Distributed Snapshots

September 2025

A Note on Channels and Goroutines

- Using channels is easy, debugging them is hard…

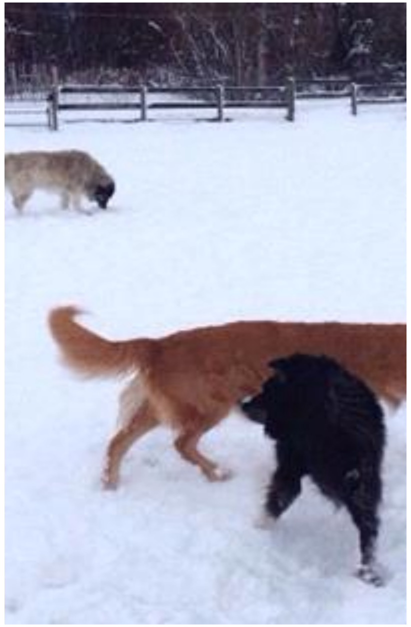
    Bullet-proof way: Keep track of how many things go in and go out

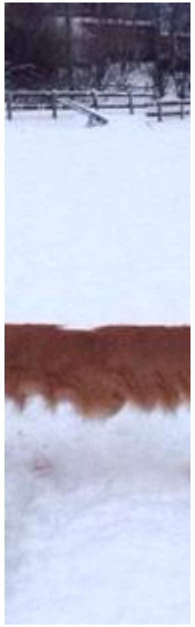
    Always ask yourself: is this channel buffered?

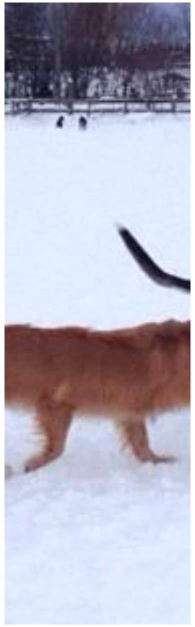
- In general, don’t use locks or atomic operations with channels (awkward)
- Try not to nest goroutines (would be hard to reason about them)

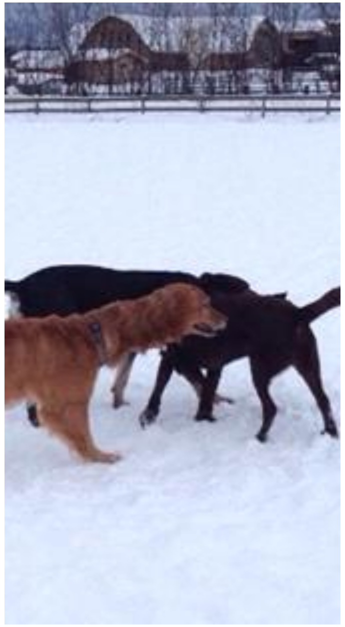
Review of Snapshot Fundamentals

Taking a panorama with a moving object in the scene.


N1


N2


N3

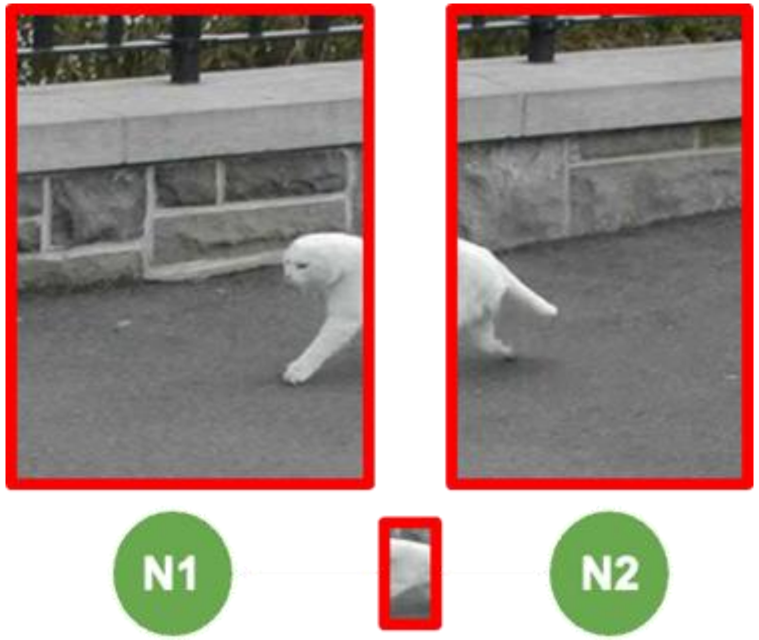

N4

Distributed snapshots are hard

Must ensure **state is not duplicated** across the cluster (we only want one dog)

Must ensure **state is not lost** across the cluster (we still want the dog there)

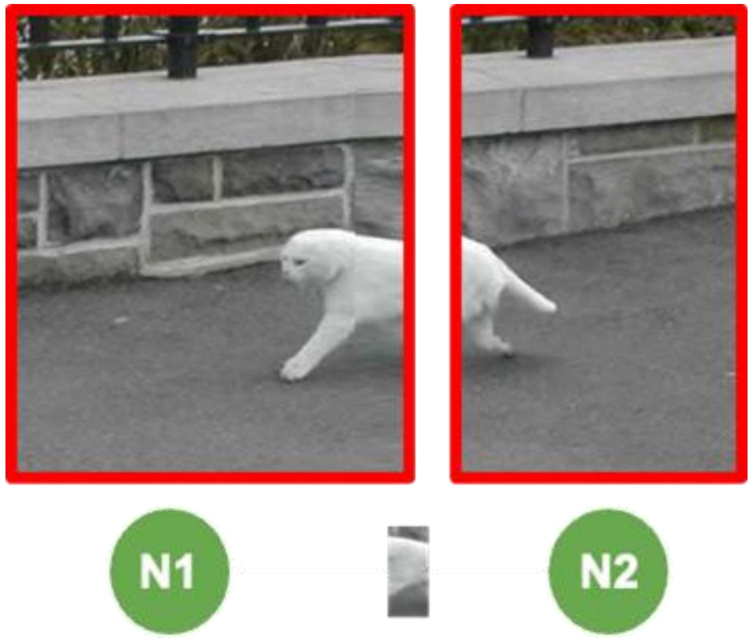
We must carefully consider how and when exactly to record state.



Event order:

1. Snap N1
2. N2 sends body
3. Snap N2
4. N1 receives body

Should record message!



Event order:

1. N2 sends body
2. Snap N2
3. N1 receives body
4. Snap N1

N1 already received the body in step 3

Should NOT record message

Intuition: guarantee zero loss + zero duplication

If you *haven't* snapshotted your local state yet:

Do NOT record future messages you receive

If you *have* snapshotted your local state:

Do record future messages you receive

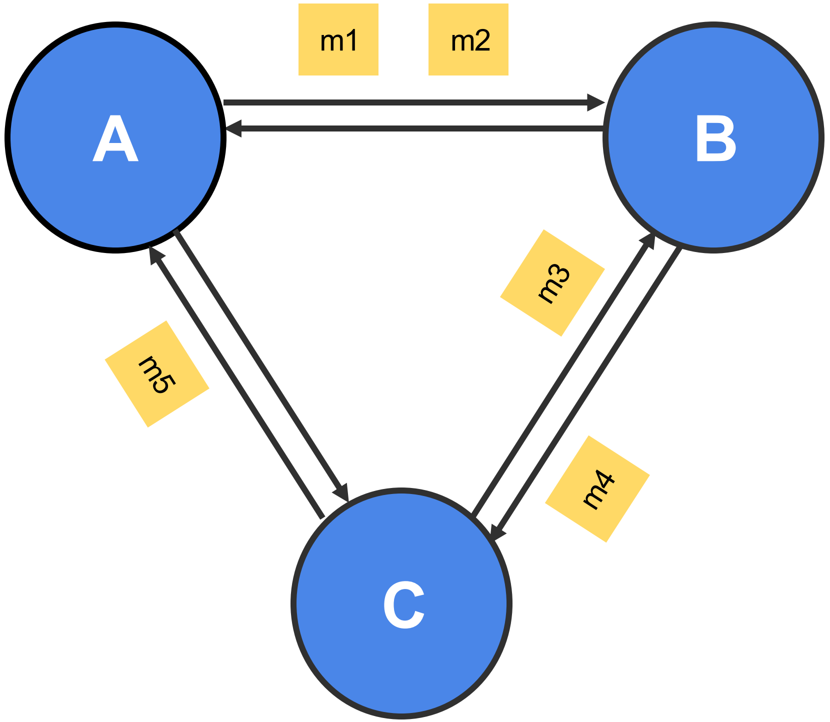
Which one guarantees zero loss?

Which one guarantees zero duplication?

What is a Global Snapshot?

- A global snapshot captures the global state of a distributed system:
  - Local state of each **process** within the distributed system
  - Local state of each **communication channel**

- These local states are **instantaneous**
  - e.g messages in transit from one node to another only exist in transit for a few milliseconds or less



Global Snapshots are Useful

- Checkpointing
  - Recover more quickly after failures
- Garbage Collection
  - Remove objects that are not referenced any more by other objects/processes at any other servers
- Deadlock Detection
  - Examine the global application state and identify any deadlocks, useful in transactional database systems
- And many others …

System Model

- N **processes** in the system
  - Each process keeps track of some state
- There are two **unidirectional communication channels** between each pair of processes P and Q
  - FIFO-ordered (First-In-First-Out)
  - Message arrives intact and is unduplicated
  - Each channel also has some state
- No failures

Messages and States

- What are the messages?
  - Application messages that differ across systems (e.g "sending $10 from A to B", "read value at memory address X and write back with a new value")
  - Special messages (e.g marker message) that should not interfere with application messages
- What are the states?
  - Process state: application-defined state, or the classic notion of state which includes heap, registers, program counters and etc
  - Channel state: the set of messages inside
- Tips for Assignment 2
  - See `*.top`, `*.events`, `*.snap` files under ./test_data to understand what states and messages mean in this assignment
  - Read `test_common.go` to understand the syntax of the above files, and their relationships with the simulator

Distributed Snapshot

"Distributed Snapshots: Determining Global States of Distributed Systems" 1985, by K. Mani Chandy and Leslie Lamport

Key Idea: Servers send ***marker messages*** to each other

Marker messages

1. Mark the beginning of the snapshot process on the server
2. Act as a barrier (stopper) for recording messages

Chandy-Lamport Algorithm

Any process can initiate the snapshot

- Record local state
- Create marker messages and send them to all outbound channels
- Start recording messages from all incoming channels

Chandy-Lamport Algorithm Continued

When receiving a marker message from channel C

If this is the **first marker message** that this process has even seen:

- Record the local state
- Record the state of C as “empty sequence”
- Send out the marker message on all outbound channels
- Start recording messages from all of its other incoming channels

If it has **already seen** a marker message (e.g. from some other channel)

- Record the state of C as the sequence of messages received since the process’s local state has been recorded
- Stop recording messages on C (i.e done with recording the channel’s state)

Chandy-Lamport Algorithm Continued

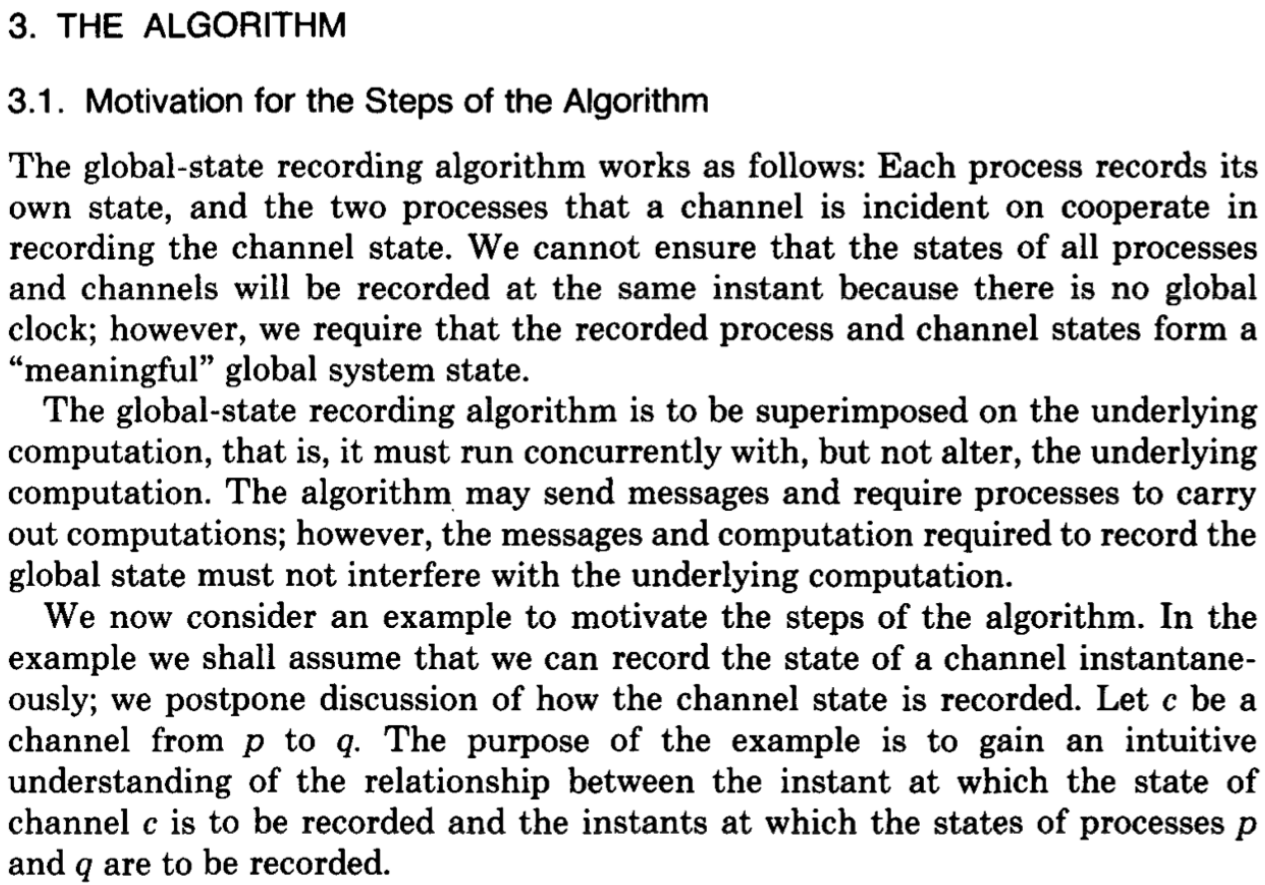
When is the algorithm terminated?

- All processes have received marker messages (i.e have recorded their local states)
- All processes have received marker messages from all of their incoming channels (i.e have recorded the local states of all channels)
- Both need to satisfy

What happens after the termination?

- Optional and out of the scope of Chandy-Lamport algorithm
- Usually, there will be a central server that collects local snapshots from all servers to build a global snapshot (e.g the `simulator` in Assignment 2) and maybe run some computations (e.g deadlock detection) on it

See **Section 3** of the original paper for more details



3. THE ALGORITHM

3.1. Motivation for the Steps of the Algorithm

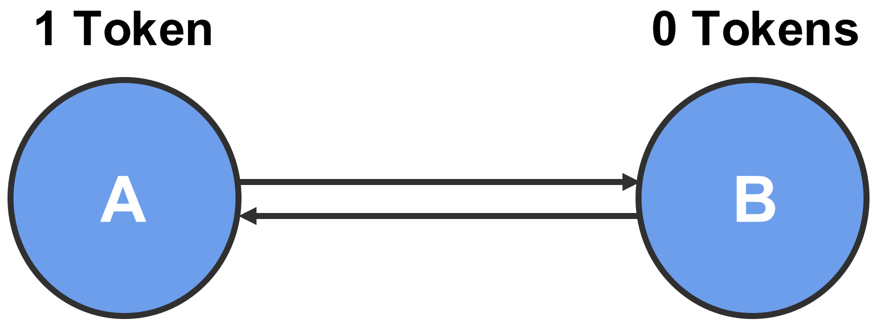
The global-state recording algorithm works as follows: Each process records its own state, and the two processes that a channel is incident on cooperate in recording the channel state. We cannot ensure that the states of all processes and channels will be recorded at the same instant because there is no global clock; however, we require that the recorded process and channel states form a "meaningful" global system state.

The global-state recording algorithm is to be superimposed on the underlying computation, that is, it must run concurrently with, but not alter, the underlying computation. The algorithm may send messages and require processes to carry out computations; however, the messages and computation required to record the global state must not interfere with the underlying computation.

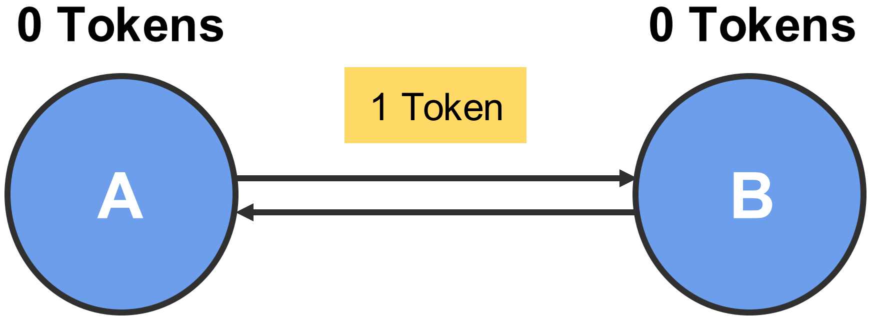
We now consider an example to motivate the steps of the algorithm. In the example we shall assume that we can record the state of a channel instantaneously; we postpone discussion of how the channel state is recorded. Let c be a channel from p to q. The purpose of the example is to gain an intuitive understanding of the relationship between the instant at which the state of channel c is to be recorded and the instants at which the states of processes p and q are to be recorded.

Exercises

Token Passing Example 1



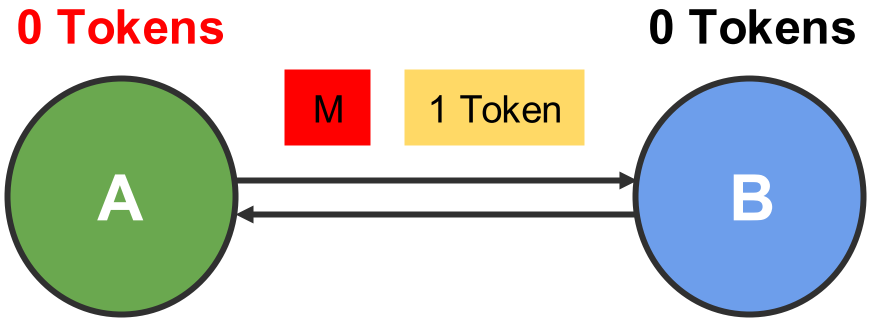
Token Passing Example 1



Event order:

1. *A* sends 1 token

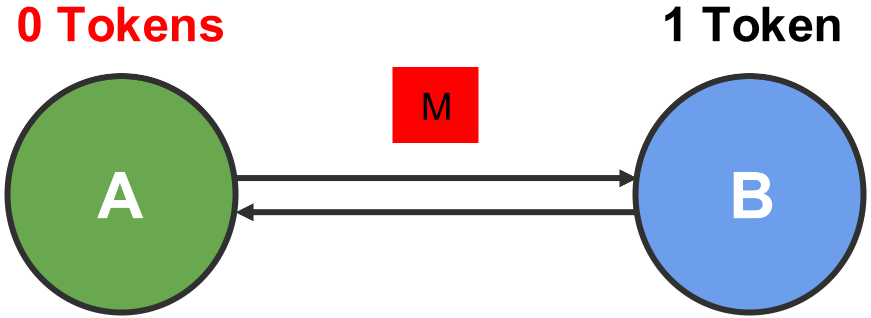
Token Passing Example 1



Event order:

1. *A* sends 1 token
2. *A* starts snapshot, sends marker

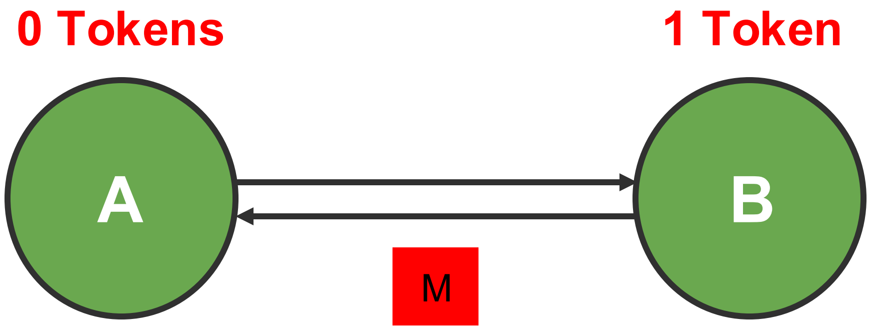
Token Passing Example 1



Event order:

1. *A* sends 1 token
2. *A* starts snapshot, sends marker
3. *B* receives 1 token

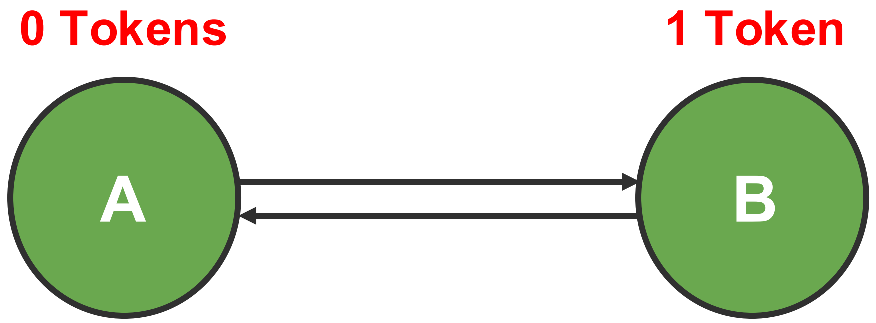
Token Passing Example 1



Event order:

1. *A* sends 1 token
2. *A* starts snapshot, sends marker
3. *B* receives 1 token
4. *B* receives marker, starts snapshot

Token Passing Example 1

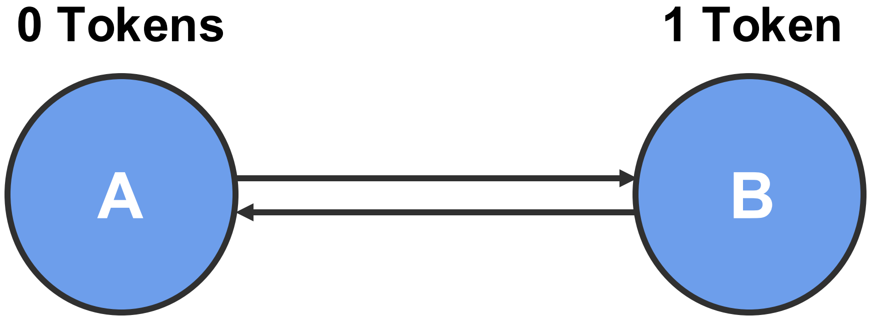


We did not record the token message because B received it before B started the snapshot process

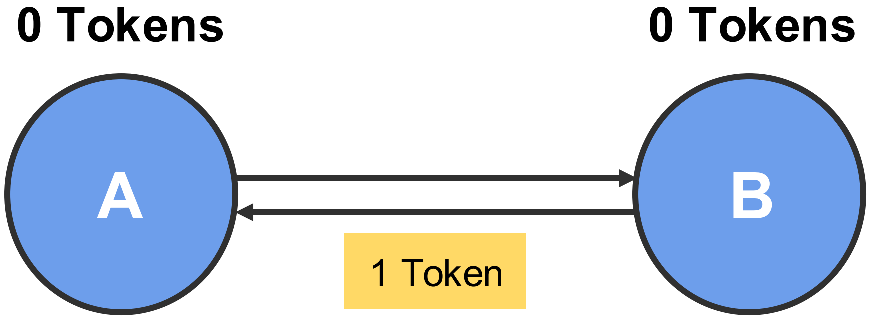
Event order:

1. *A* sends 1 token
2. *A* starts snapshot, sends marker
3. *B* receives 1 token
4. *B* receives marker, starts snapshot
5. *A* receives marker, ends snapshot

Token Passing Example 2



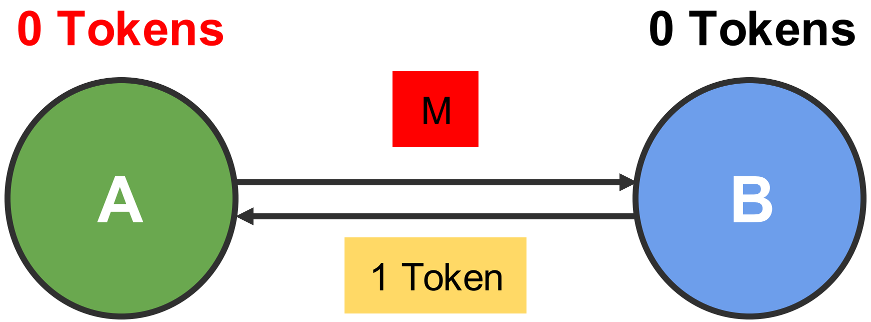
Token Passing Example 2



Event order:

1. *B* sends 1 token

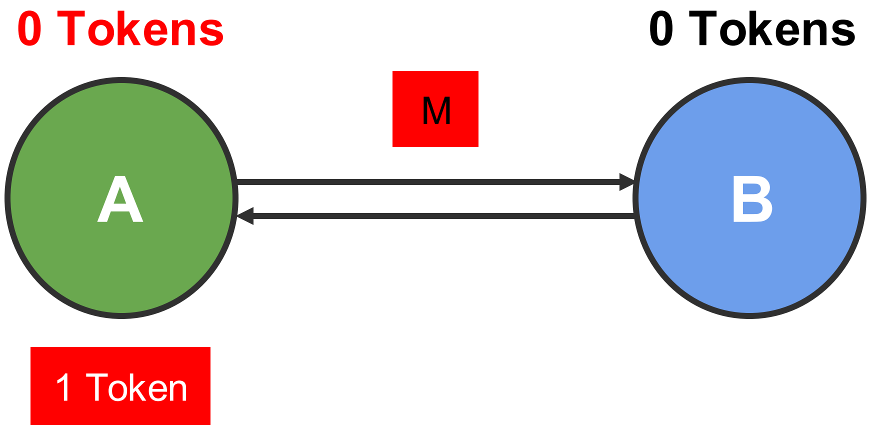
Token Passing Example 2



Event order:

1. *B* sends 1 token
2. *A* starts snapshot, sends marker

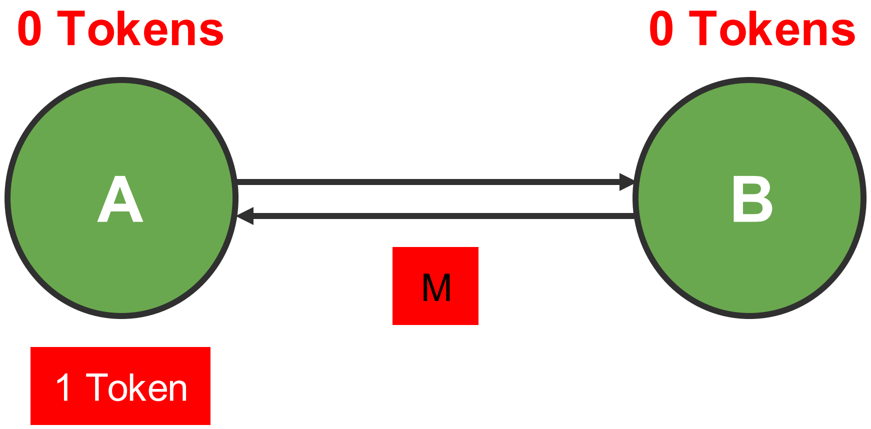
Token Passing Example 2



Event order:

1. *B* sends 1 token
2. *A* starts snapshot, sends marker
3. *A* receives 1 token, records message

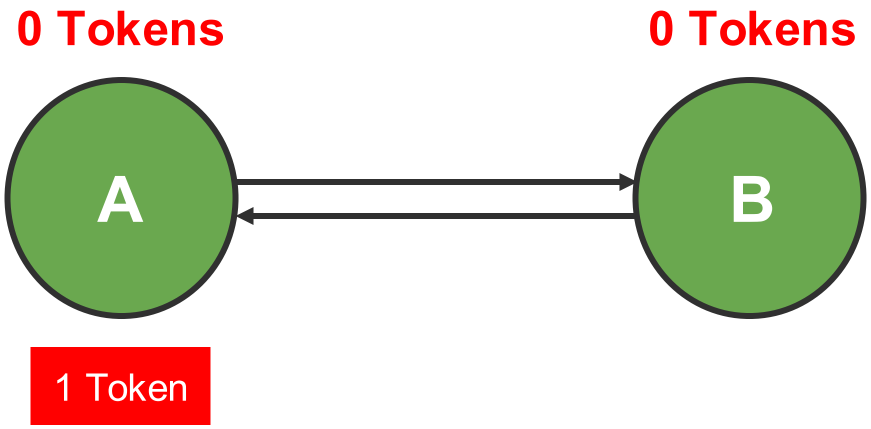
Token Passing Example 2



Event order:

1. *B* sends 1 token
2. *A* starts snapshot, sends marker
3. *A* receives 1 token, records message
4. *B* receives marker, starts snapshot

Token Passing Example 2

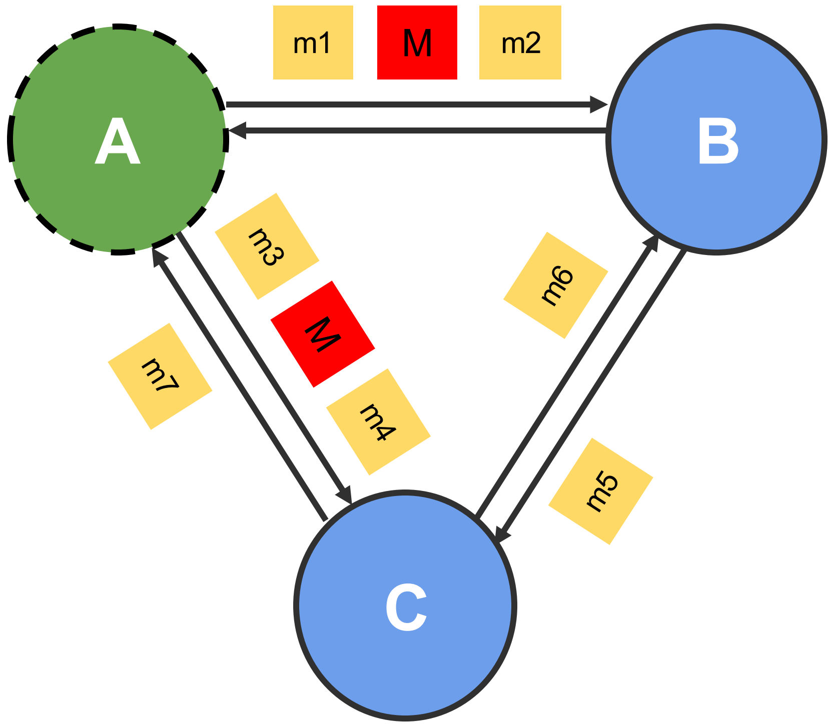


We recorded the token message because A received it **after** *it has already started the snapshot process*

Event order:

1. *B* sends 1 token
2. *A* starts snapshot, sends marker
3. *A* receives 1 token, records message
4. *B* receives marker, starts snapshot
5. *A* receives marker, ends snapshot

Token Passing Example 3



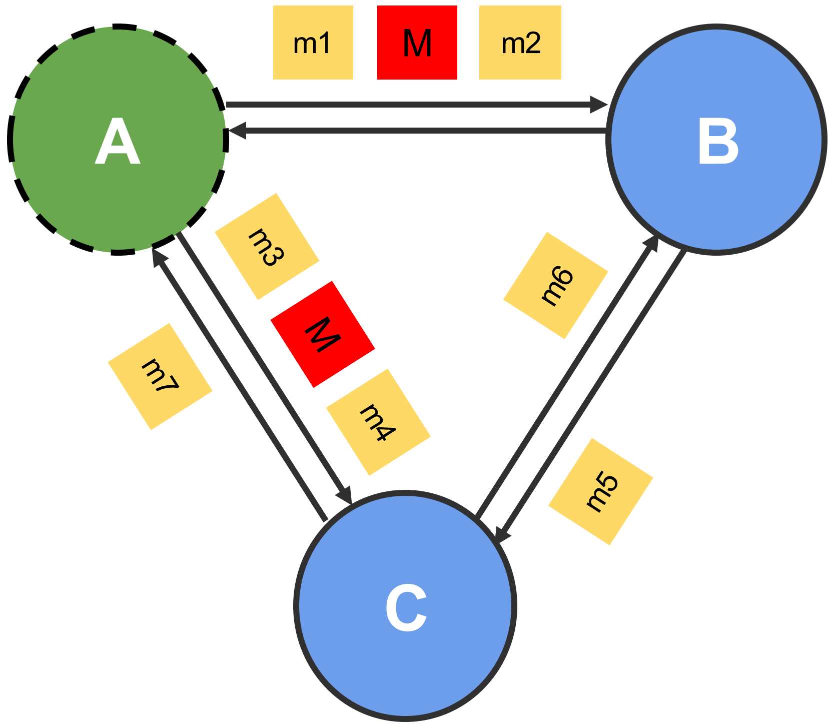
Which messages are definitely recorded*?

Which messages are definitely *not* recorded?

Which messages *might* be recorded?

* recorded as in-flight messages, i.e., as part of *channel state* rather than *process state*

Token Passing Example 3



Which messages are definitely recorded*?

m7

Which messages are definitely *not* recorded?

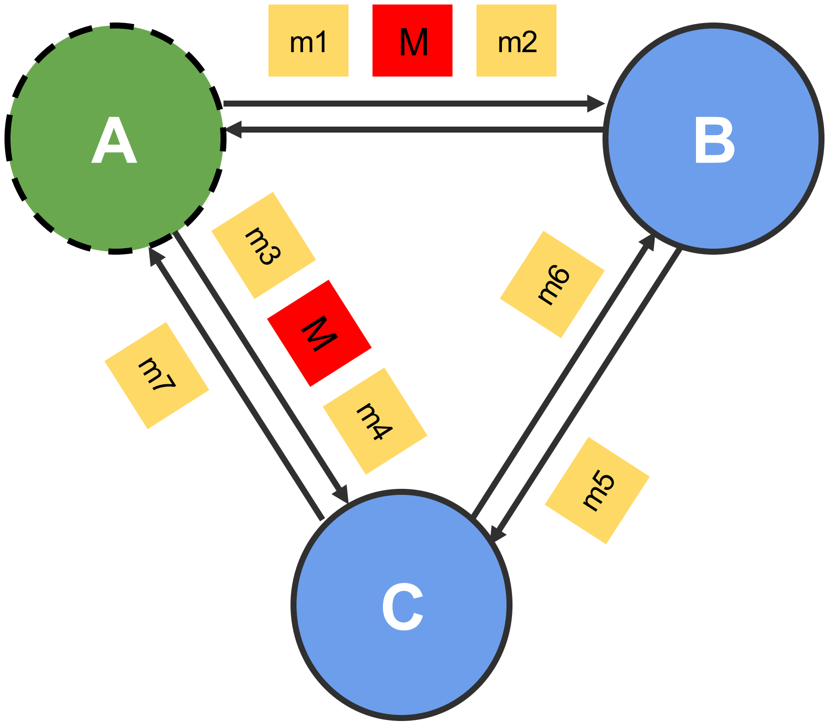
m1, m3

Which messages *might* be recorded?

m2, m4, m5, m6

*recorded as in-flight messages

Takeaways



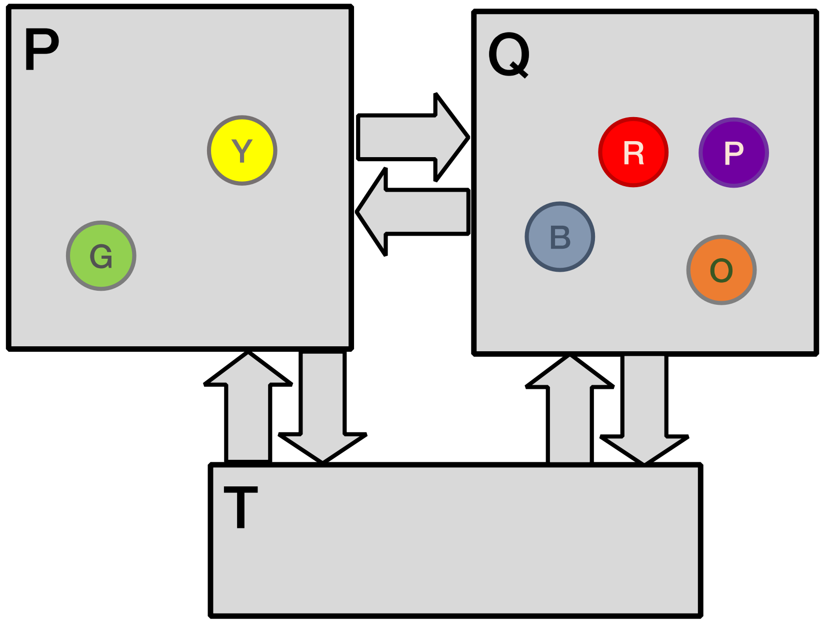
- ❑ Tokens arriving on the incoming channels of a process that has already started the snapshot are recorded as the state of that channel.
- ❑ Tokens sent on an outbound channel after a process has sent a marker on that channel are not included in the state of the channel.
- ❑ The channels through which each process receives the marker for the first time are recorded to be empty.

Puzzles from the Lecture

Chandy-Lamport Puzzle #5

Is this snapshot possible? And if so, how?

P	= { G, Y, }
chan(P, Q)	= { }
chan(P, T)	= { }
Q	= { B, O }
chan(Q, P)	= { P }
chan(Q, T)	= { R }
T	= { }
chan(T, P)	= { }
chan(T, Q)	= { }



P was captured in (Q,P)

1. Process P started the snapshot before Token P was received

2. It received the marker back after receiving Token P

R was captured in (Q,T)

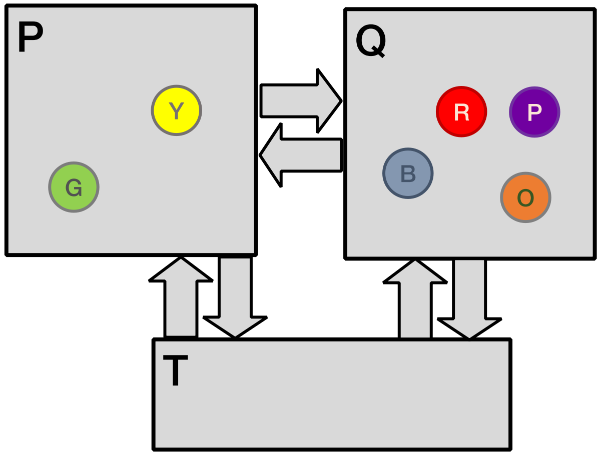
1. Process T started the snapshot before Token R was received

2. It received the marker back after receiving Token R

Chandy-Lamport Puzzle #5

Is this snapshot possible? And if so, how?

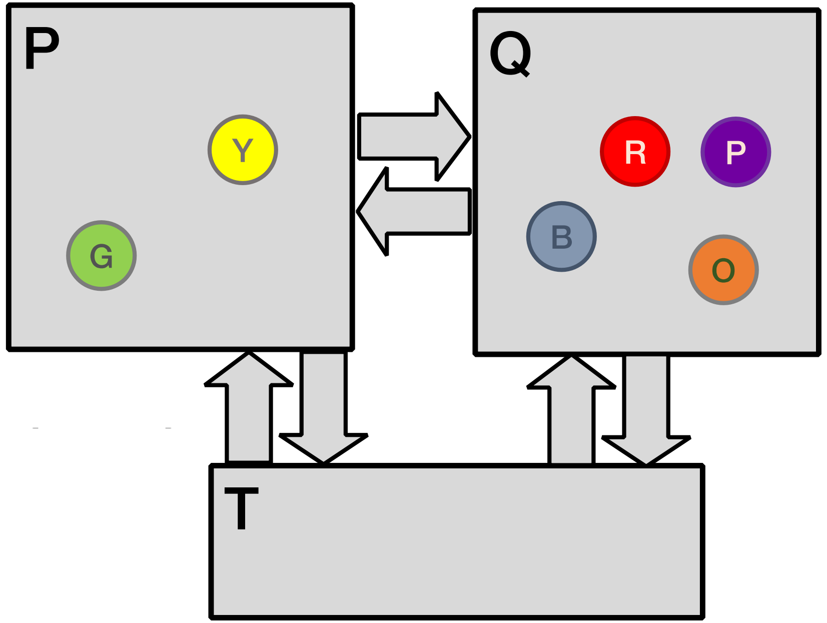
P	= { G, Y, }
chan(P, Q)	= { }
chan(P, T)	= { }
Q	= { B, O }
chan(Q, P)	= { P }
chan(Q, T)	= { R }
T	= { }
chan(T, P)	= { }
chan(T, Q)	= { }



Chandy-Lamport Puzzle #6

Is this snapshot possible? And if so, how?

P	= { G, Y, }
chan(P, Q)	= { }
chan(P, T)	= { }
Q	= { B }
chan(Q, P)	= { P }
chan(Q, T)	= { R }
T	= { O }
chan(T, P)	= { }
chan(T, Q)	= { }



Assignment 2 Overview

- You will implement the Chandy-Lamport snapshot algorithm
- Application is a token passing system
  - Number of tokens must be preserved in your snapshots
- Implementation uses *discrete time* simulator to order events
  - `Simulator` manages servers and injects events into the system
  - `Server` implements the snapshot algorithm
- Allow multiple active snapshot processes
  - E.g, The second snapshot can start before the first snapshot completes in the system

Assignment 2 Interfaces

```
func (sim *Simulator) Tick()

func (sim *Simulator) StartSnapshot(serverId string)

func (sim *Simulator) NotifySnapshotComplete(serverId string, snapshotId int)

func (sim *Simulator) CollectSnapshot(snapshotId int) *SnapshotState
```

- What kind of state does the simulator need to keep track of?
  - Time
  - Topology
  - Channels to signal the completion of snapshots
  - ...

Assignment 2 Interfaces

```
func (server *Server) SendToNeighbors(message interface{})

func (server *Server) SendTokens(numTokens int, dest string)

func (server *Server) HandlePacket(src string, message interface{})

func (server *Server) StartSnapshot(snapshotId int)
```

- What kind of state does the server need to keep track of?
  - Local state
  - Neighbors
  - Which channels received markers
  - Recorded messages
  - ...

Assignment 2

Start Early ☺

Due October 3 (Friday) at 11:59pm!