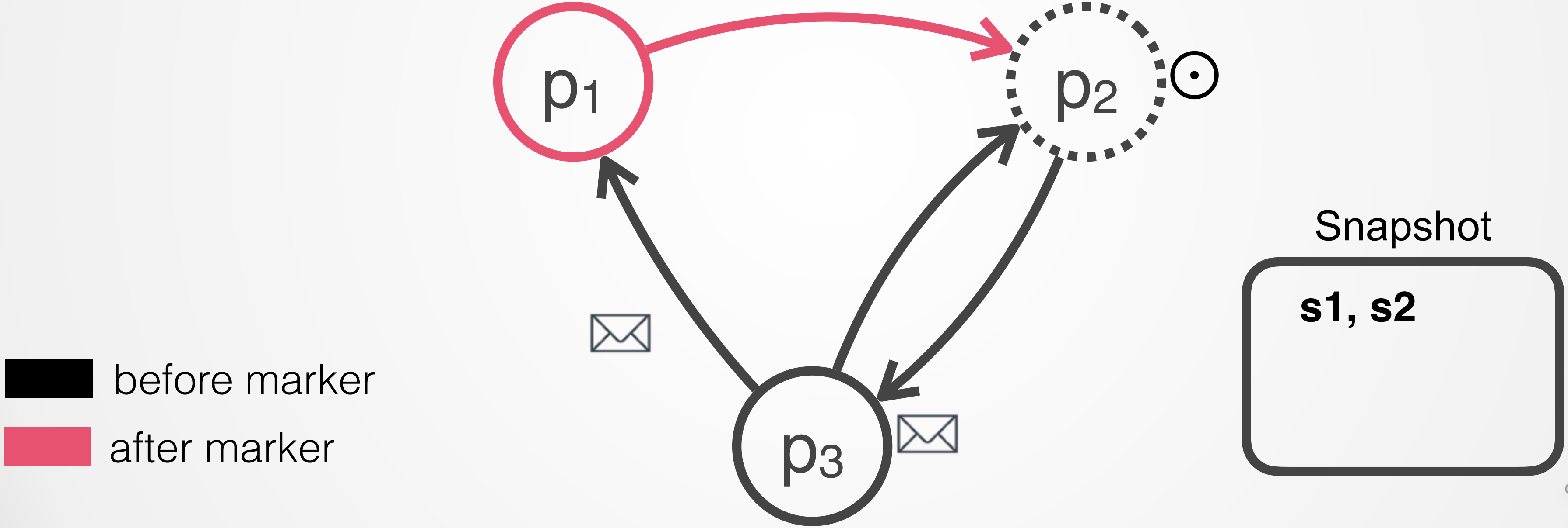


ID2203
KTH-2023

Example Execution

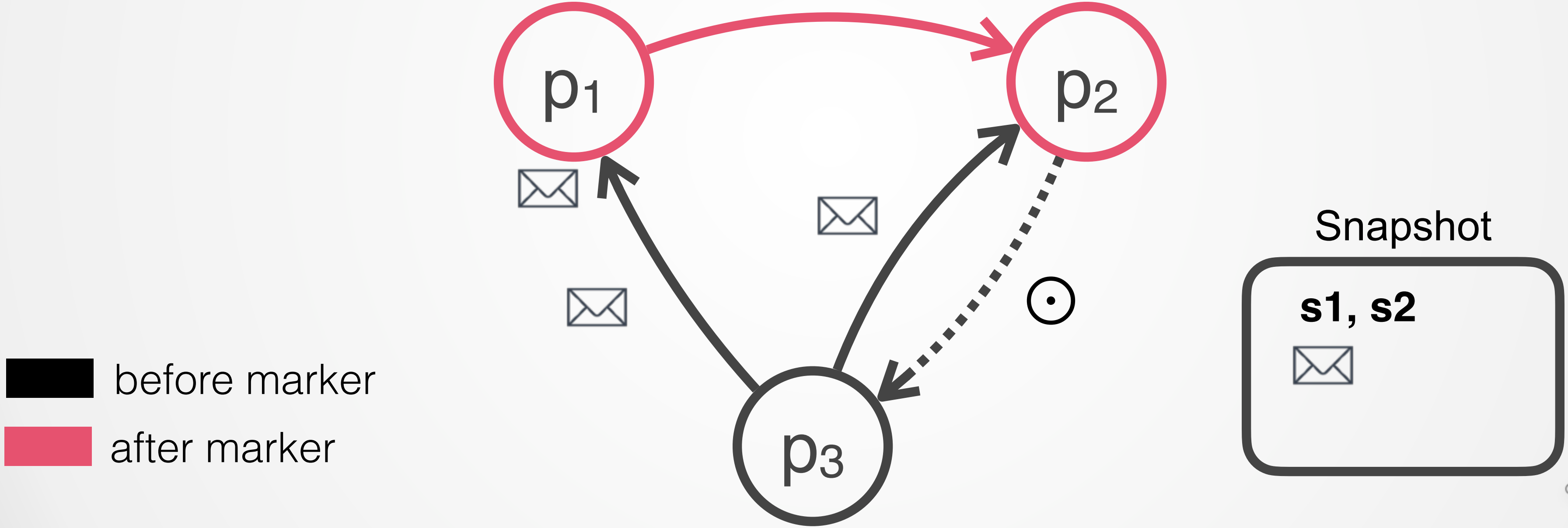


EXAMPLE EXECUTION

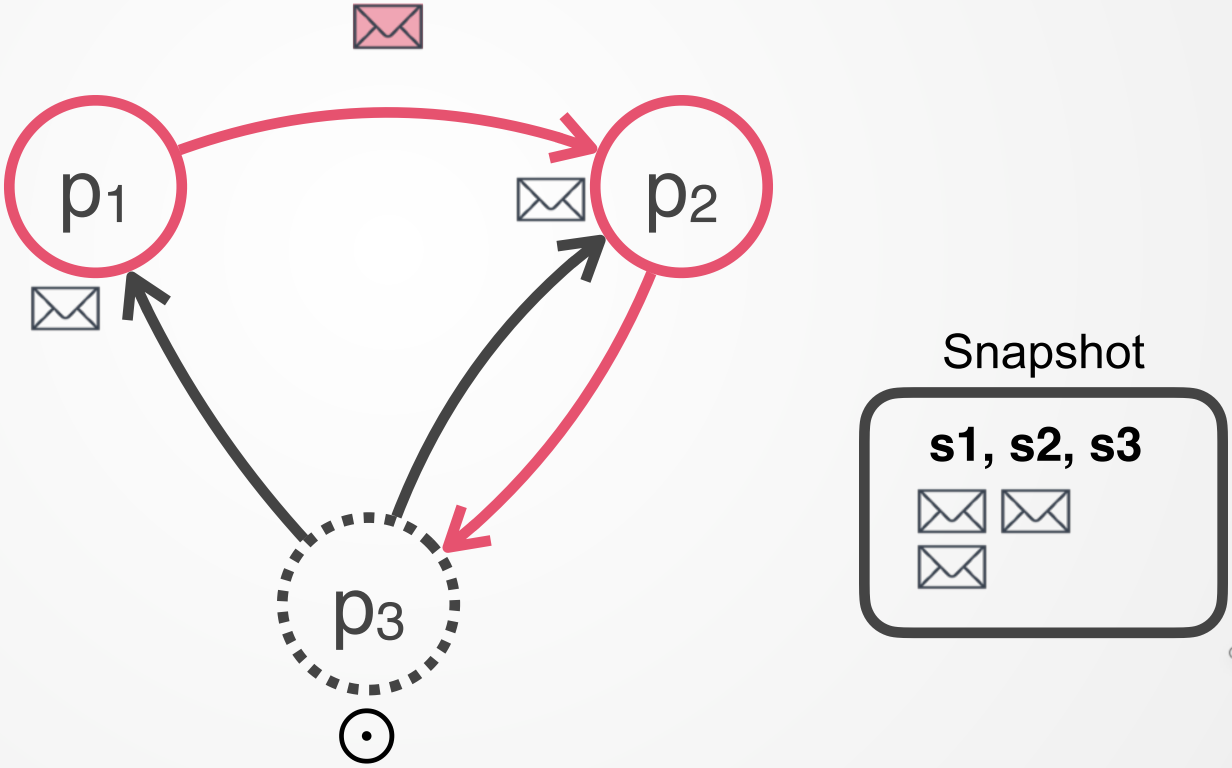


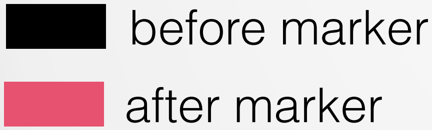
ID2203
KTH-2023

EXAMPLE EXECUTION

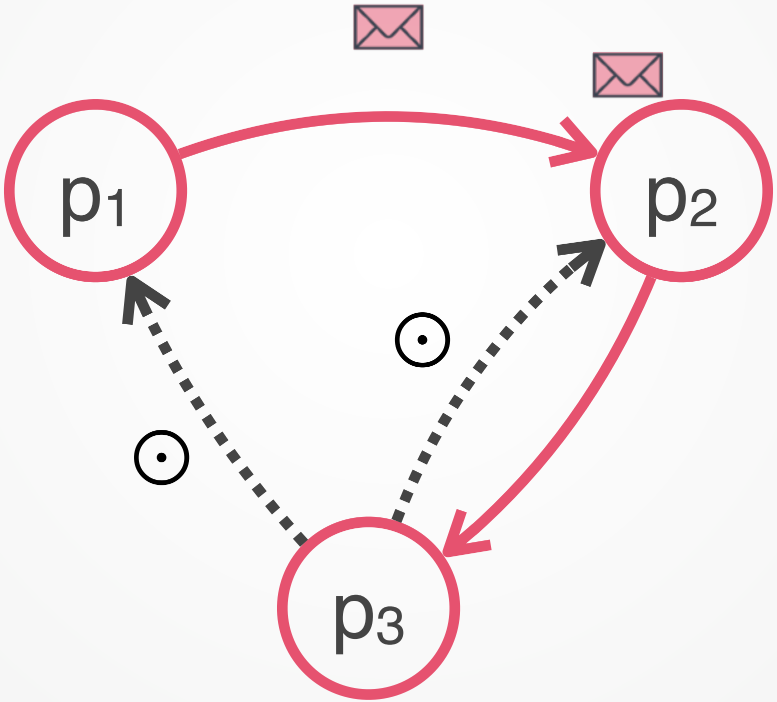


Example Execution





EXAMPLE EXECUTION

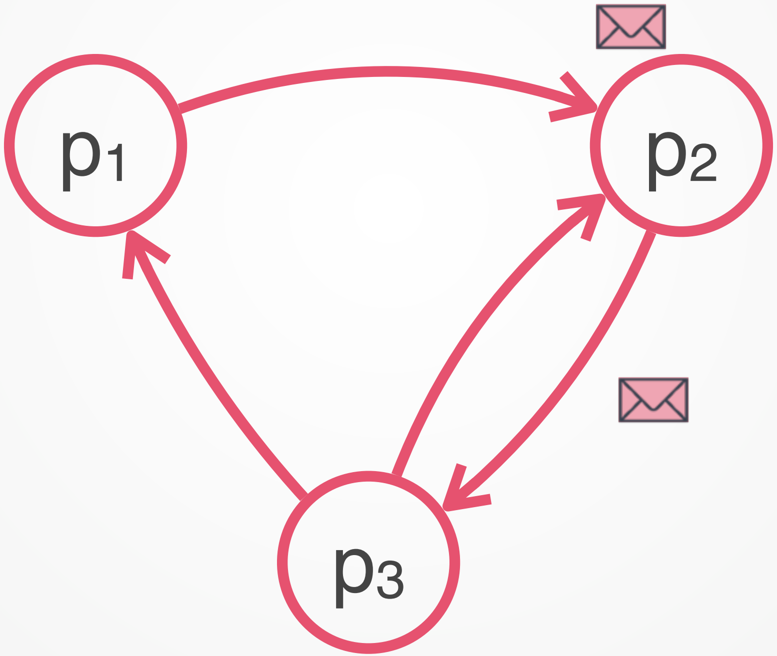


before marker
after marker

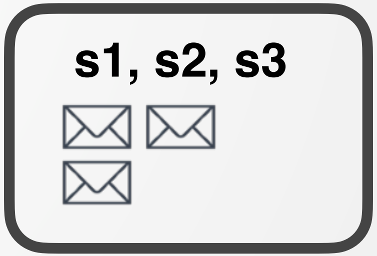
Snapshot

s1, s2, s3

EXAMPLE EXECUTION



Snapshot



before marker

after marker

ID2203

KTH-2023

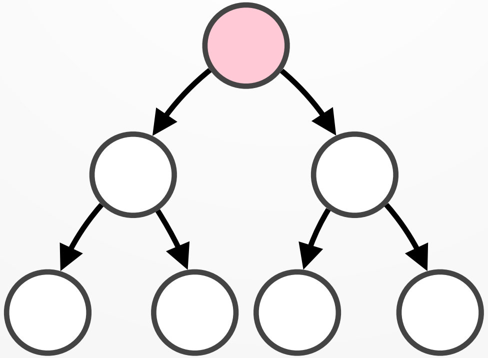
PROOF SKETCH

- **Validity**

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

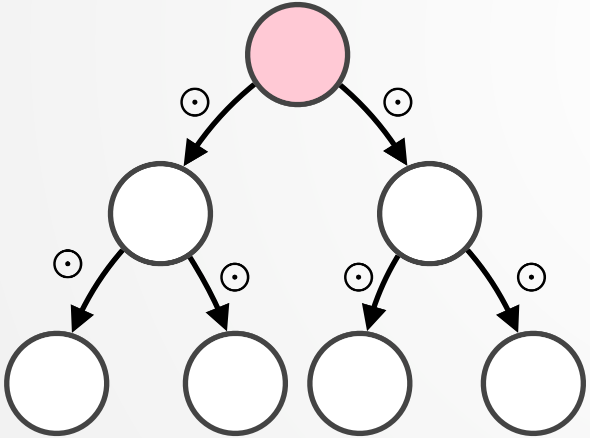
  - Validity applies to the transitive closure of reachable processes (through induction)

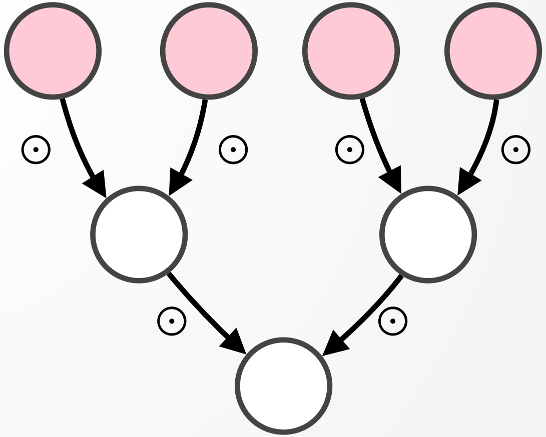
- **Termination** is satisfied **if** initiator can **reach** all tasks.



GENERALIZATION

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





Epoch Snapshotting

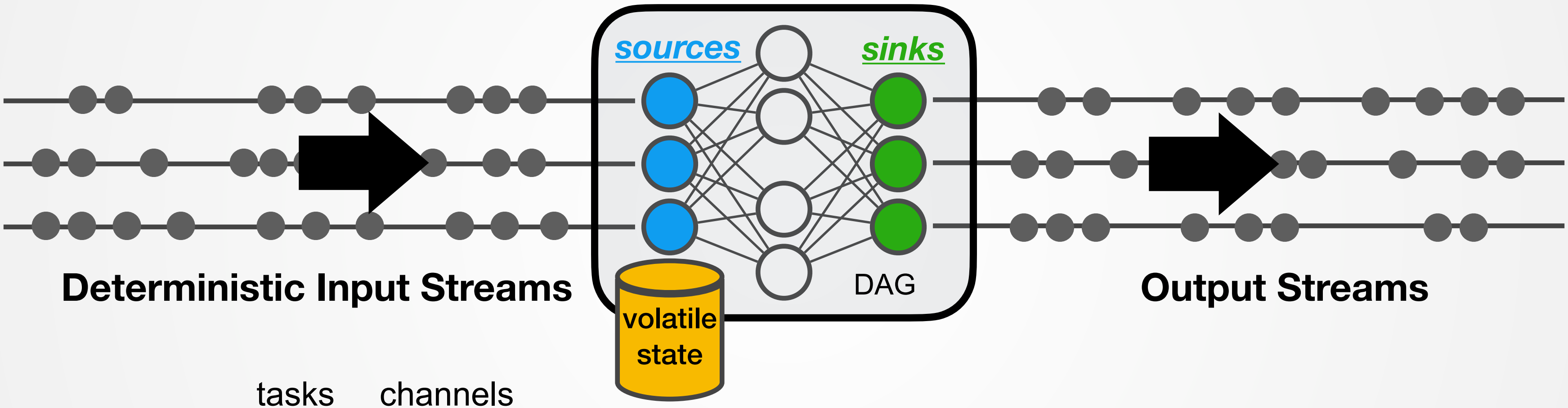
DATA PROCESSING SNAPSHOTS

- **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 PROCESSING

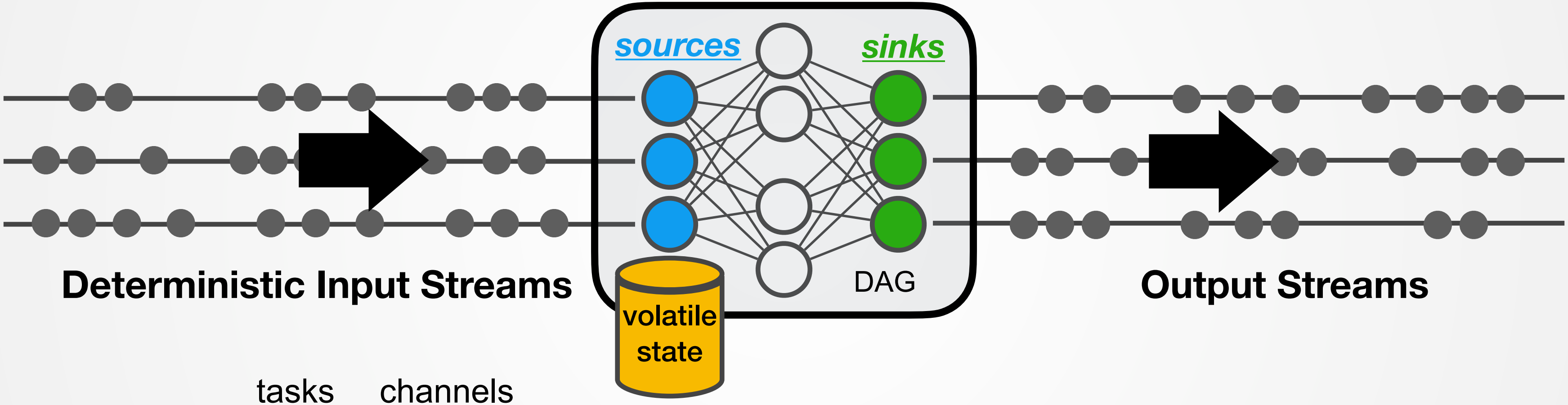


tasks channels

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

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

STREAM PROCESSING



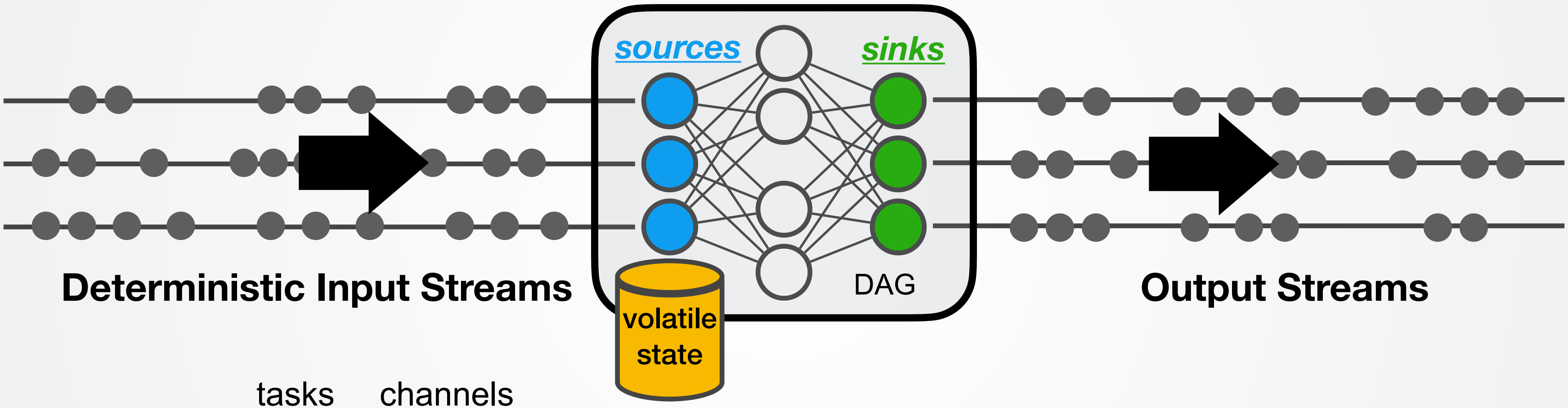
tasks channels

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

Task Actions

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

STREAM PROCESSING



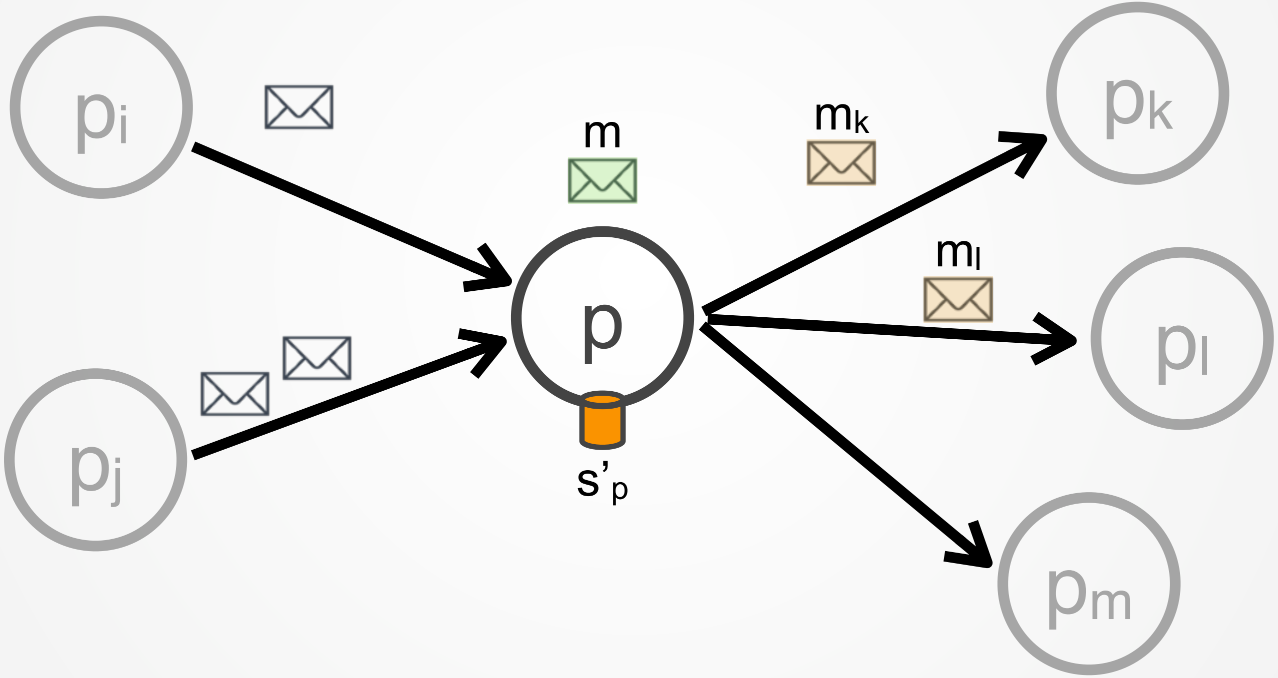
tasks channels

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

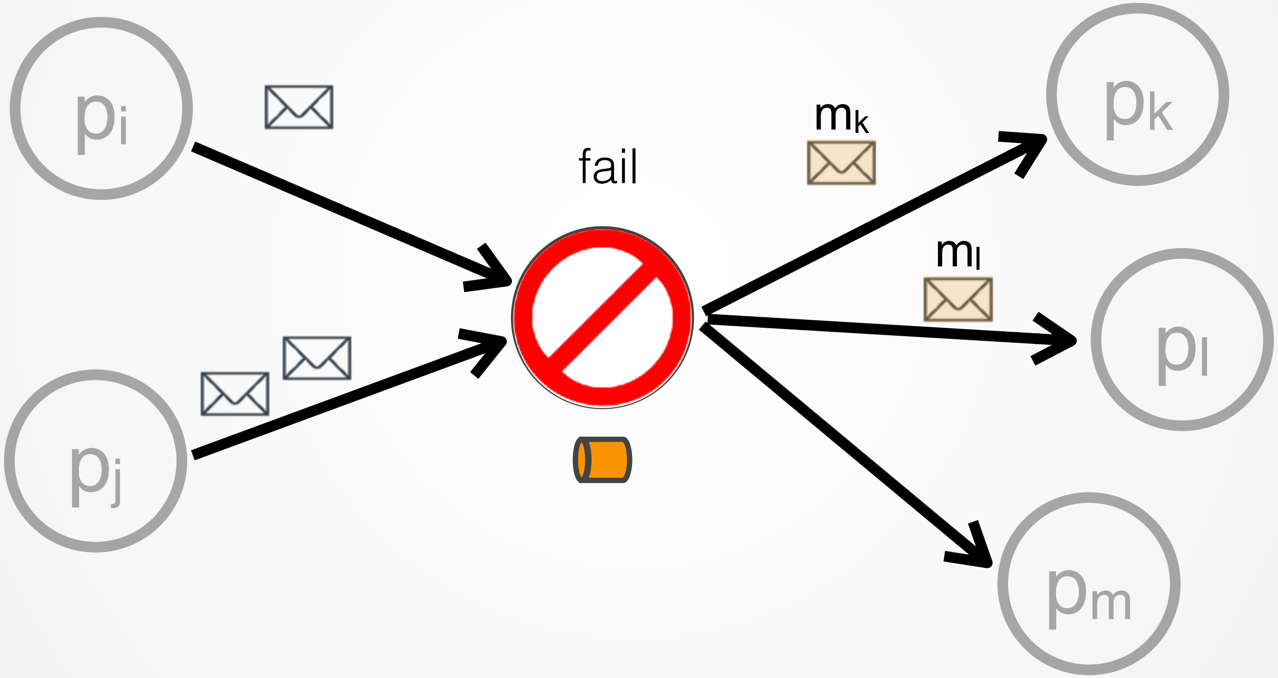
System Configurations (states, messages in-transit)

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

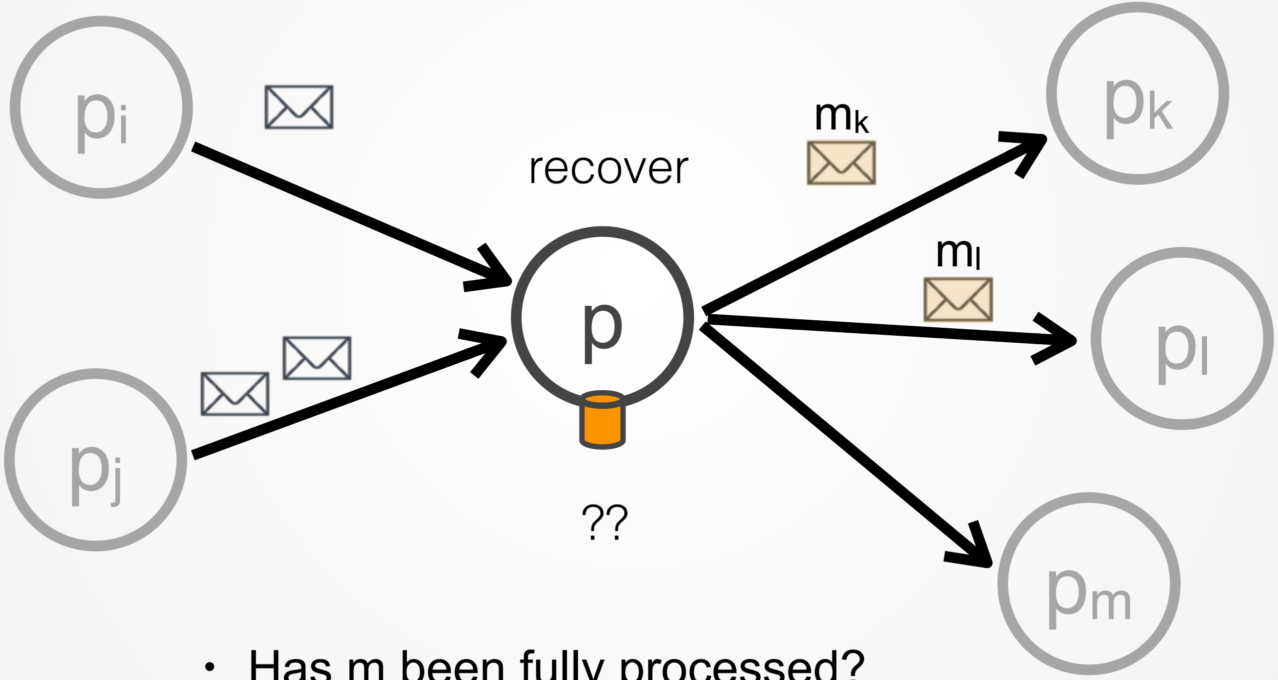
Fault Tolerance



Fault Tolerance



FAULT TOLERANCE



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

RELIABLE STREAM PROCESSING

- Past 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?

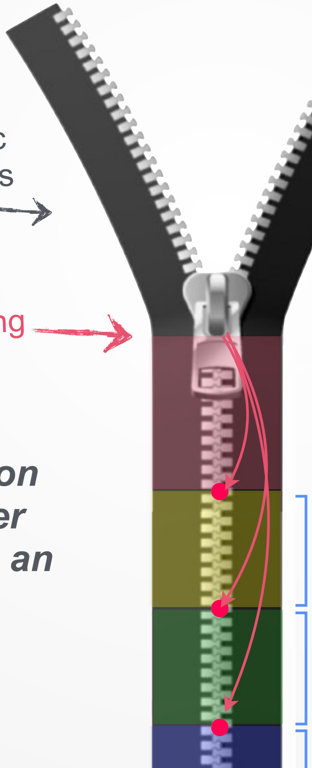
We need a system-wide coarse-grained commit mechanism.

Continuous 2PC for Data Streaming

deterministic input streams

task states

stream processing system



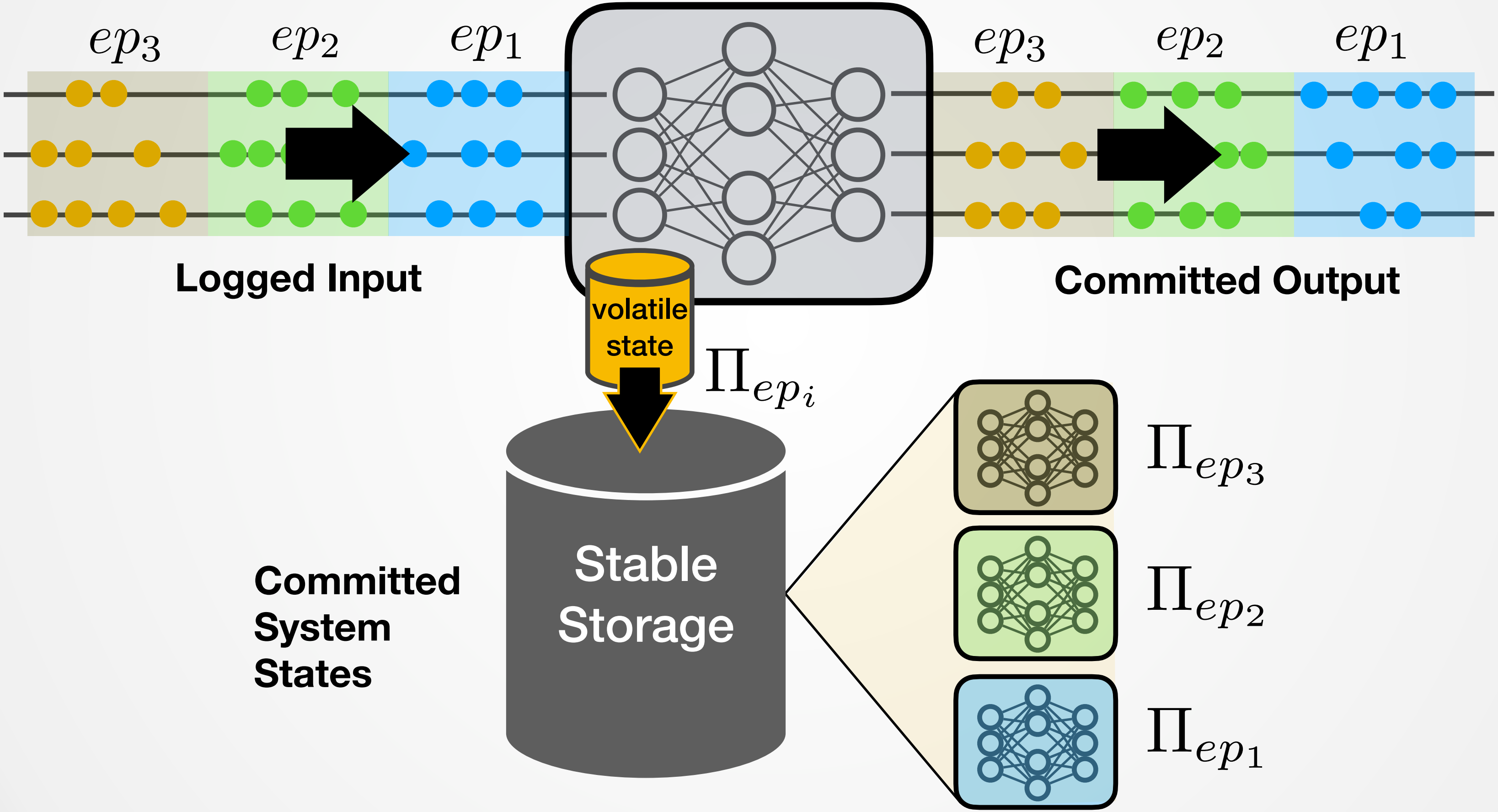
- ***system configuration (states) after completing an epoch***

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

divide computation into epochs

epoch ~ Distributed ACID Transaction

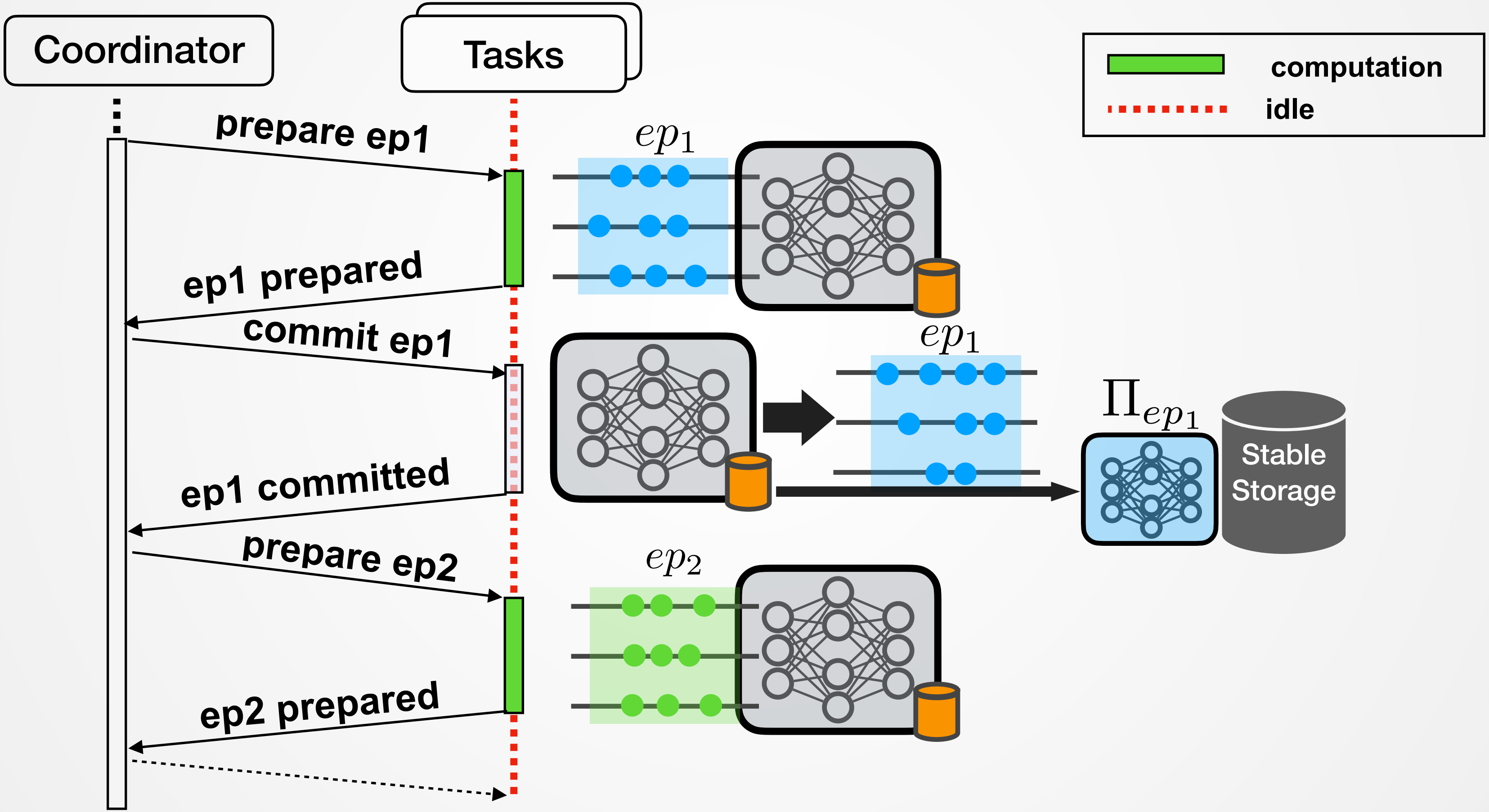
TRANSACTIONAL STREAM EXECUTION



ID2203

KTH-2023

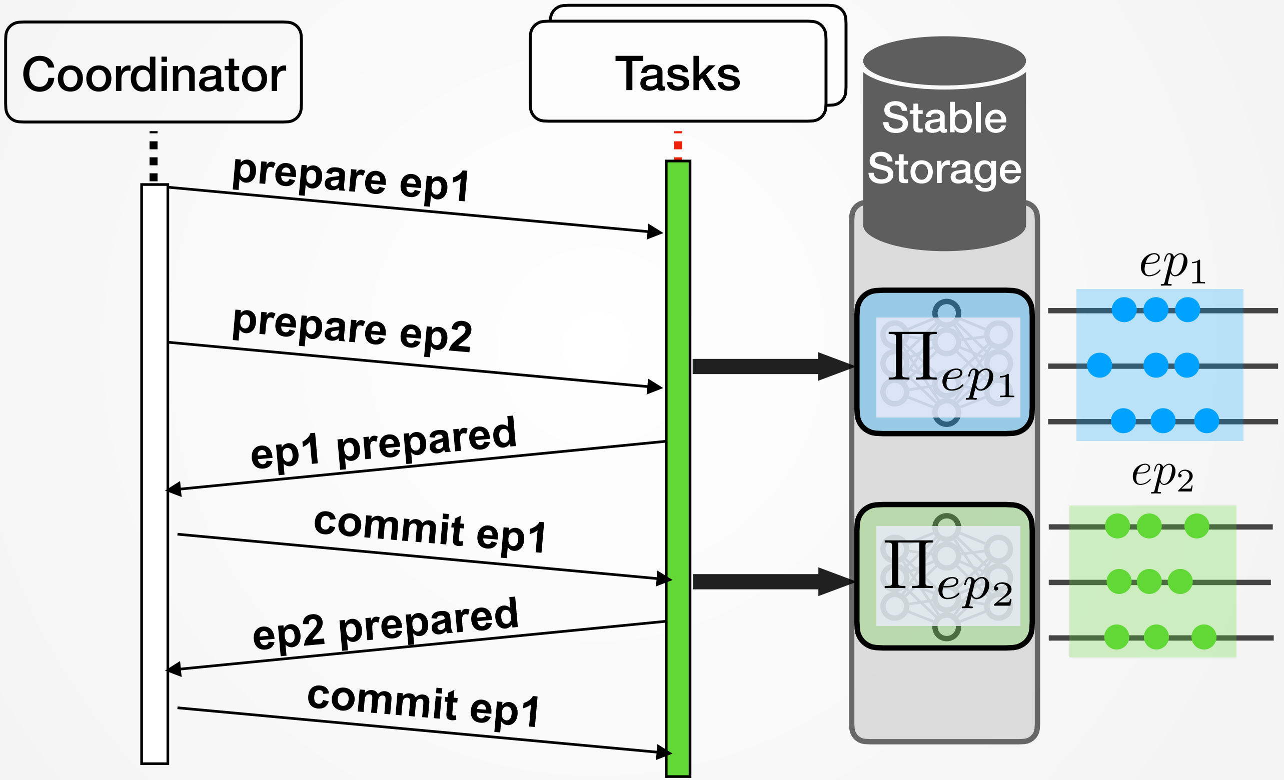
SYNCHRONOUS 2PC



SYNCHRONOUS 2PC

- 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 2PC

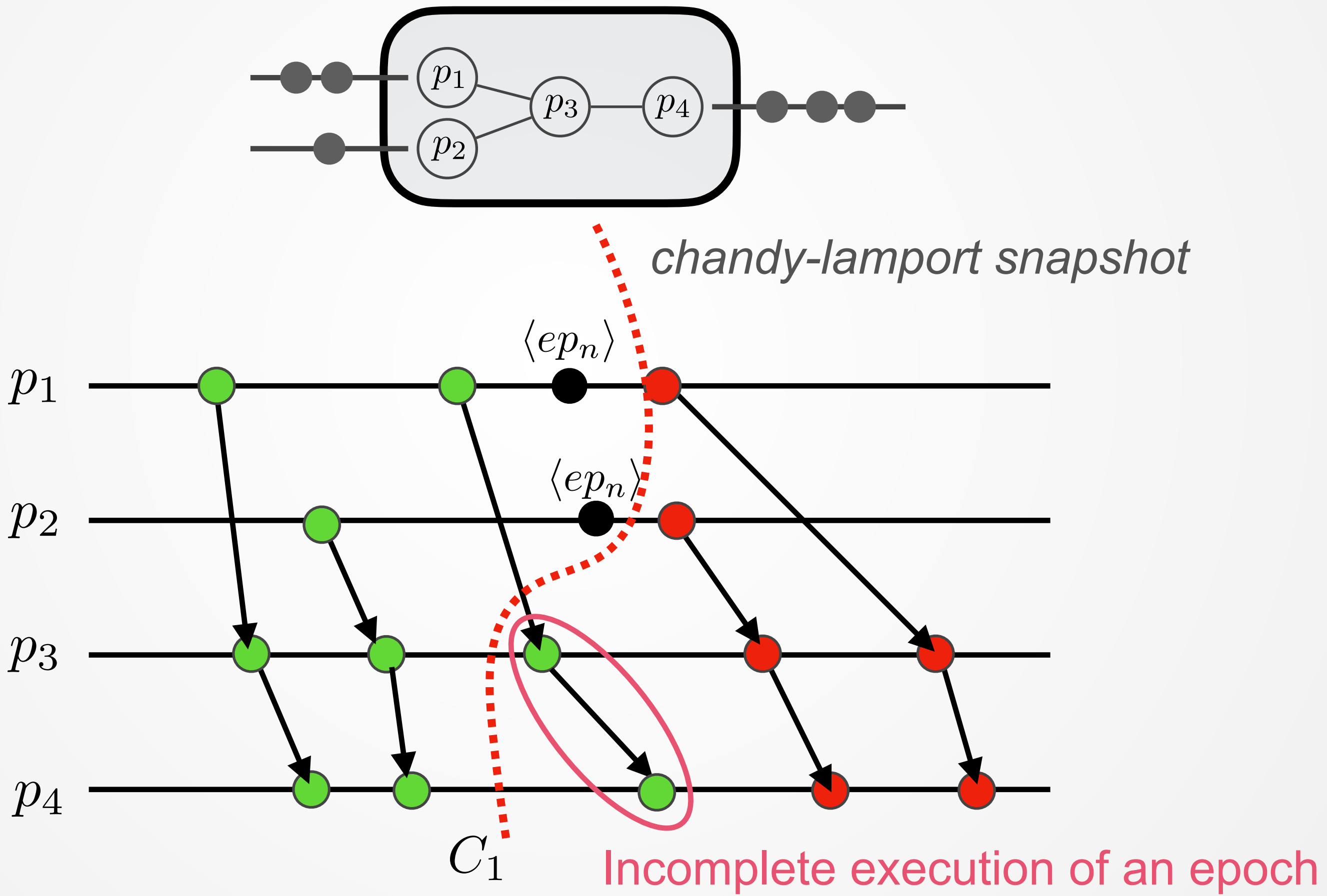


How? Using Snapshots

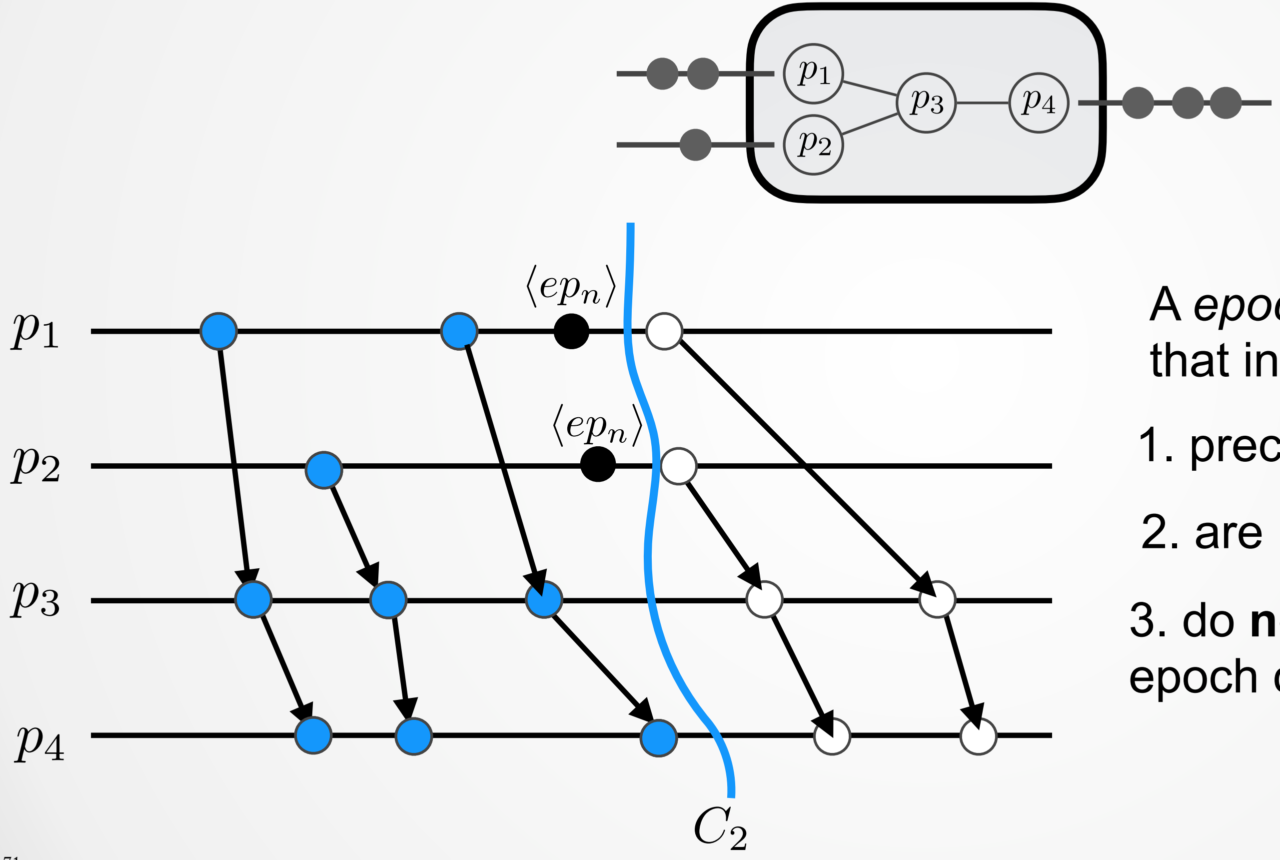
EPOCH SNAPSHOTTING

- 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



Transactional 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

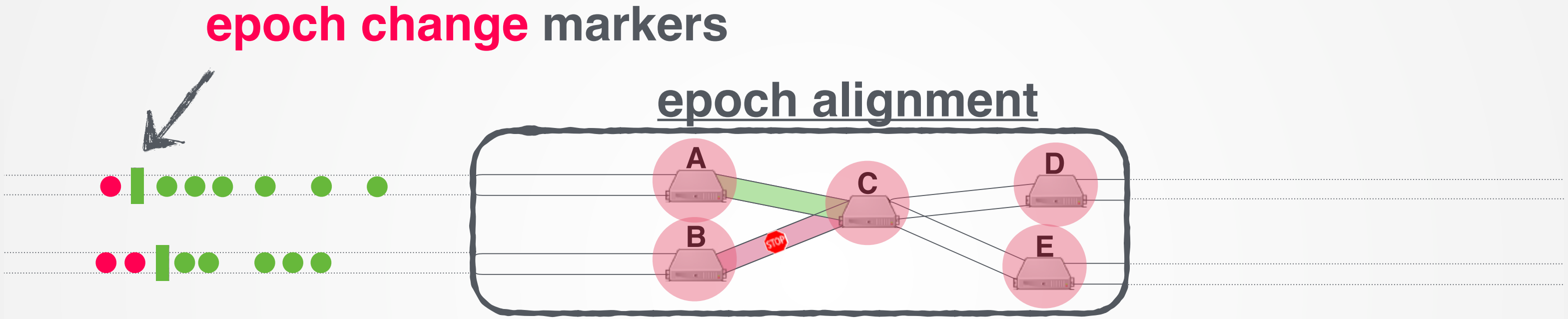
Validity (safety):

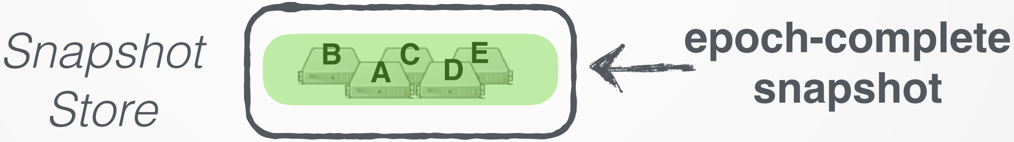
Obtain a **valid** system configuration (consistent cut)

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:

```
𝕆p ← configured_channels;
sp ← ∅;
---
/* Source Task Logic
Upon ⟨rcvd, m⟩
    (sp) ← process(sp, m, 𝕆p);
Upon ⟨ep|n⟩
    esnap → ⟨record|self, n, sp⟩;
    foreach out ∈ 𝕆p do
        out → ⟨send, ⊙n⟩;
```

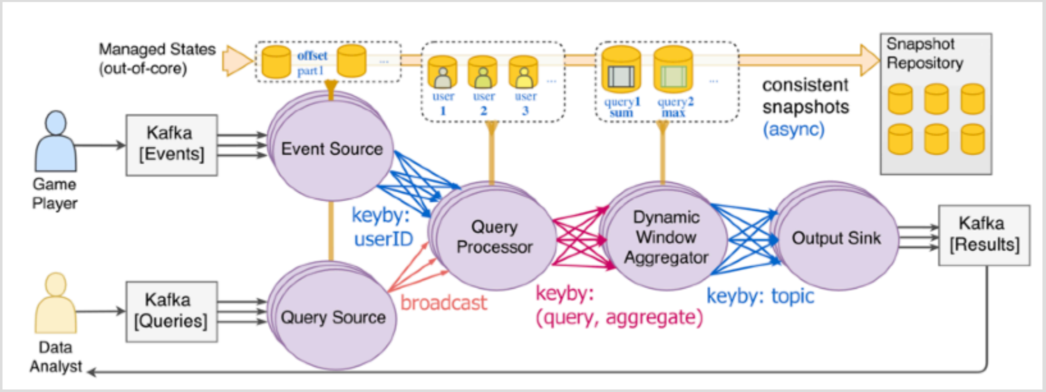
Epoch-Based Snapshots (Regular Tasks)

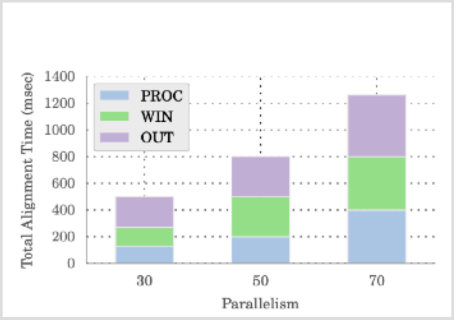
Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($\mathbb{I}_p, \mathbb{O}_p$)
Algorithm:

```
(𝕀p, 𝕆p) ← configured_channels;
Enabled ← 𝕀p ;
sp ← ∅;
---
/* Common Task Logic
Upon ⟨rcvd, m⟩ on c ∈ Enabled
    sp ← process(sp, m, 𝕆p);
Upon ⟨rcvd, ⊙n⟩ on c ∈ Enabled
    esnap → ⟨record|self, n, sp⟩;
    Enabled ← Enabled/{c};
    if Enabled = ∅ then
        foreach out ∈ 𝕆p do
            out → ⟨send, ⊙n⟩;
        Enabled ← 𝕀p ;
```

Carbone, Ewen 2015

PERFORMANCE INSIGHTS





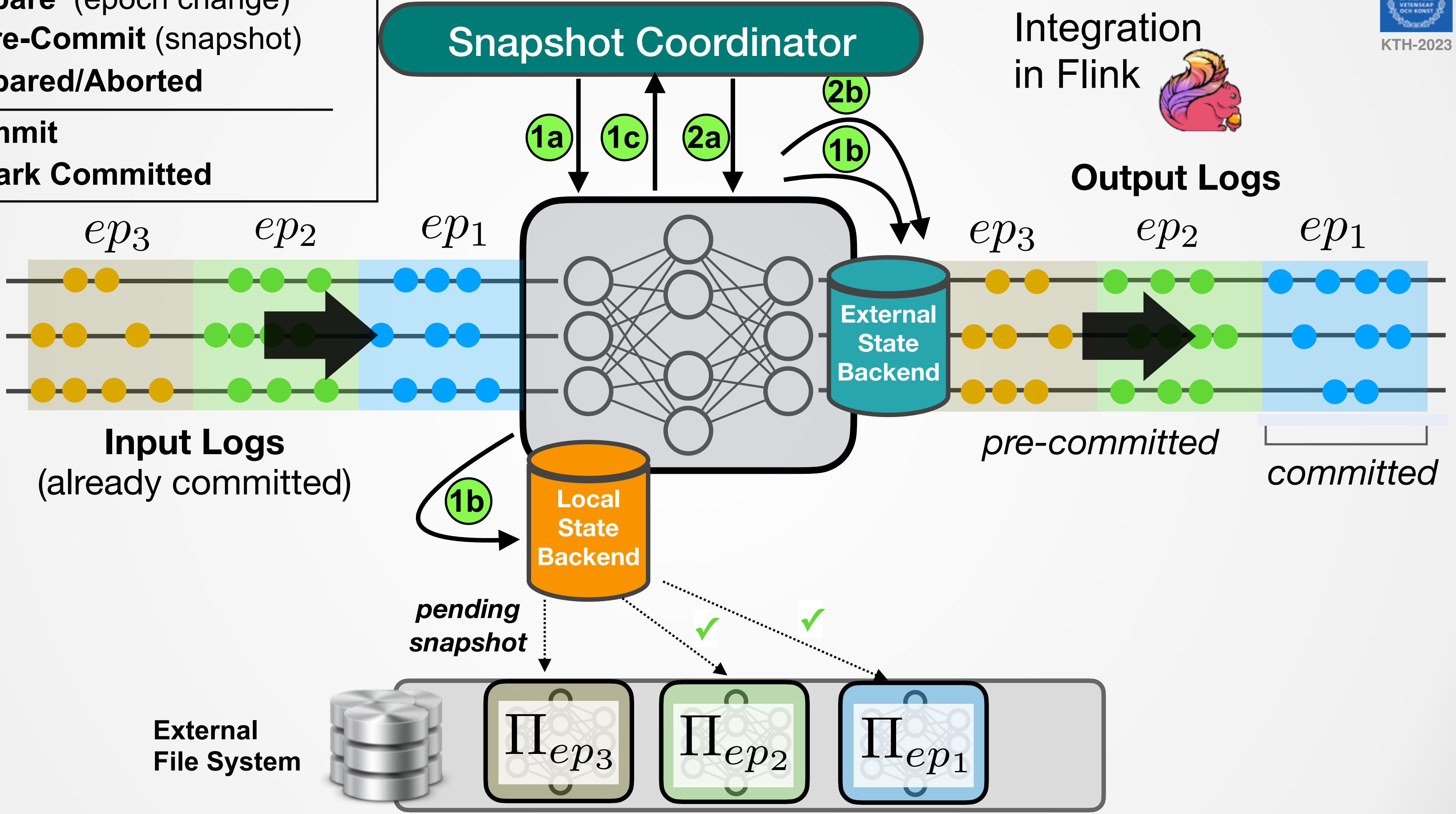
Carbone, Paris, et al. "State management in Apache Flink®: consistent stateful distributed stream processing." Proceedings of the VLDB Endowment 10.12 (2017)

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

- The Continuous Deep Analytics Team
  - https://cda-group.github.io/
- We will contact you soon for topics and internships (RISE, KTH) in
  - Distributed Algorithms
  - Distributed Data Management (Graphs, ML, Relational)
  - Data Storage Optimisation for Data Analytics