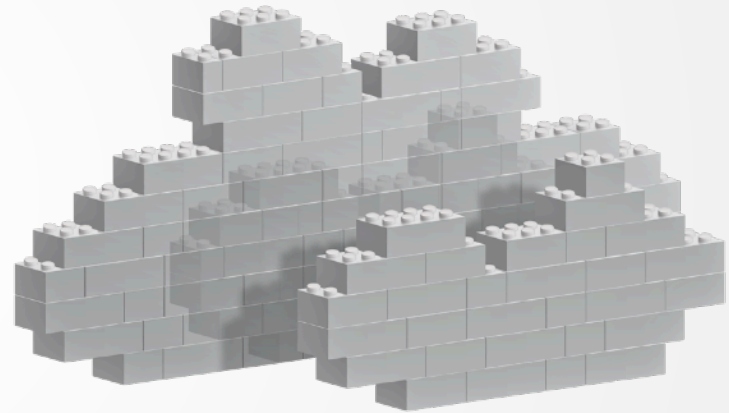


# Advanced Course Distributed Systems

## Distributed Data Management



# COURSE TOPICS

---

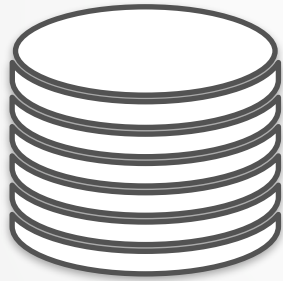


- ▶ Intro to Distributed Systems
- ▶ Basic Abstractions and Failure Detectors
- ▶ Reliable and Causal Order Broadcast
- ▶ Distributed Shared Memory
- ▶ Consensus (Paxos, Raft, etc.)
- ▶ Replicated State Machines + Virtual Logs
- ▶ Advanced Time Abstractions (Spanner etc.)
- ▶ Distributed ACID Transactions (Cloud DBs)
- ▶ Consistent Snapshotting (Stream Data Management)

# WHY DO WE NEED DISTRIBUTED SYSTEMS AGAIN?

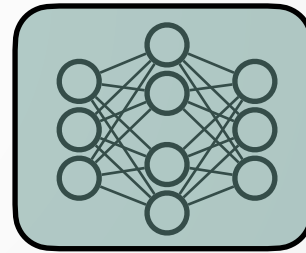
---

- 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

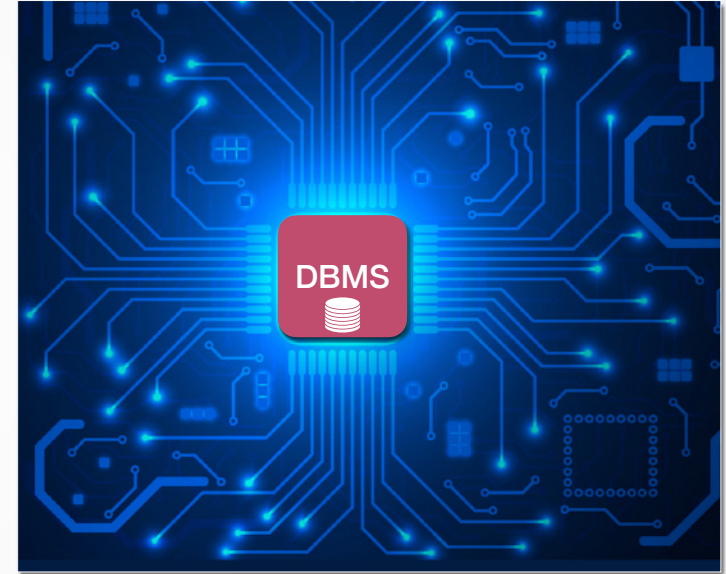
# Distributed ACID Databases

# 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

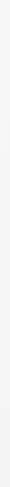
**T1:** We want to transfer **100sek** from **X** to **Q**.

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



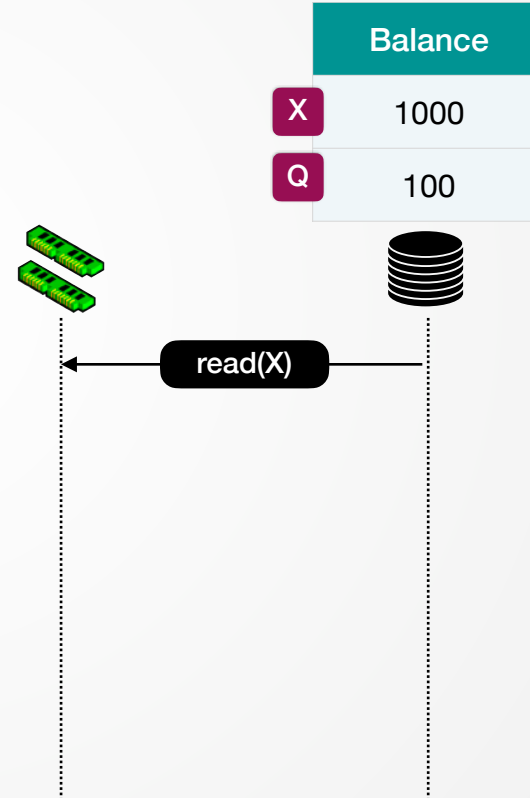
# ANATOMY OF A TRANSACTION

## 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)



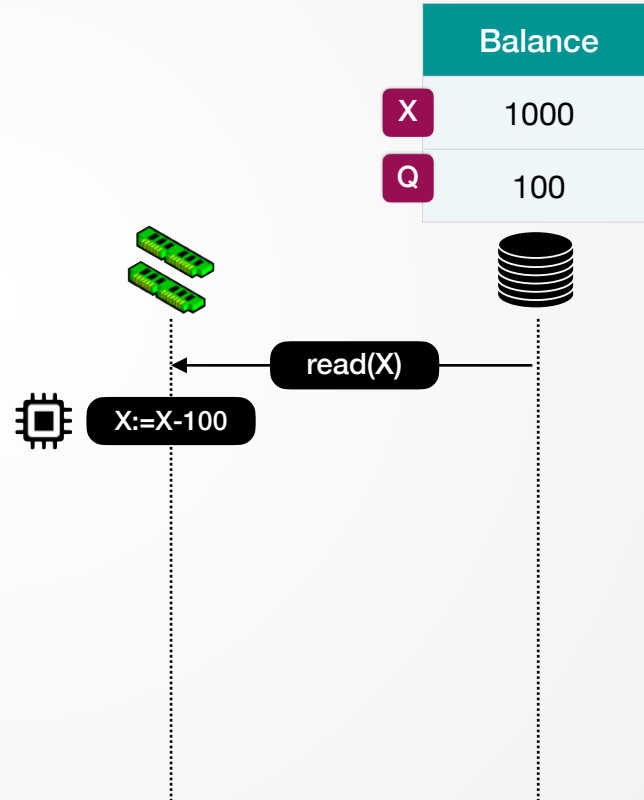
# ANATOMY OF A TRANSACTION

## 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)





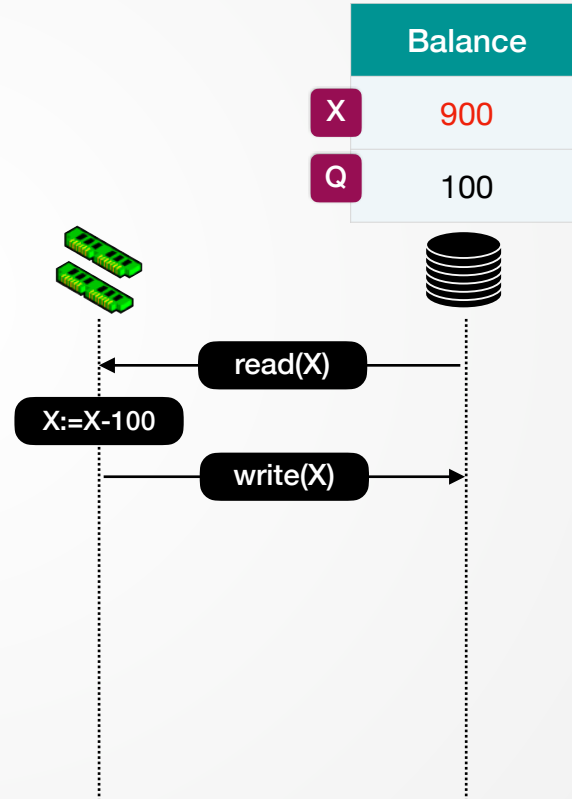
# ANATOMY OF A TRANSACTION

## 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)



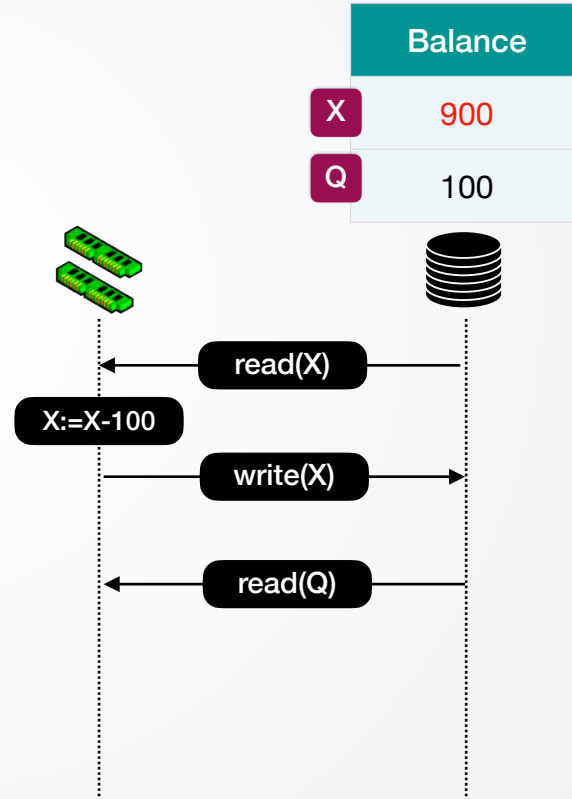
# ANATOMY OF A TRANSACTION

## 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)



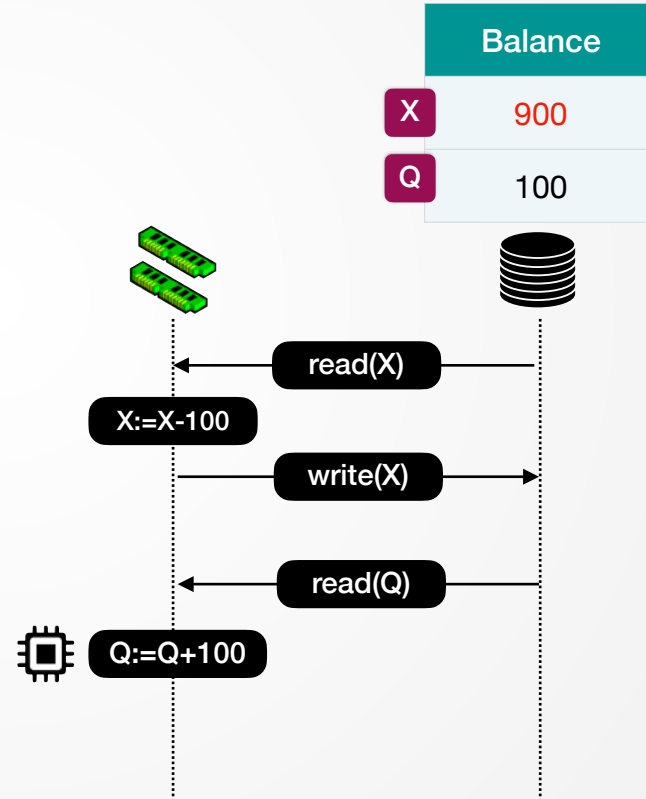
# ANATOMY OF A TRANSACTION

## 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)



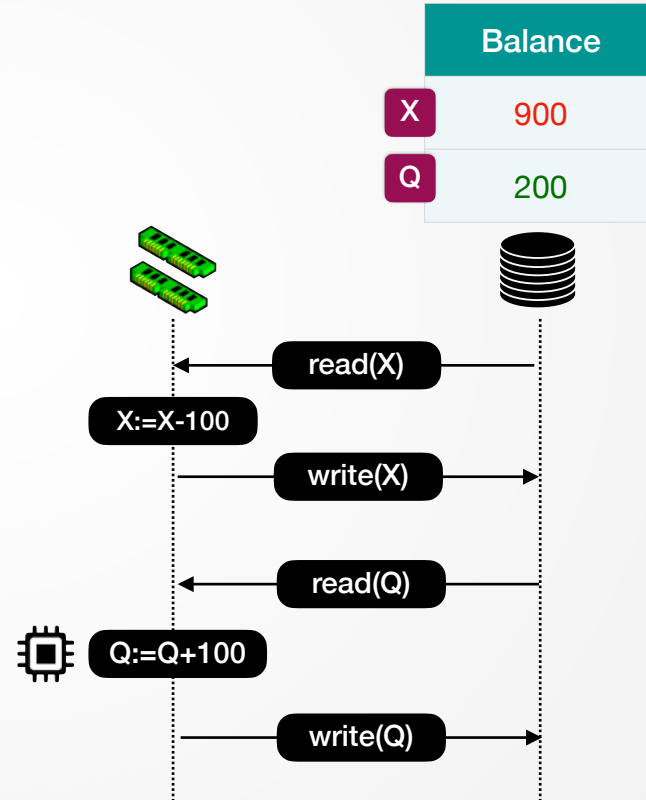
# ANATOMY OF A TRANSACTION

## Classic Example

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

That involves the following operations:

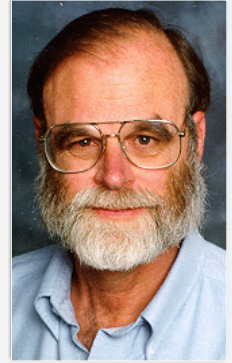
1. read(X)
2.  $X := X - 100$
3. write(X)
4. read(Q)
5.  $Q := Q + 100$
6. write(Q)



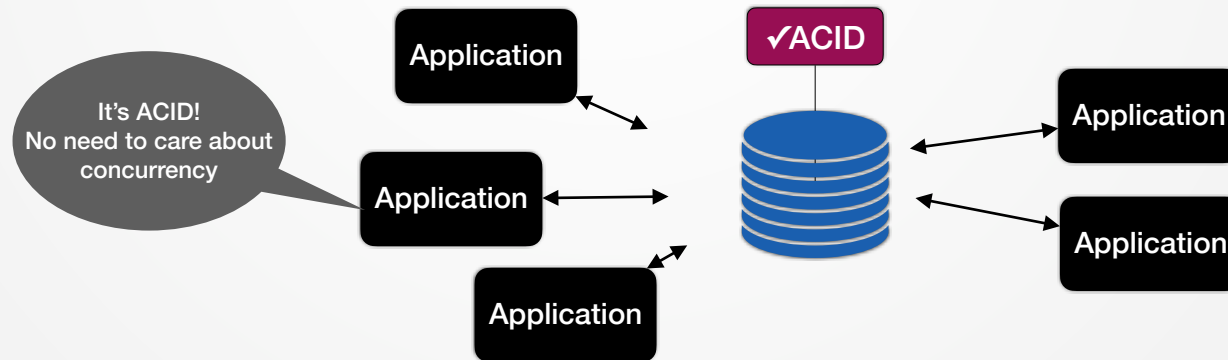
# ACID

## The core 4 properties for Transactions

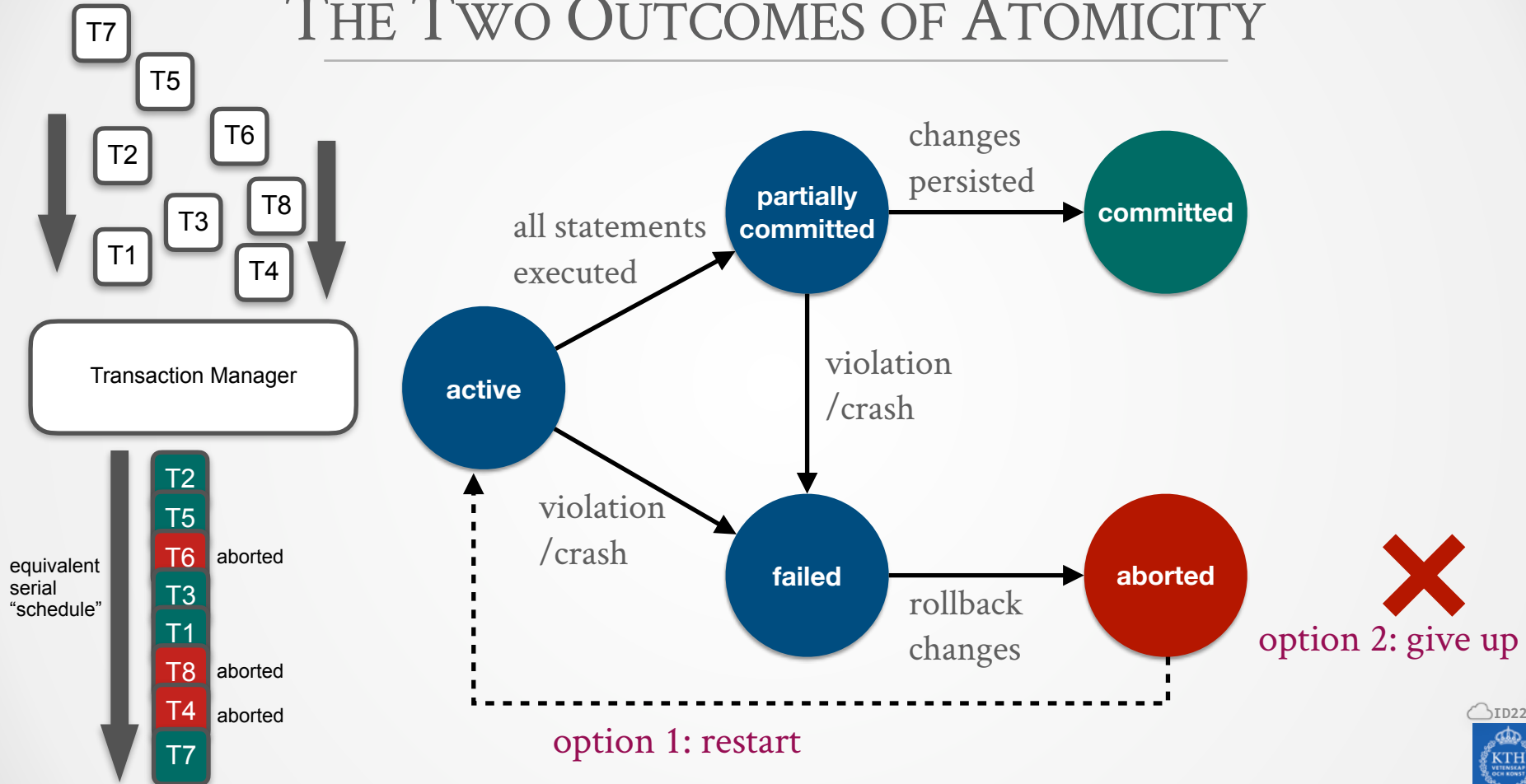
- ▶ **Atomicity**: “all transaction commands are committed or none”
- ▶ **Durability**: “all transaction object updates are persisted”
- ▶ **Isolation**: “transactions do not ‘compete’ but are isolated”
- ▶ **Consistency**: “no relational model/constraint violations”



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:**

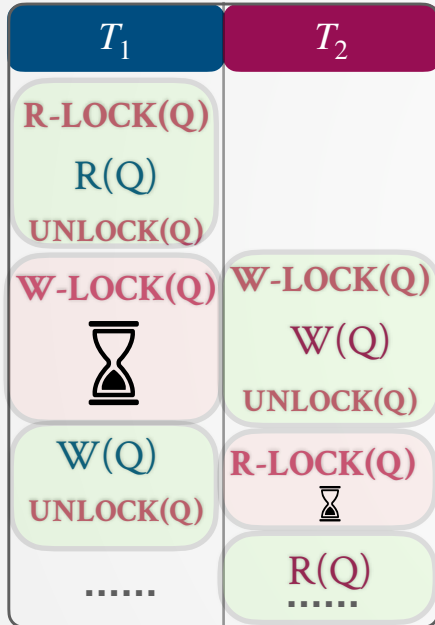
persistent storage

+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)

---

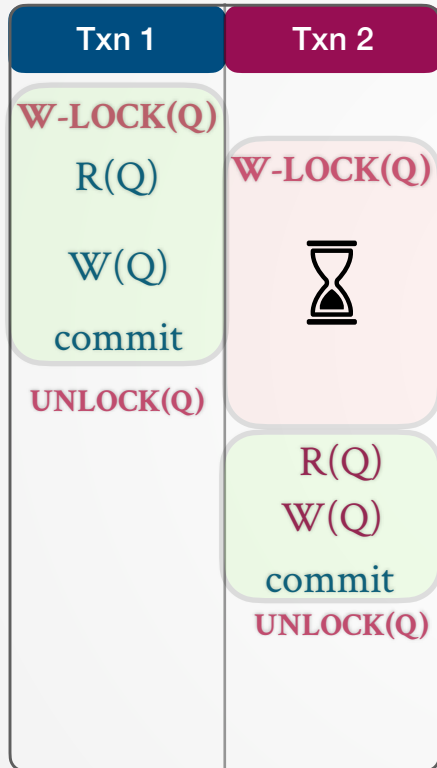
Each transaction should acquire all necessary locks first and then release them.

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

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

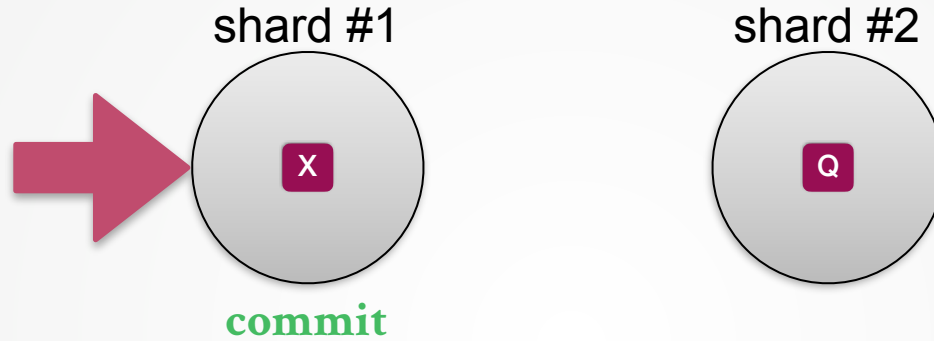
**Core invariant:** never acquire any lock after a lock has been released.

# STRONG STRICT 2PL EXAMPLE



# DISTRIBUTED ACID

---

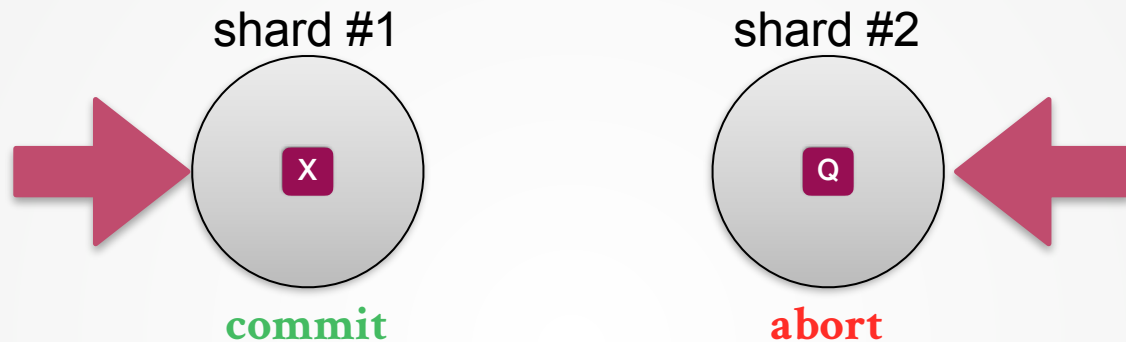


Transaction  $T_1$

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

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

# 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

---

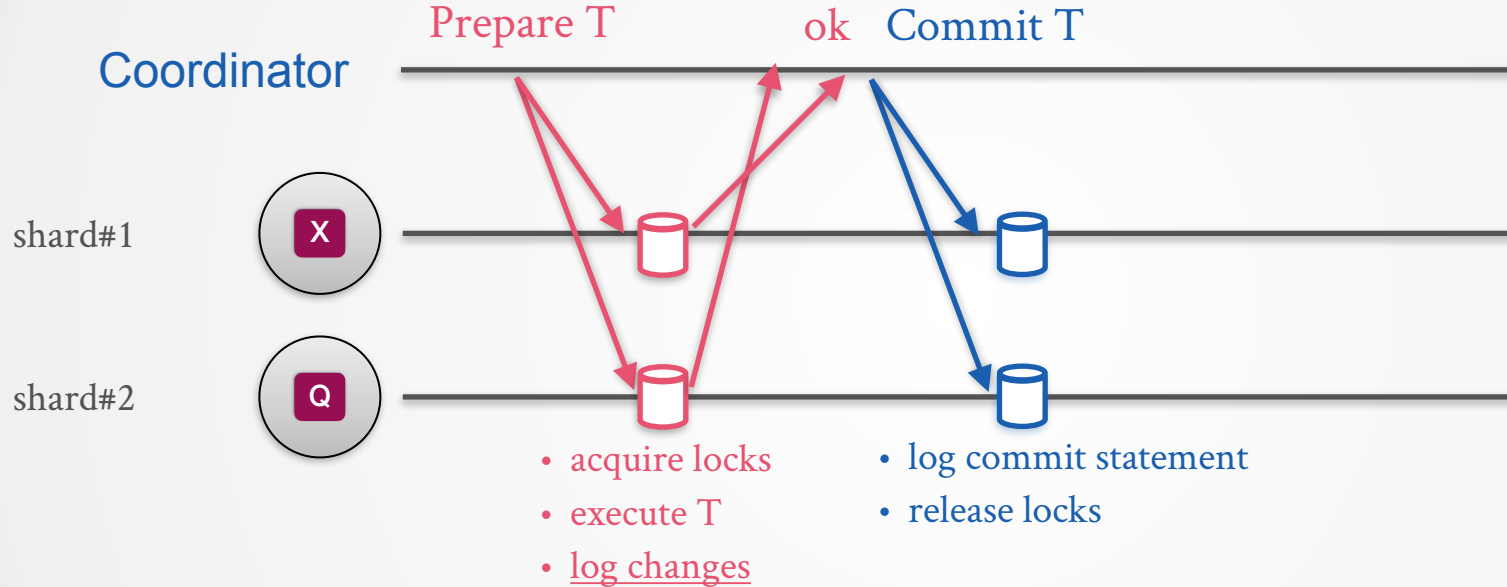
- Transaction Coordinator (leader)
- Cohorts (followers)
  - **Request:** Transaction T
  - **Indication:** Commit | Abort
- Given a proposed transaction T
  - **Commit** if **all** followers agree to commit
  - **Abort** if **at least one** follower aborts or fails

# ATOMIC COMMIT VS CONSENSUS(PAXOS)

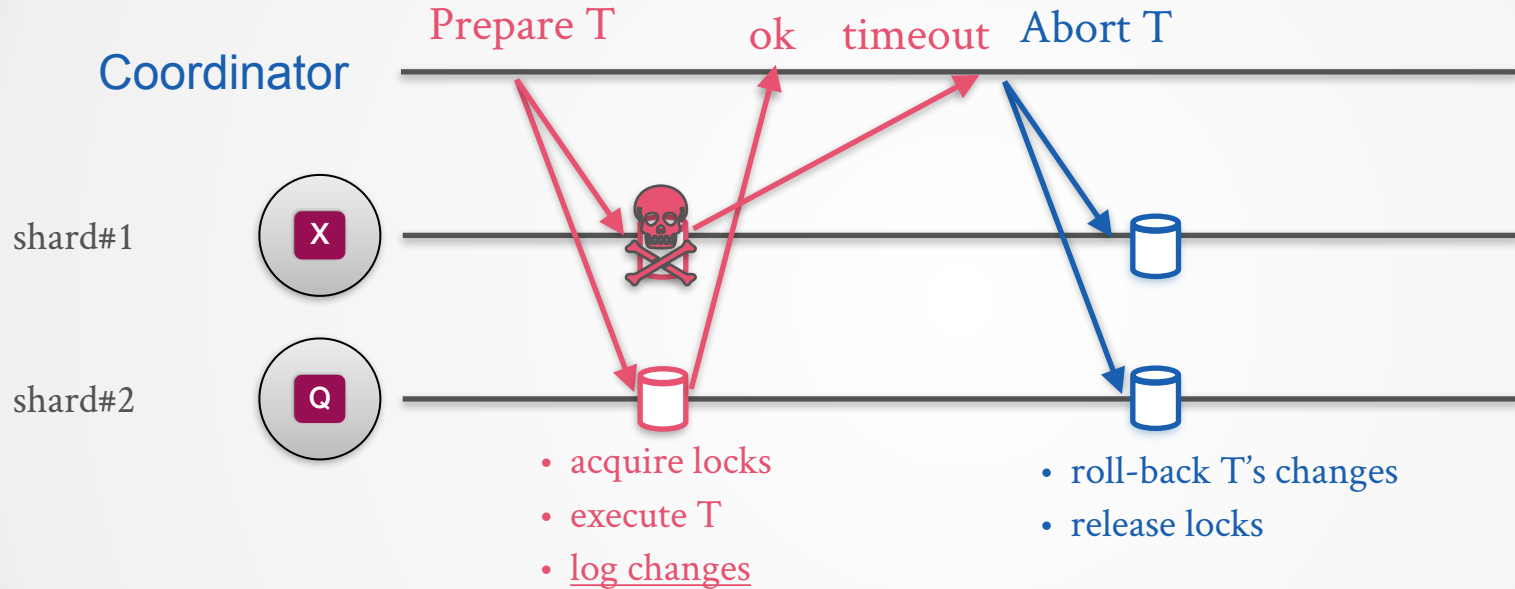
Validity	Decide Commit or Abort	Decide any Proposed Value
Fault Tolerance	$f = 0$ (but can be improved)	$f < N/2$
Leader	Single Coordinator Process	Any process can propose
Agreement	Unanimous	Quorum-based

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

# 2PC (TWO PHASE COMMIT)

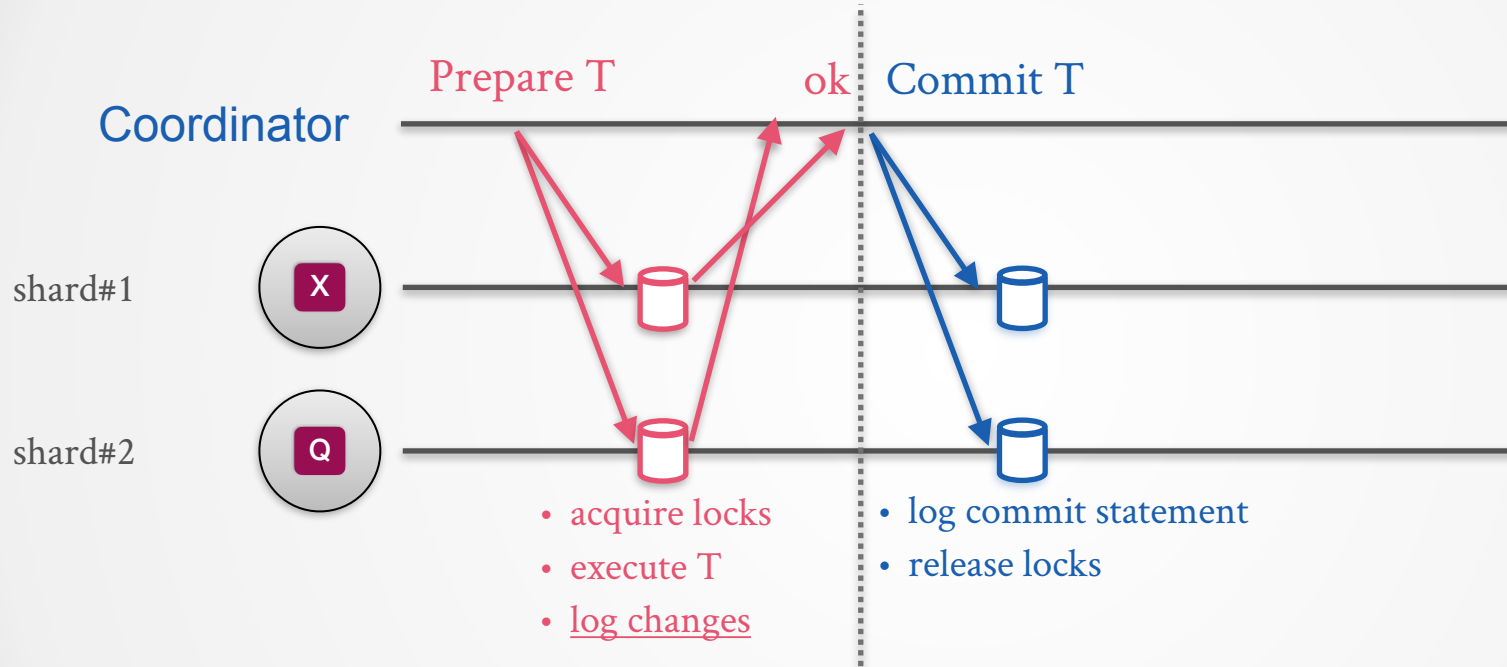


# 2PC (TWO PHASE COMMIT)





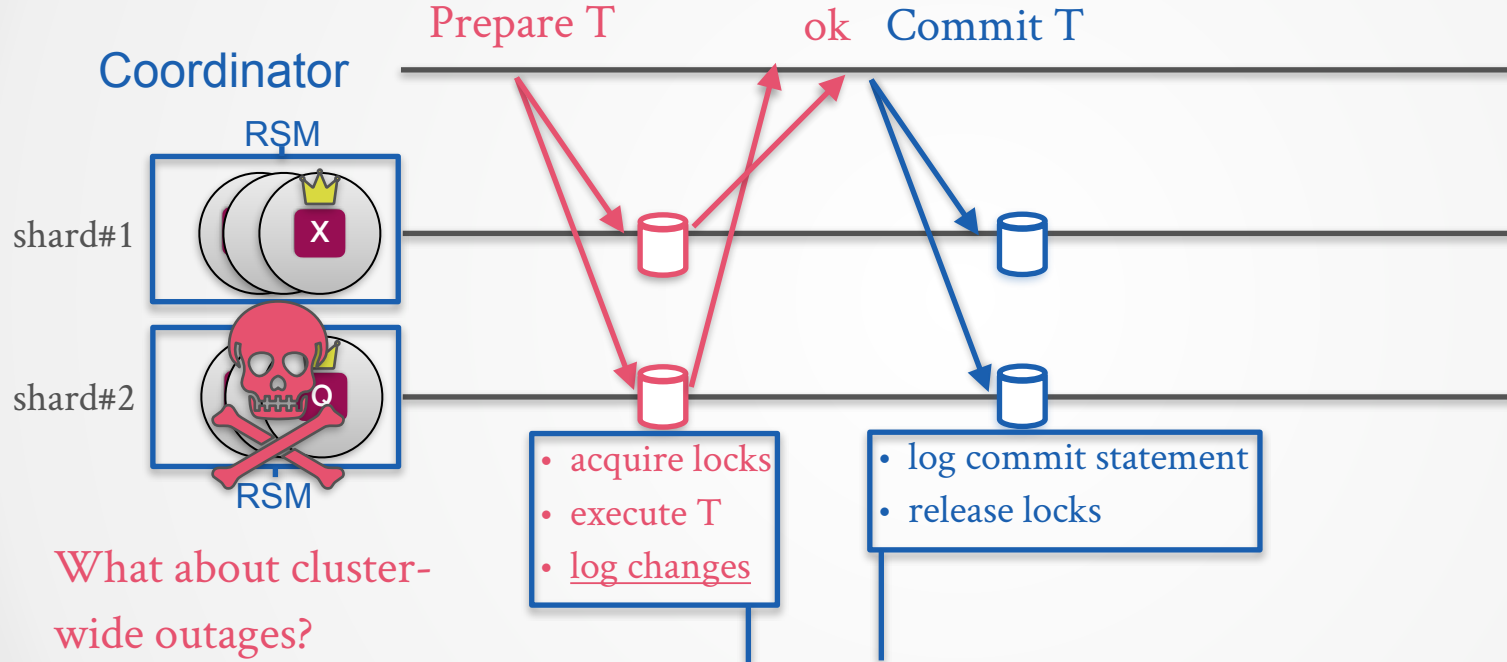
# 2PC (TWO PHASE COMMIT)



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)

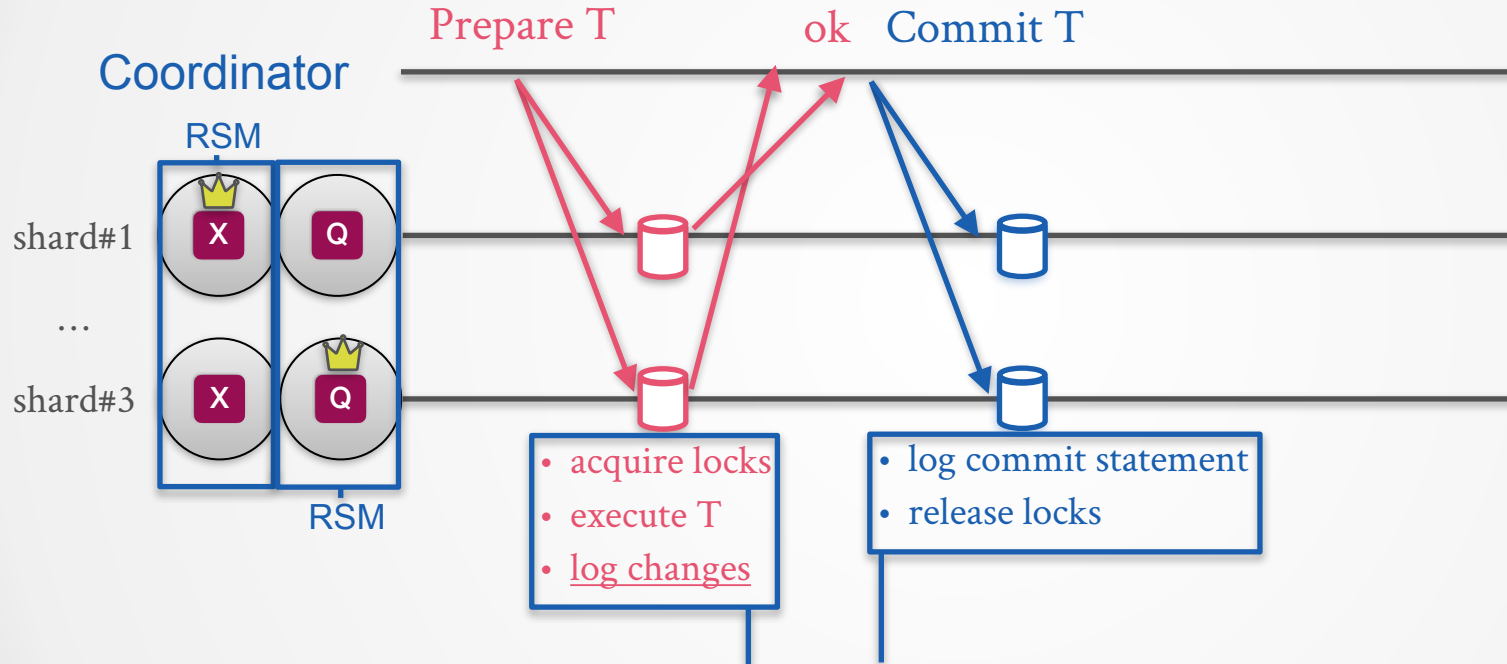


What about cluster-wide outages?

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)

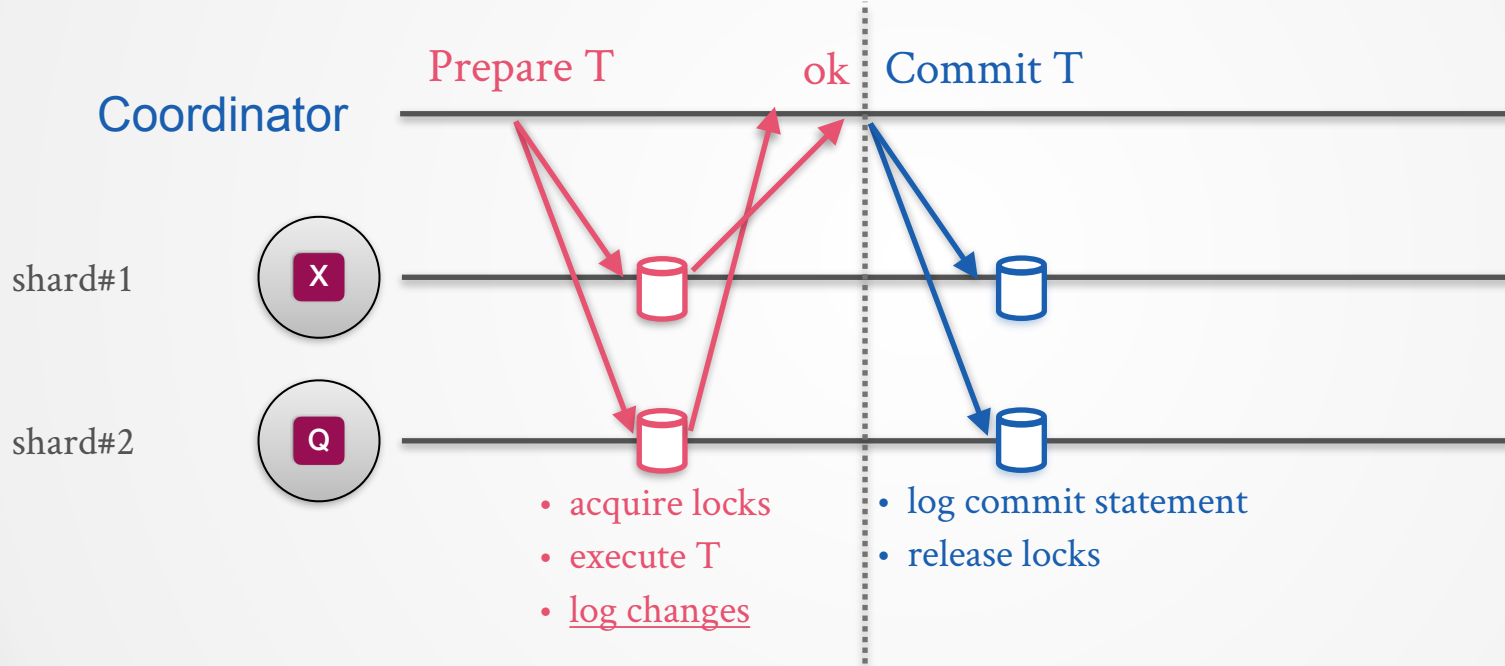


• i.e., Google Spanner geo-replication

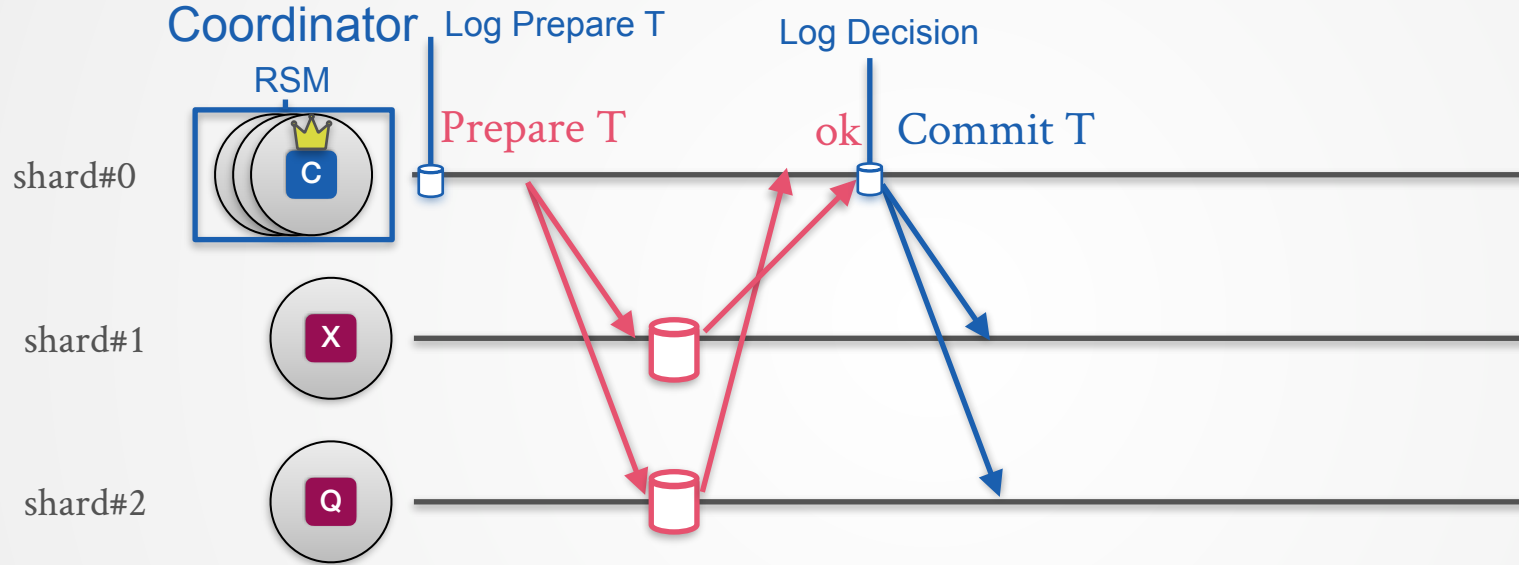
- All 2PC actions are decided commands in each shard's RSM
- Protocol is executed by each respected leader shard and replicated to other shards

# 2PC COORDINATOR CRASHES

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

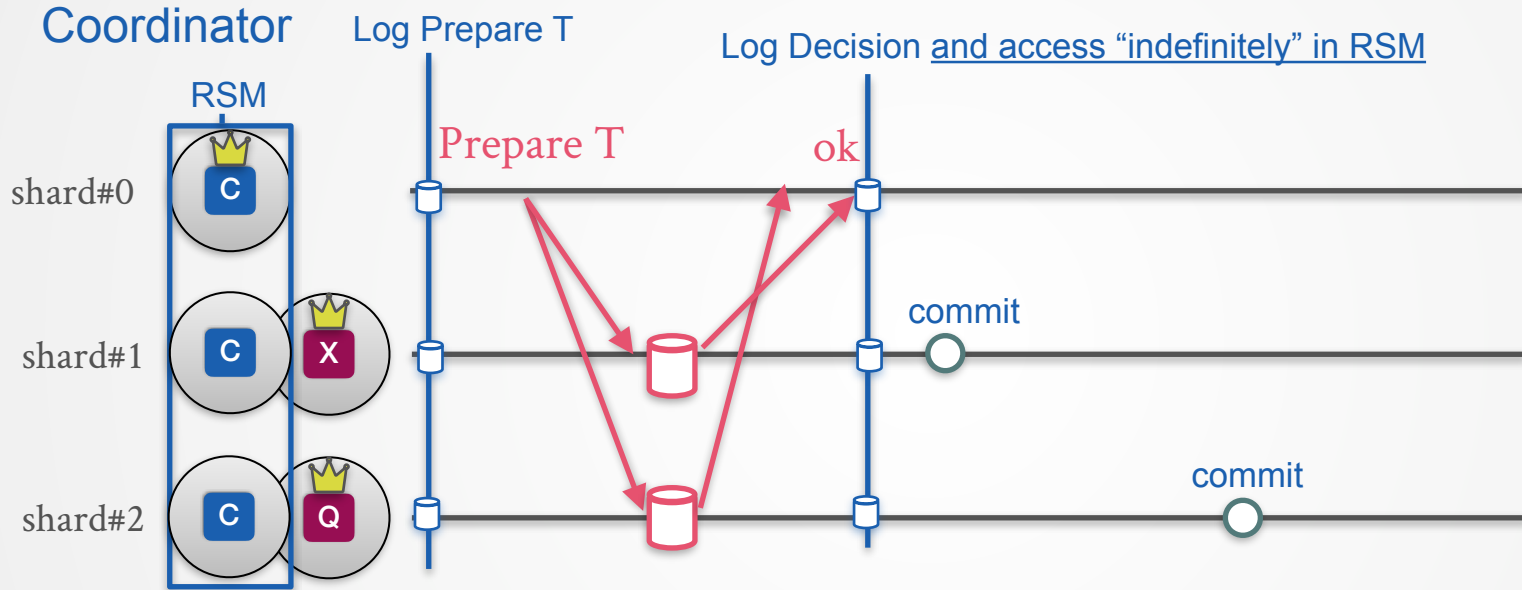


# RELIABLE 2PC V.1



- 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)

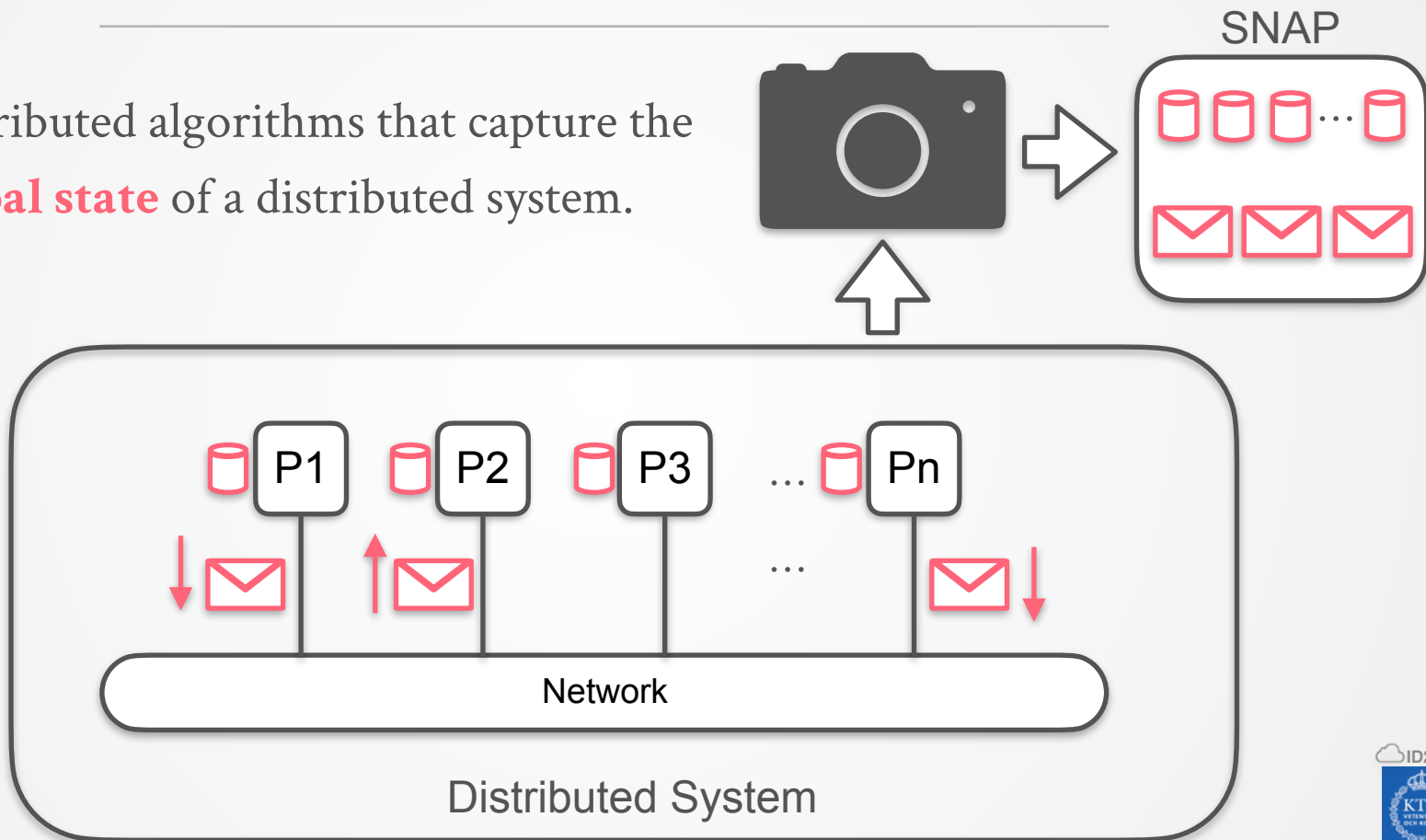


- This approach ensures that Transaction Decisions are reliably replicated across shards.
- All servers can apply finalize (rollbacks/commits) based on transaction status read from local RSM replica (Zookeeper, Raft, OmniPaxos)

# Distributed Data Processing and Snapshots

# DISTRIBUTED SNAPSHOTS

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

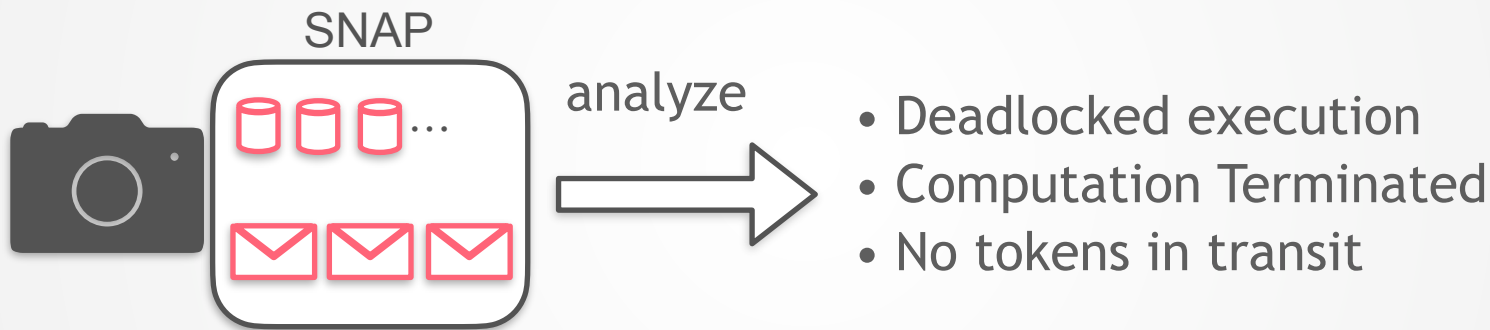




# SNAPSHOT USAGES

---

## 1. Stable Property Detection

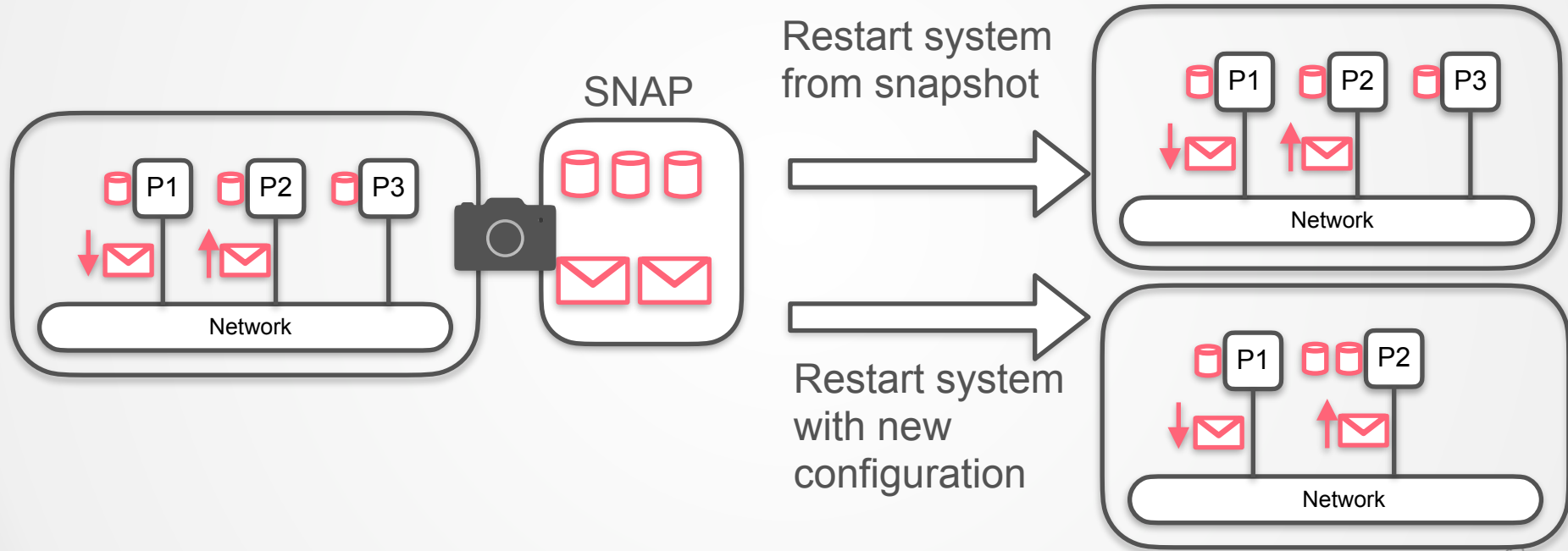


**“A stable property is one that persists: once a stable property becomes true it remains true thereafter”**

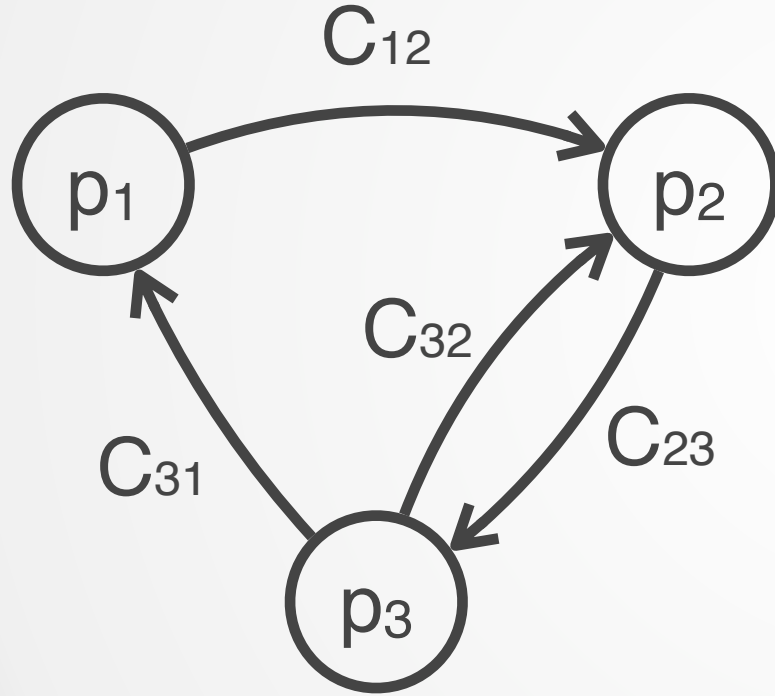
**- Chandy, Lamport 88**

# 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  $I_p$ :
  - ▶  $S'_p = \text{process}(m, S_p, O_p)$
  - ▶ 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 LAMPART  
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, deadlock, multiprocessor/multiprogramming, mutual exclusion; scheduling; synchronization; D.4.6 [Operating Systems]: Reliability—backup procedures; checkpoints; testing; 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

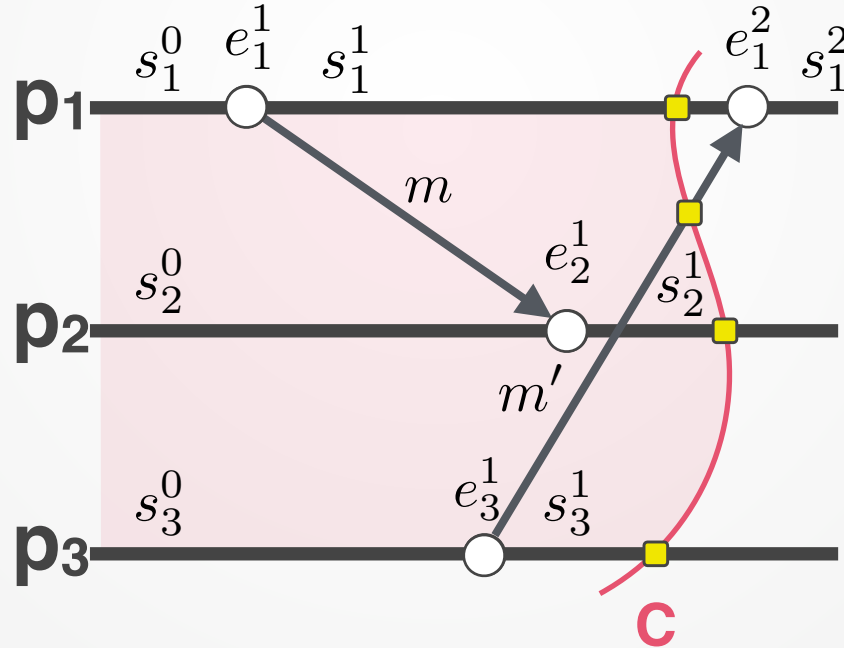
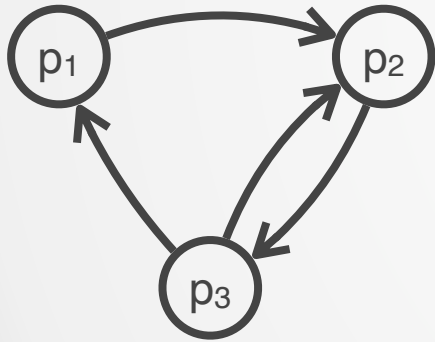
This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR 83-0235 and in part by the National Science Foundation under Grant MCS 81-04458. Authors' addresses: K. M. Chandy, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. Lamport, 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 0734-0017/85/0000-0000 \$00.75

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

# DISTRIBUTED CUTS

- ▶ A snapshot implements a cut  $\mathbf{C}$  of an execution (prefix) and returns the system's corresponding states/configuration.

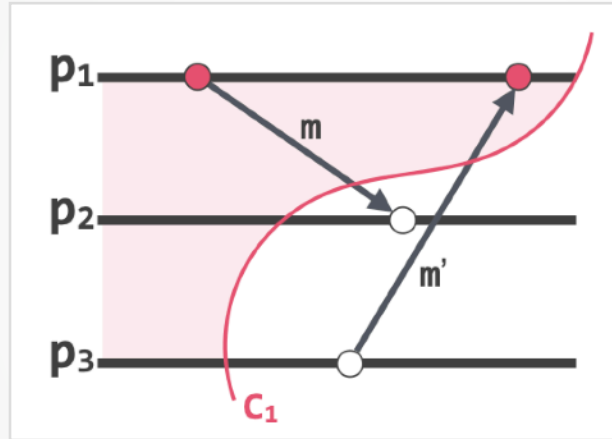
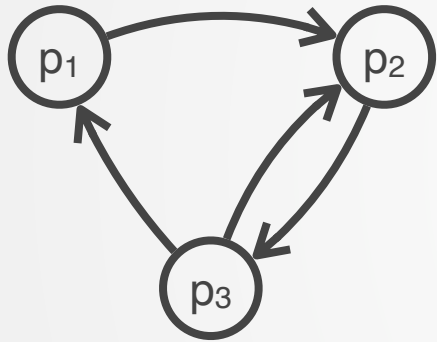


## Snapshot of $\mathbf{C}$

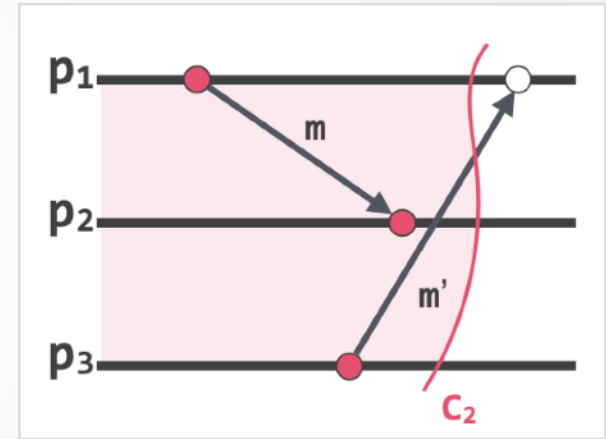
$\{s_1^1, s_2^1, s_3^1\}$   
 $\{m'\}$

# 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

---

## *Events*

**Request:**  $\langle \text{snapshot} \rangle$

**Indication:**  $\langle \text{record} \mid p, [S_p, M_p] \rangle$

$S_p$ : state of  $p$

$M_p$ : messages in  $I_p$

## *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  $\odot$  to mark events before and after the consistent cut.

# THE ALGORITHM

## Chandy-Lamport Consistent Snapshots

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

```

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

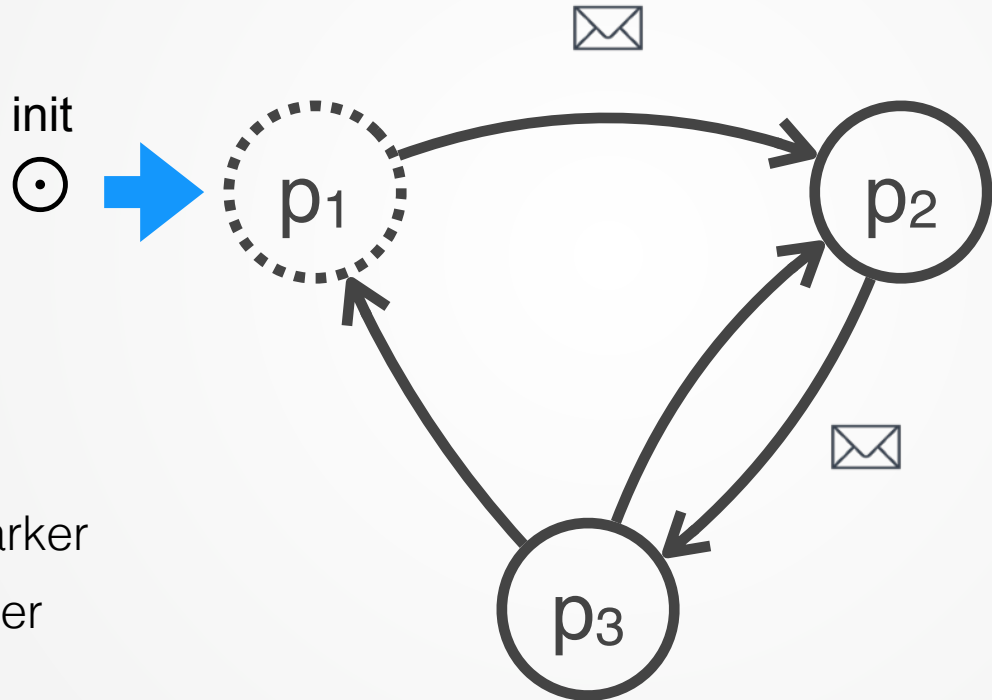
5: Upon (rcvd, m) on  $c_{qp} \notin \text{Recorded}$ ,  $m \neq \odot$ 
6:    $s_p \leftarrow \text{process}(m, s_p, \mathbb{O}_p)$ ; ▷ regular process logic
7: Upon (rcvd, m) on  $c_{qp} \in \text{Recorded}$ ,  $m \neq \odot$ 
8:    $M_p \leftarrow M_p \cup \{m\}$ ; ▷ record in-transit message
9:    $s_p \leftarrow \text{process}(m, s_p, \mathbb{O}_p)$ ;
10: Upon (rcvd,  $\odot$ ) on  $c_{qp} \in \mathbb{I}_p$ 
11:   if  $s_p^* = \text{empty}$  then
12:     startRecording();
13:   Recorded = Recorded  $\cup \{c_{qp}\}$ ;
14:   if Recorded =  $\emptyset$  then
15:     csnap  $\rightarrow \langle \text{record|self}, s_p^*, M_p \rangle$ ;

16: Upon (snapshot) on csnap
17:   startRecording();
18:   if Recorded =  $\emptyset$  then
19:     csnap  $\rightarrow \langle \text{record|self}, s_p, \emptyset \rangle$ ;

20: Fun startRecording()
21:    $s_p^* \leftarrow s_p$ ; ▷ record local state
22:   foreach out  $\in \mathbb{O}_p$  do
23:     out  $\rightarrow \langle \text{send}, \odot \rangle$ ;
24:   Recorded  $\leftarrow \mathbb{I}_p$ 

```

# EXAMPLE EXECUTION

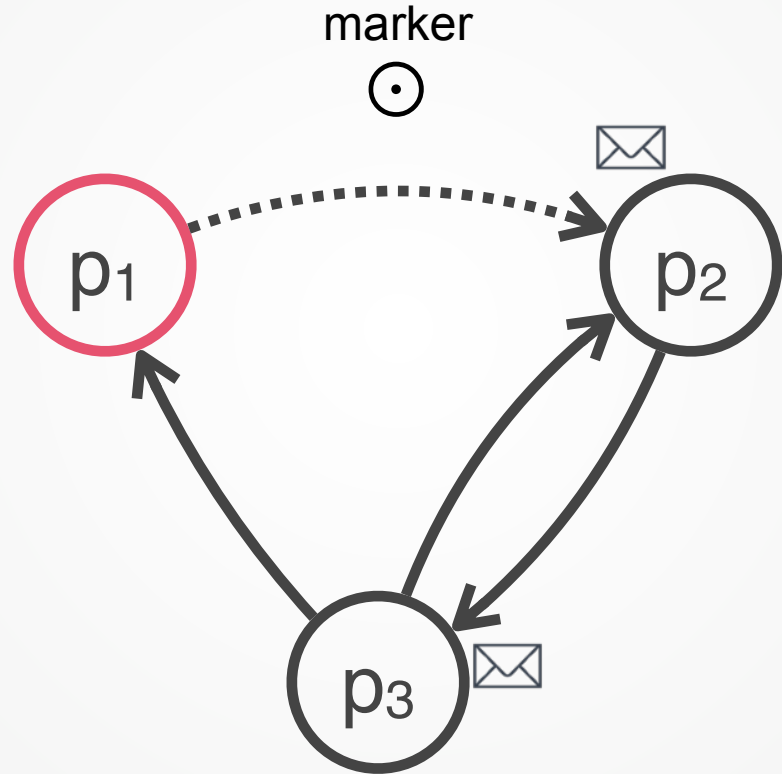


Snapshot

**s1**

before marker  
after marker

# EXAMPLE EXECUTION



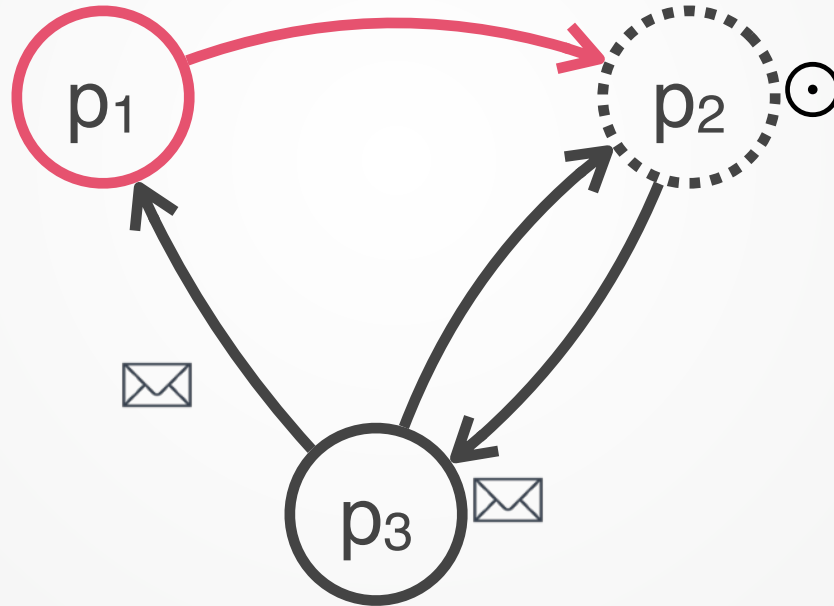
Snapshot

s1

before marker

after marker

# EXAMPLE EXECUTION



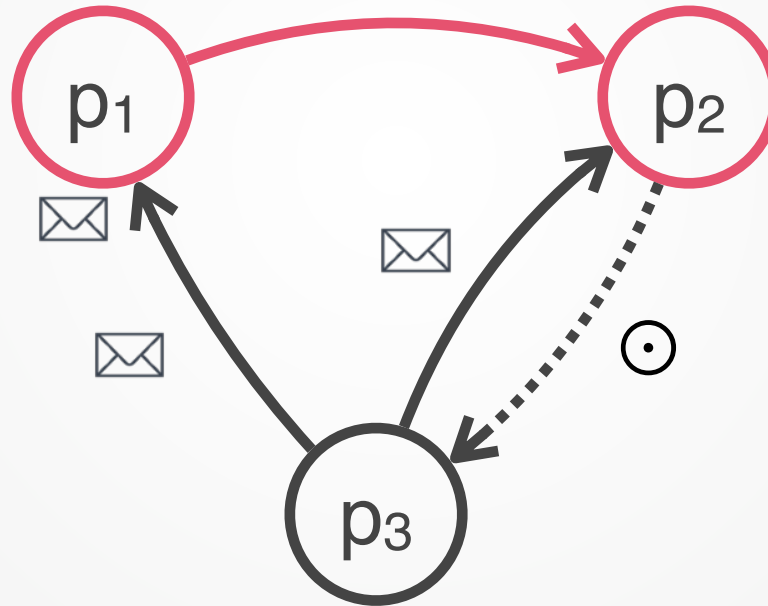
Snapshot

**s1, s2**

before marker

after marker

# EXAMPLE EXECUTION



Snapshot

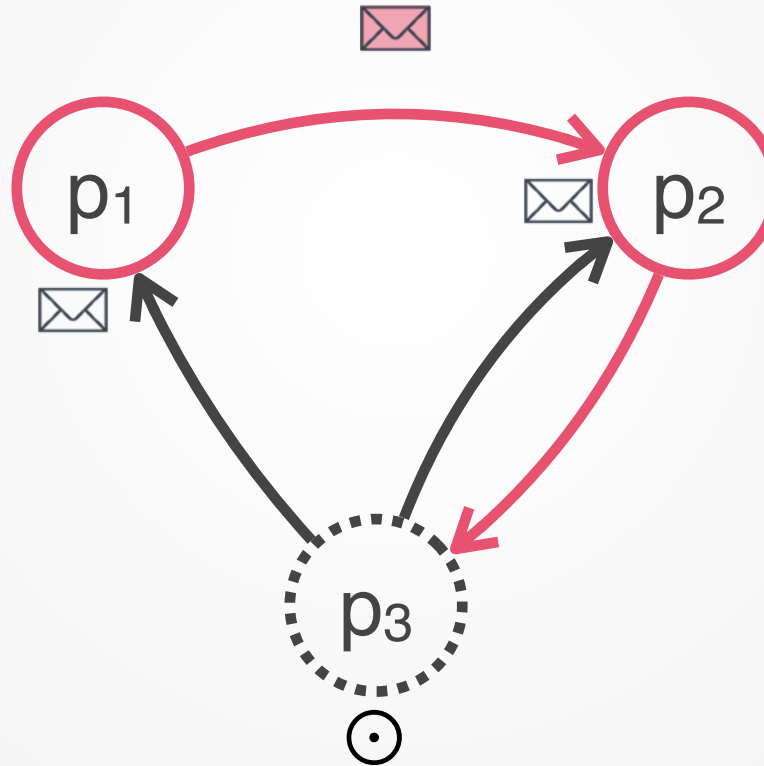
**s1, s2**



before marker

after marker

# EXAMPLE EXECUTION



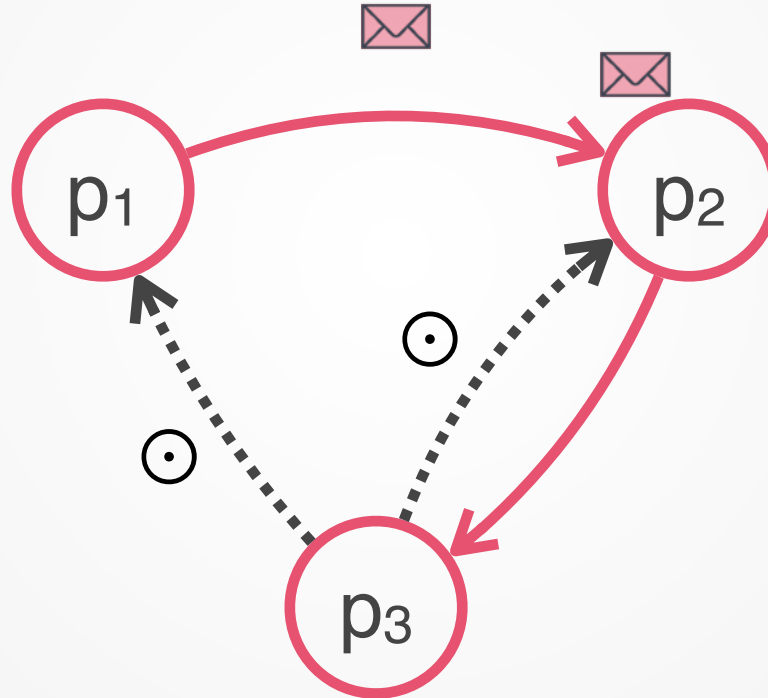
Snapshot

**s1, s2, s3**





# EXAMPLE EXECUTION



Snapshot

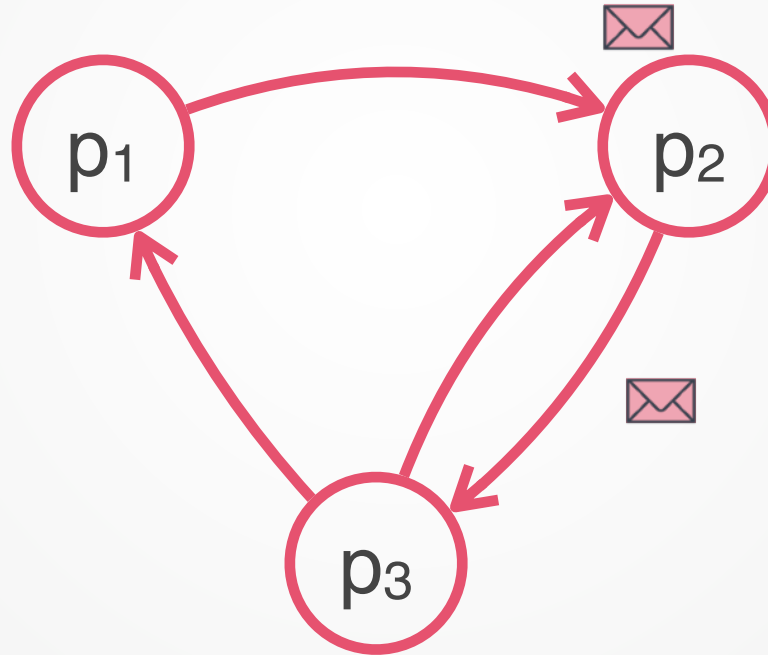
**s1, s2, s3**



before marker

after marker

# EXAMPLE EXECUTION



Snapshot

**s1, s2, s3**



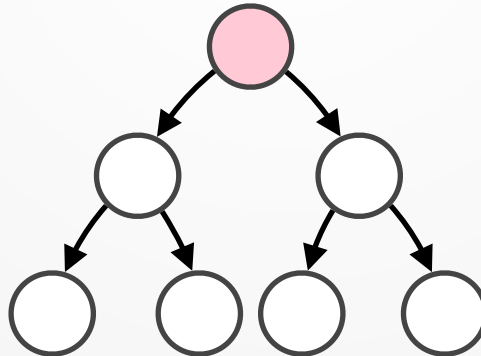
before marker

after marker

# PROOF SKETCH

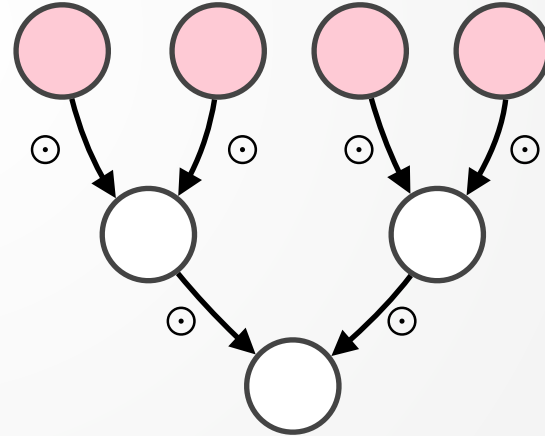
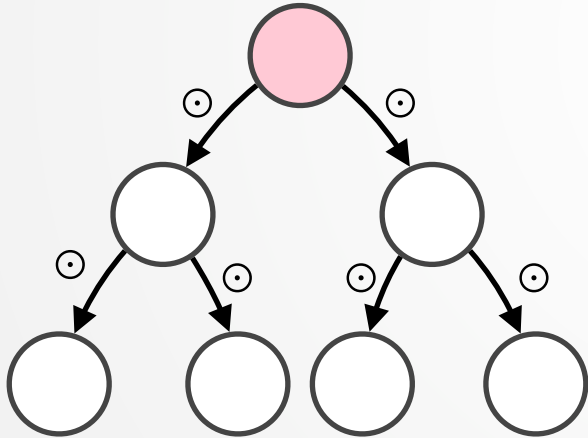
---

- **Validity**
  - **Marker** sent between  $p_i$  and  $p_j$  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)



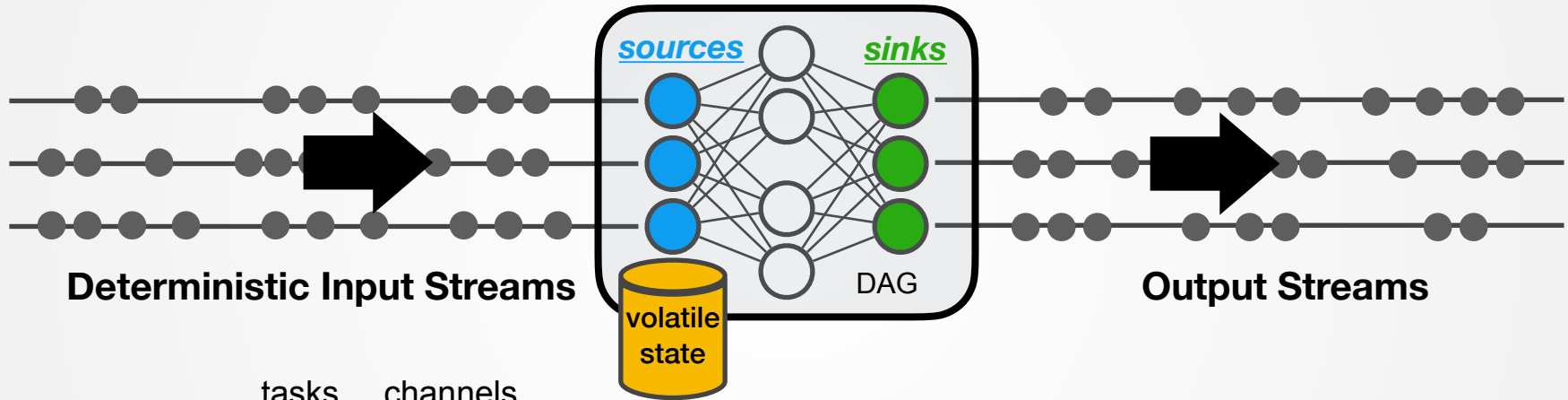
# Epoch Snapshotting

# 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

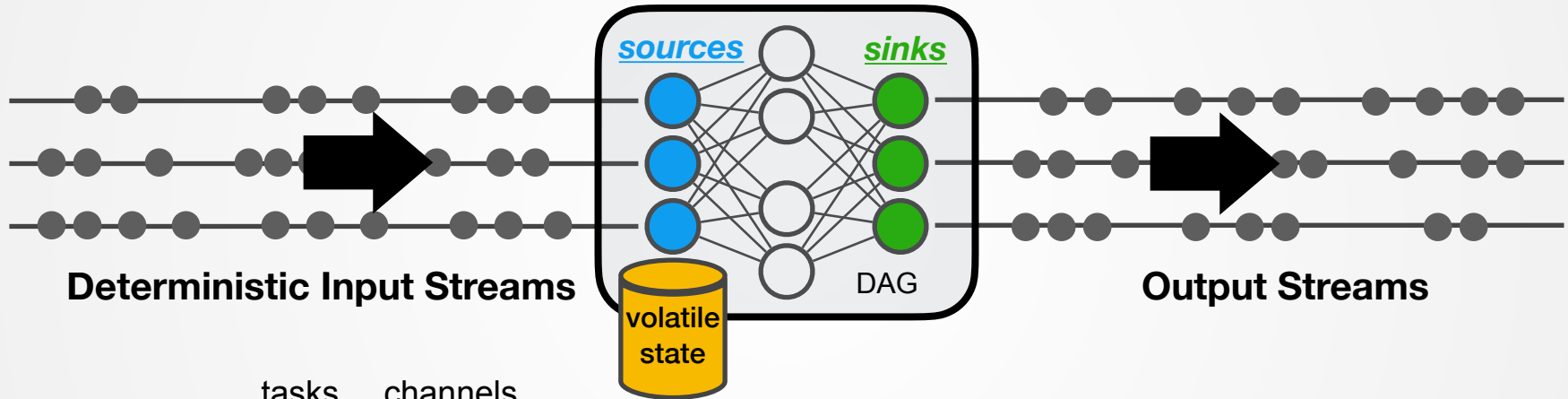
# STREAM PROCESSING



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

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

# STREAM PROCESSING



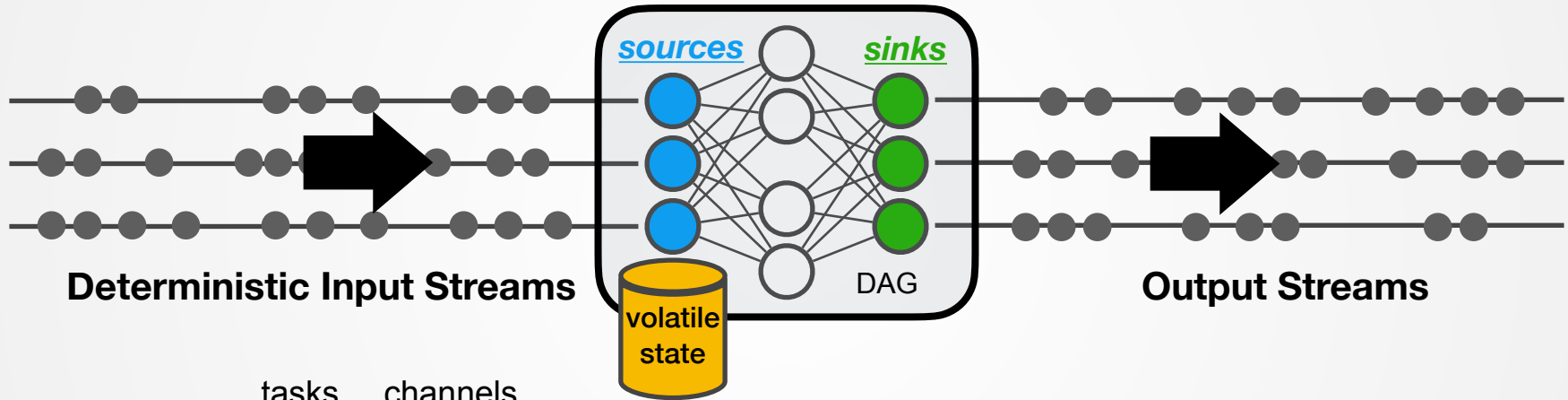
tasks channels  
System :  $\{\Pi, \mathbb{E}\}$

Task Actions

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



# STREAM PROCESSING



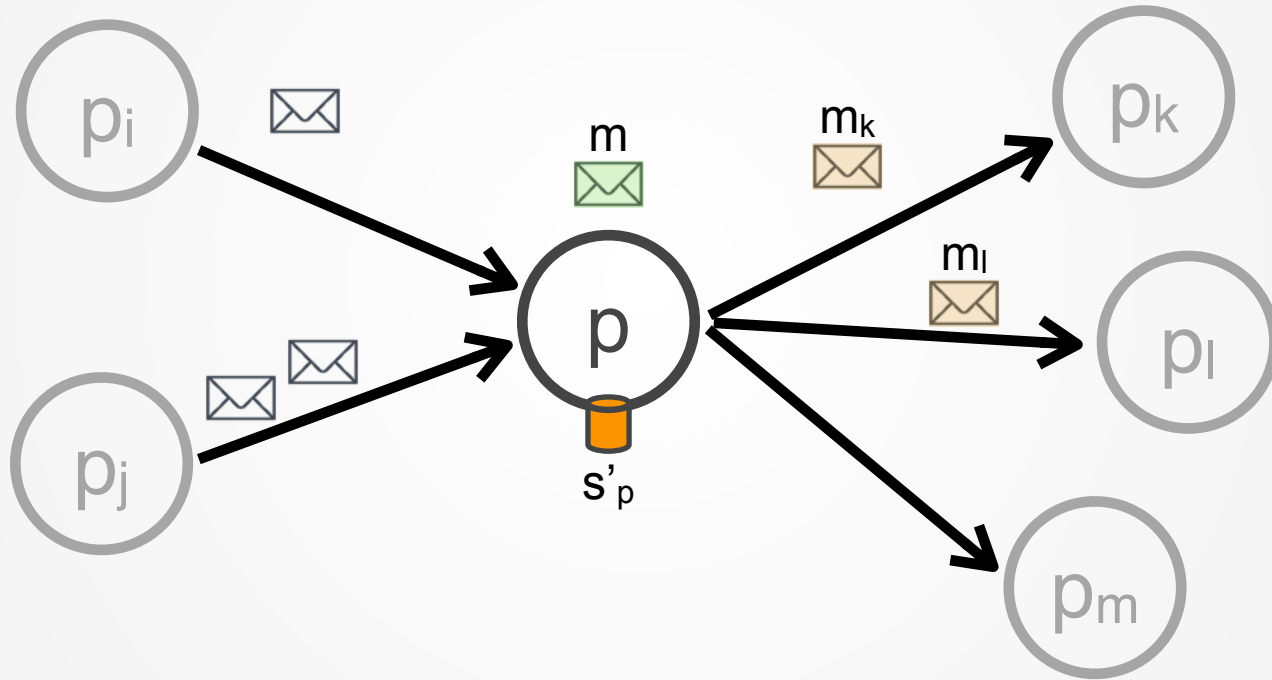
tasks channels  
 System :  $\{\Pi, \mathbb{E}\}$

**System Configurations** (states, messages in-transit)

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

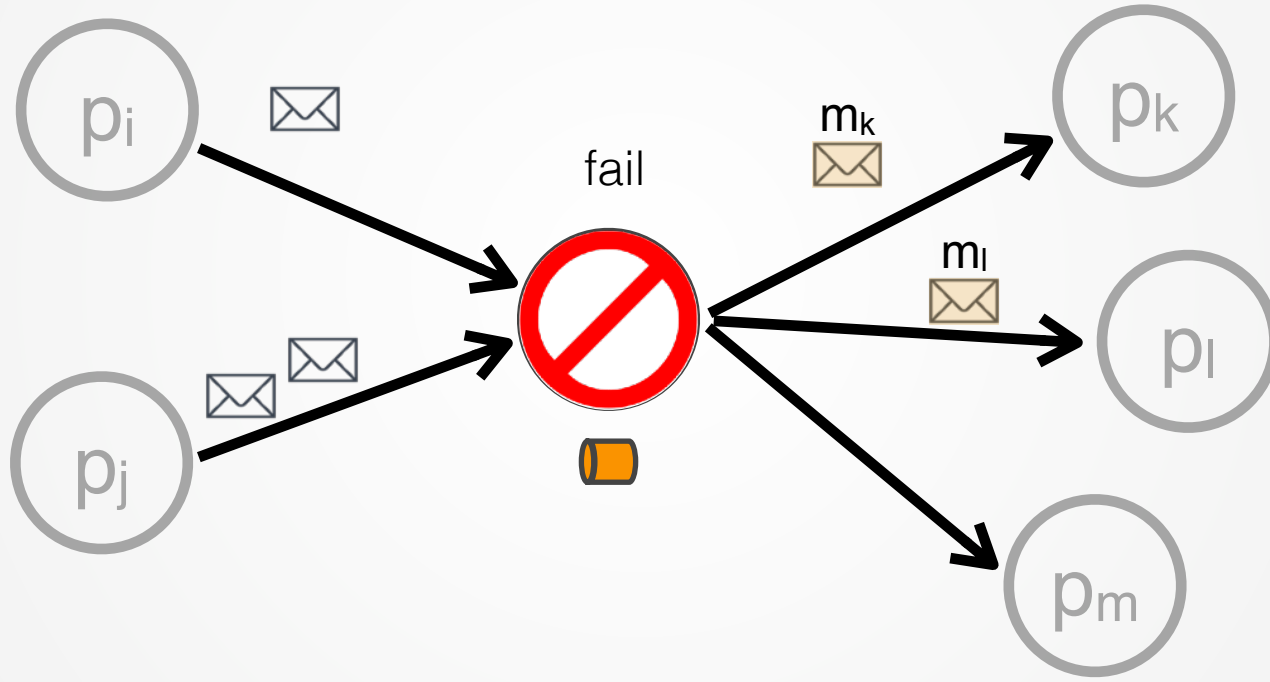
# FAULT TOLERANCE

---

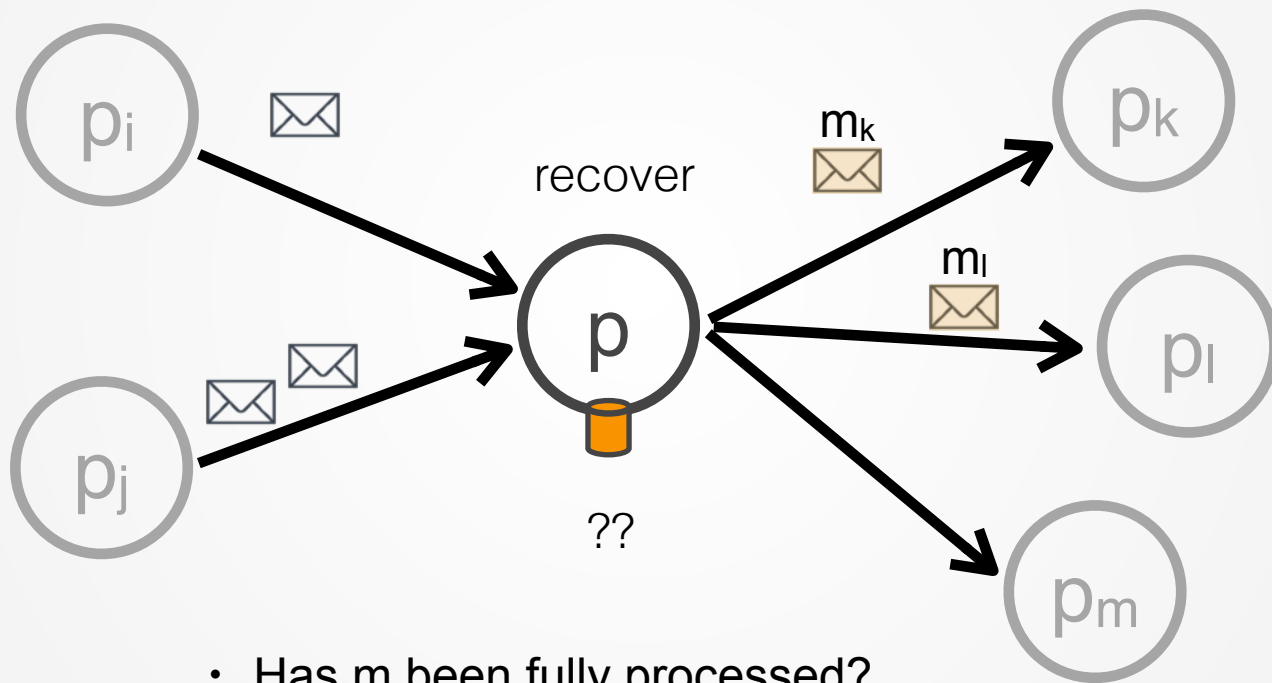


# FAULT TOLERANCE

---



# 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

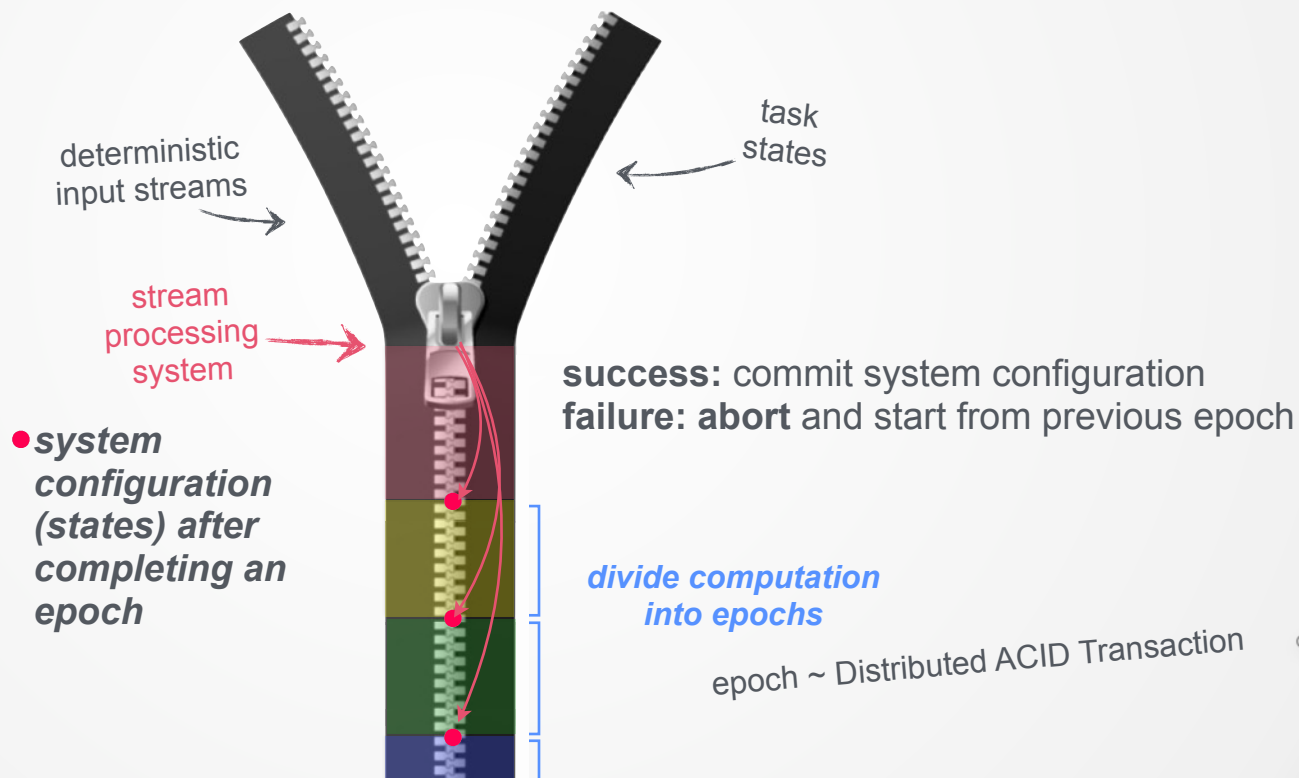
Fault-tolerance in the Borealis distributed stream processing system, in SIGMOD 2005

# FAULT TOLERANCE IS NOT ENOUGH

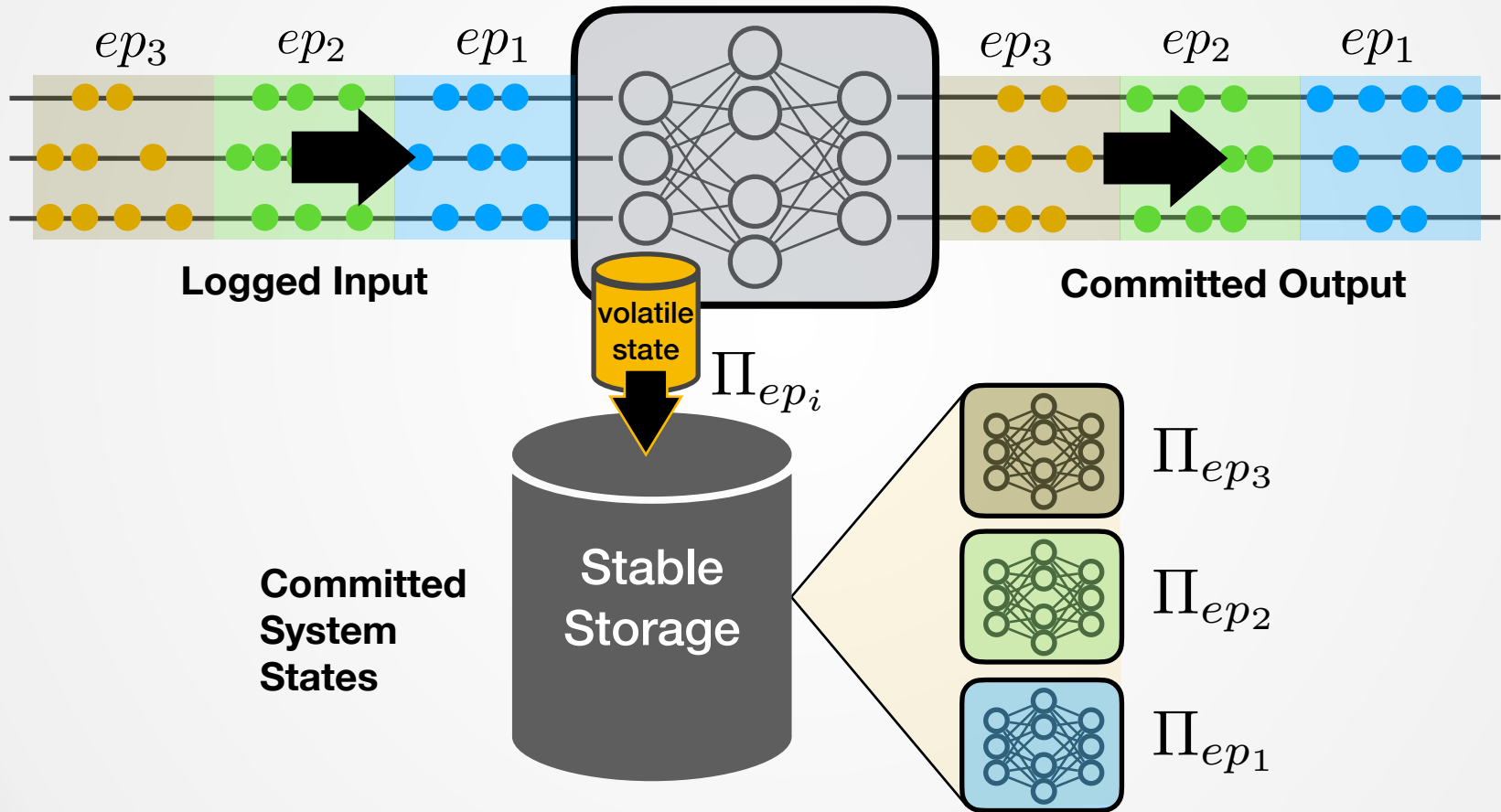
- Are output and states always correct?
- Can we reconfigure the system without losing computation?
- Can applications migrate without loss?
- Is external state access isolation possible?

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

# CONTINUOUS 2PC FOR DATA STREAMING

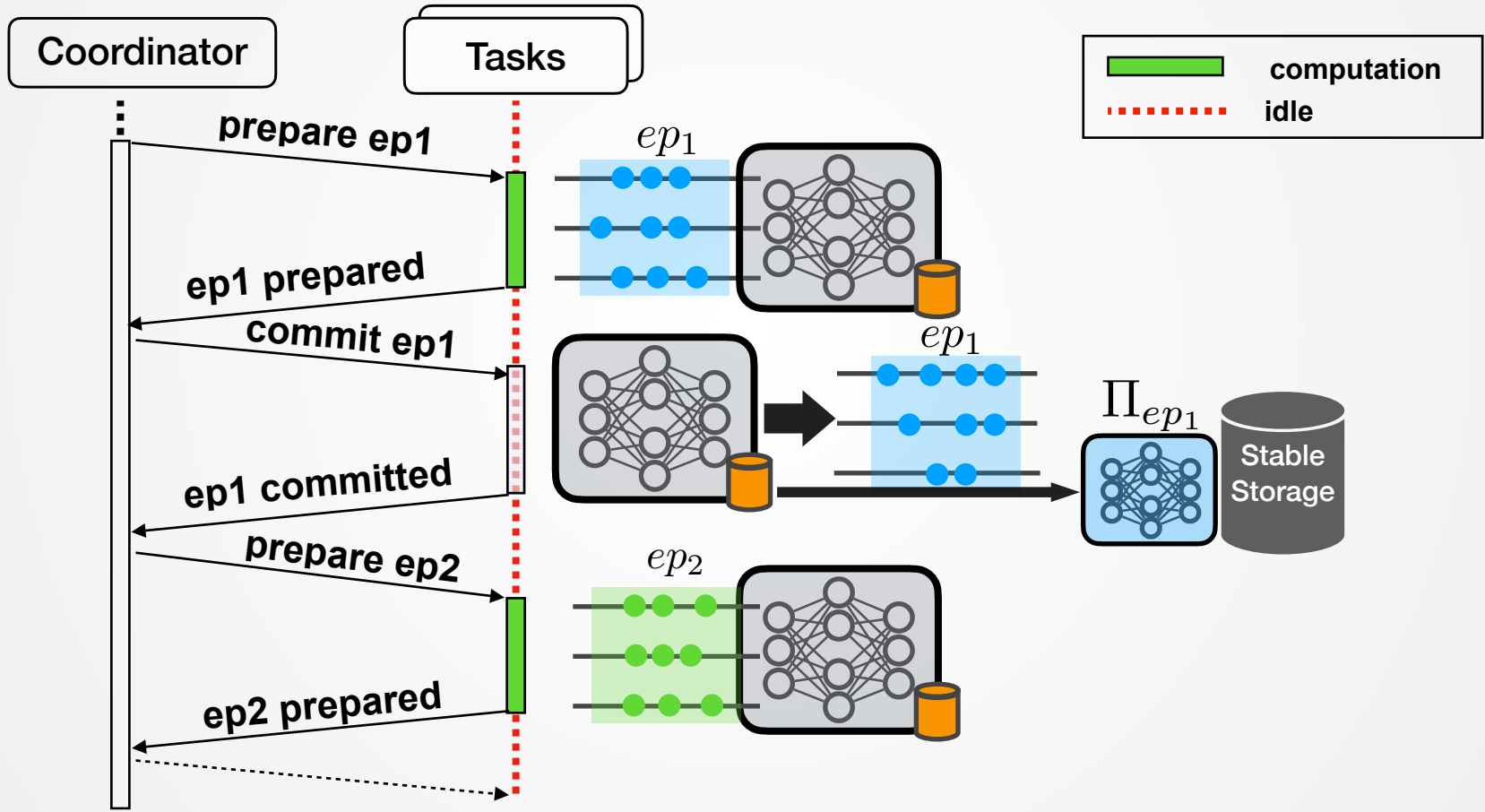


# TRANSACTIONAL STREAM EXECUTION





# SYNCHRONOUS 2PC

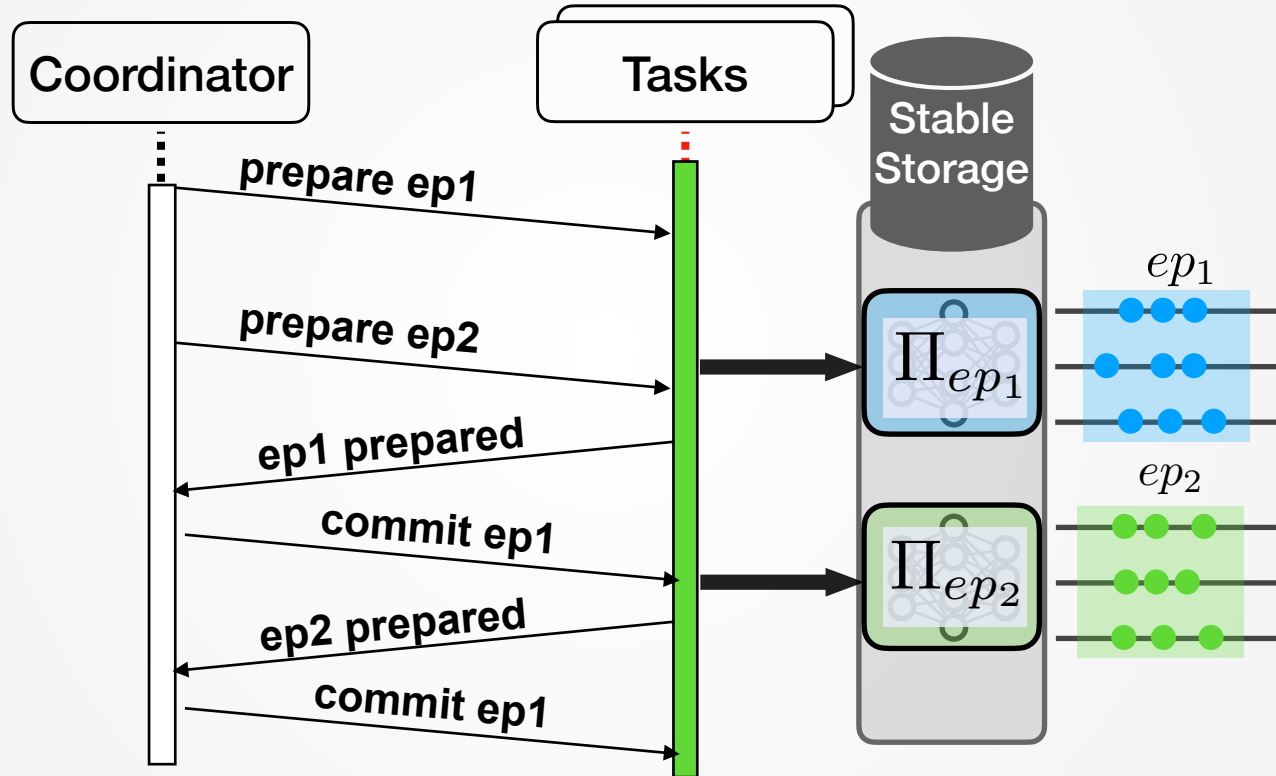


# SYNCHRONOUS 2PC

---

- 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



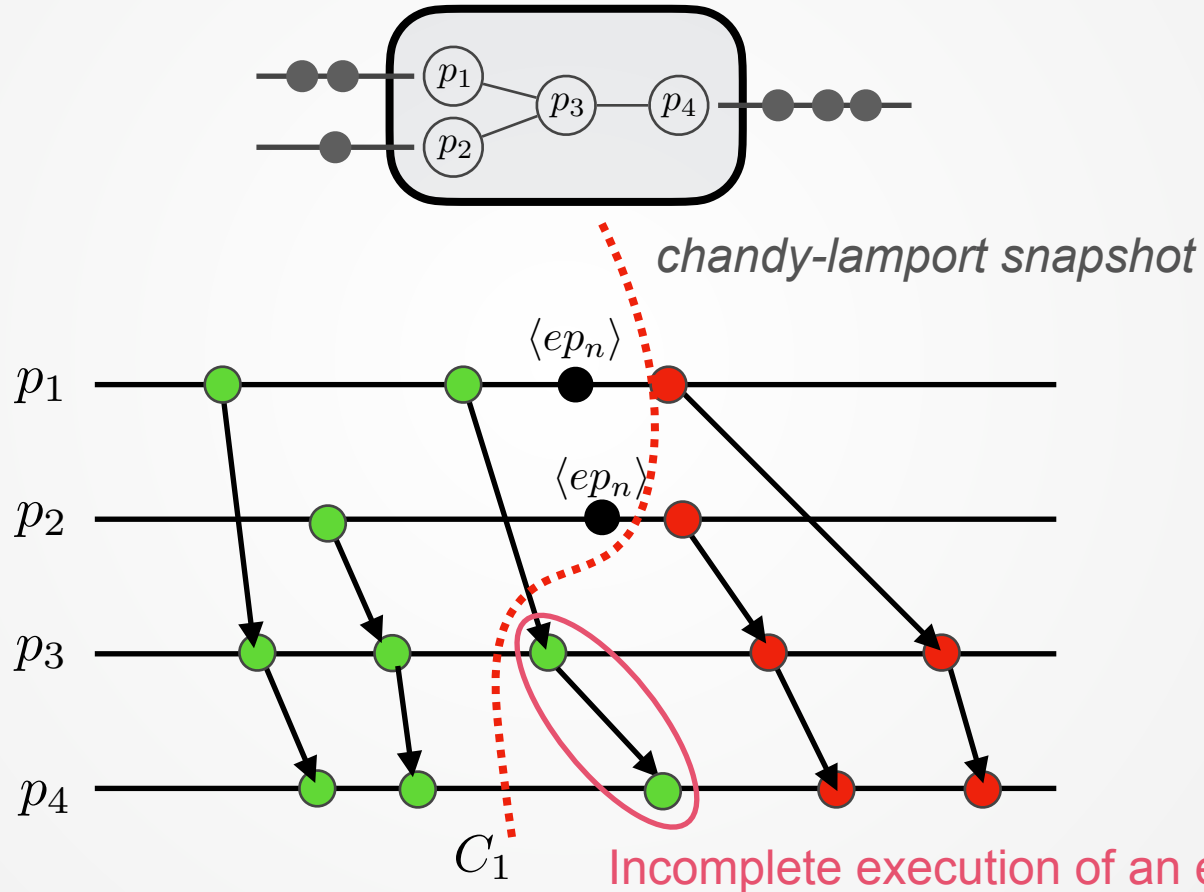
How? Using Snapshots

# EPOCH SNAPSHOTTING

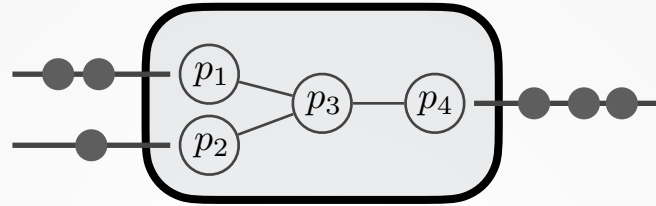
---

- Assumptions:
  - DAG of tasks
  - **Epoch change** events triggered on each **source** task ( $\langle \text{ep1} \rangle, \langle \text{ep2} \rangle, \dots$ )
    - Issued by master or generated periodically
- We want to snapshot stream process graphs after the **complete computation** of an epoch.

# VALIDITY IS NOT ENOUGH



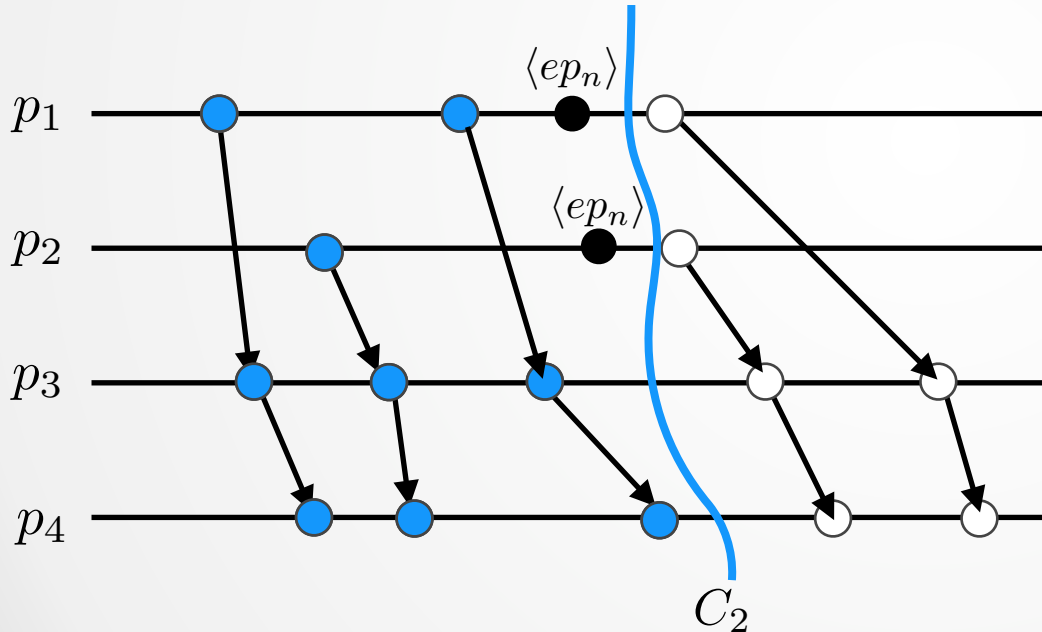
# TRANSACTIONAL EPOCH CUTS



## 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

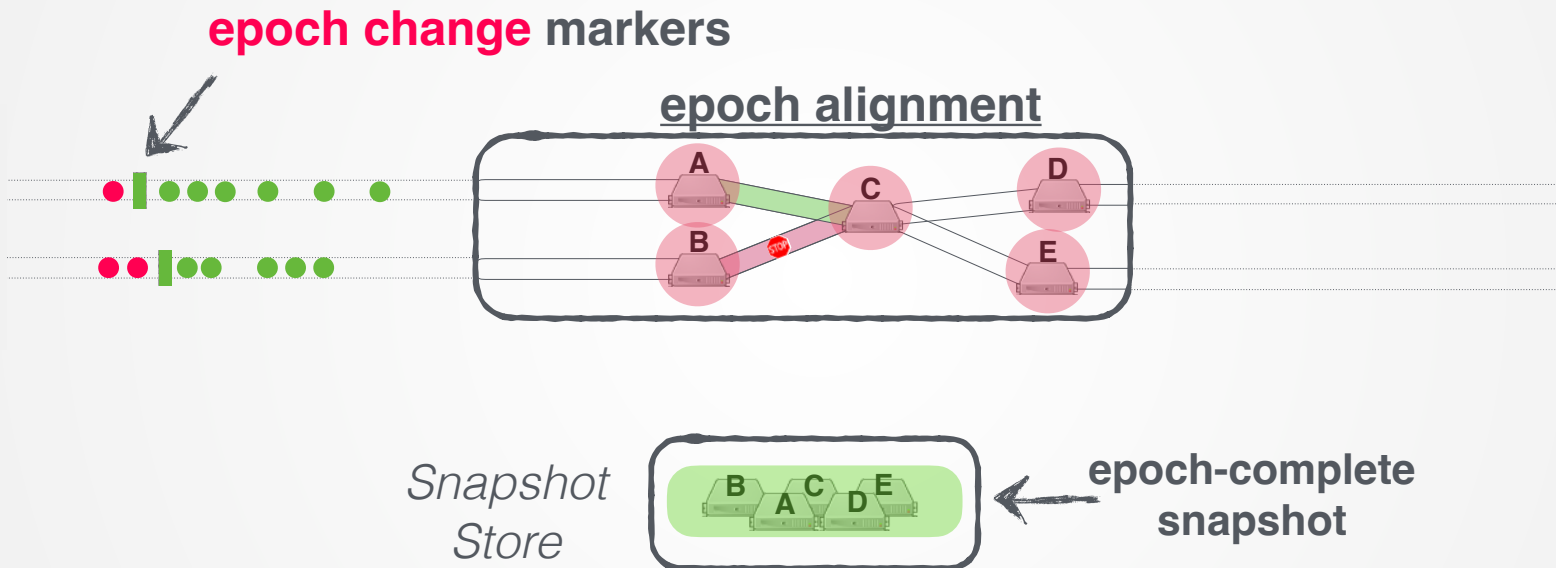
## **Validity (safety):**

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:**

```

1:  $\mathbb{O}_p \leftarrow \text{configured\_channels};$ 
2:  $s_p \leftarrow \emptyset;$ 

3: /* Source Task Logic
4: Upon  $\langle \text{rcvd}, m \rangle$ 
5:    $s_p \leftarrow \text{process}(s_p, m, \mathbb{O}_p);$ 
6: Upon  $\langle \text{ep}|n \rangle$ 
7:    $\text{esnap} \rightarrow \langle \text{record|self}, n, s_p \rangle;$ 
8:   foreach  $\text{out} \in \mathbb{O}_p$  do
9:      $\text{out} \rightarrow \langle \text{send}, \odot_n \rangle;$ 
```

## Epoch-Based Snapshots (Regular Tasks)

**Implements:** Epoch-Based Snapshotting (esnap)

**Requires:** FIFO Reliable Channel ( $\mathbb{I}_p, \mathbb{O}_p$ )


**Algorithm:**

```

1:  $(\mathbb{I}_p, \mathbb{O}_p) \leftarrow \text{configured\_channels};$ 
2:  $\text{Enabled} \leftarrow \mathbb{I}_p;$ 
3:  $s_p \leftarrow \emptyset;$ 

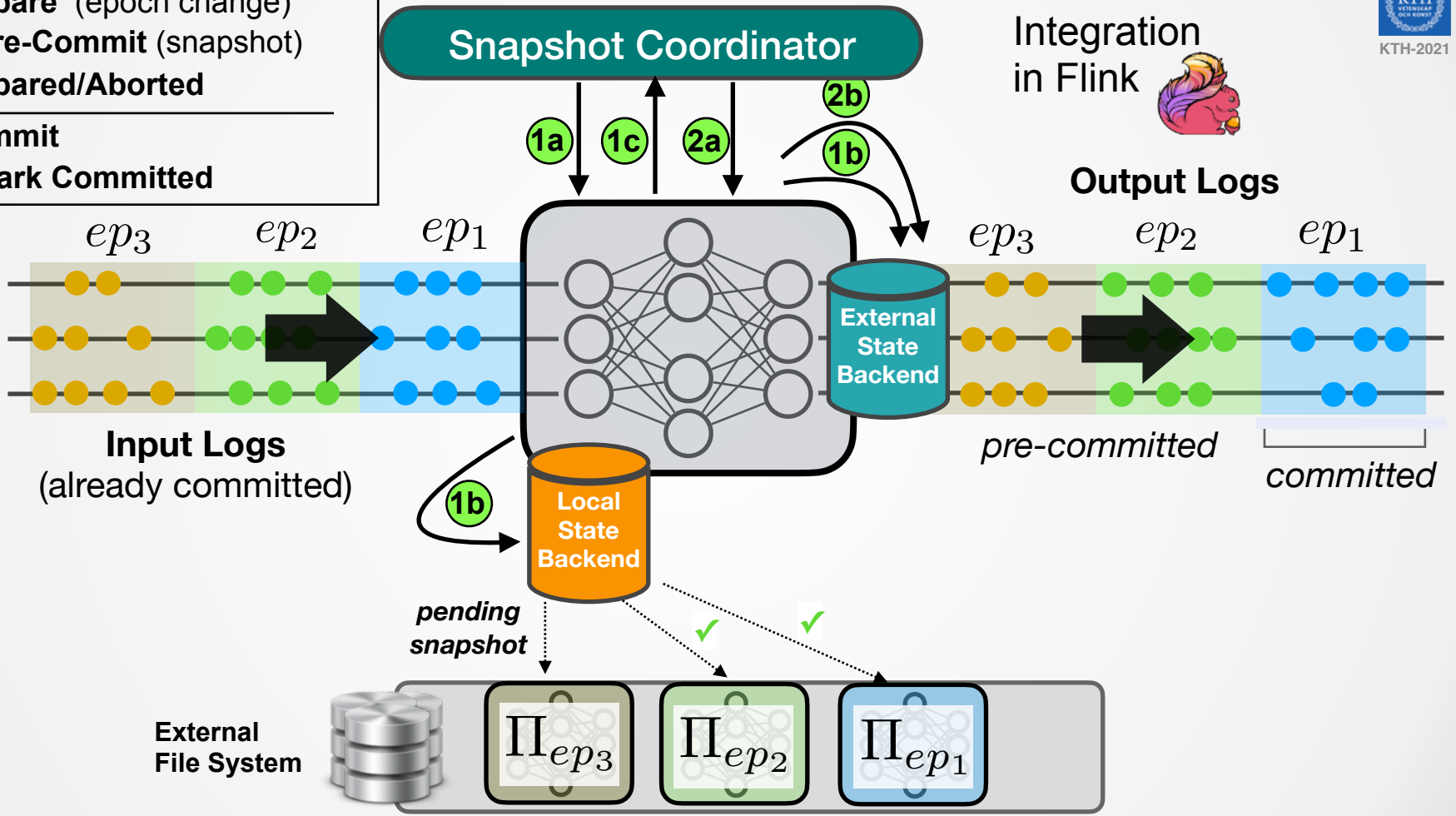
4: /* Common Task Logic
5: Upon  $\langle \text{rcvd}, m \rangle$  on  $c \in \text{Enabled}$ 
6:    $s_p \leftarrow \text{process}(s_p, m, \mathbb{O}_p);$ 
7: Upon  $\langle \text{rcvd}, \odot_n \rangle$  on  $c \in \text{Enabled}$ 
8:    $\text{esnap} \rightarrow \langle \text{record|self}, n, s_p \rangle;$ 
9:    $\text{Enabled} \leftarrow \text{Enabled}/\{c\};$ 
10:  if  $\text{Enabled} = \emptyset$  then
11:    foreach  $\text{out} \in \mathbb{O}_p$  do
12:       $\text{out} \rightarrow \langle \text{send}, \odot_n \rangle;$ 
13:     $\text{Enabled} \leftarrow \mathbb{I}_p;$ 
```

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