Distributed Snapshots

10/4/19

A note on channels and goroutines...

Using channels is easy, debugging them is hard…

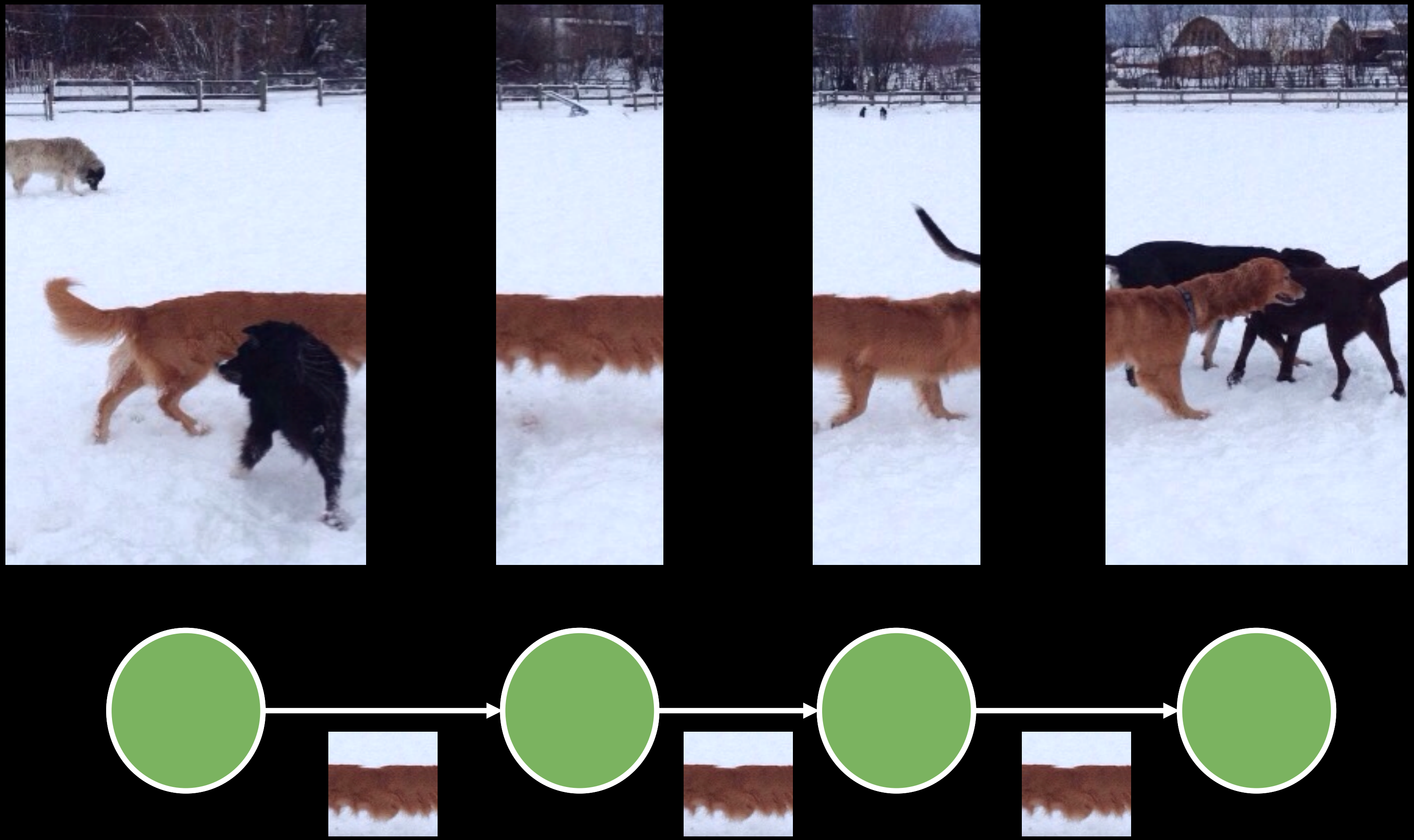
Bullet-proof way: Keep track of how many things go in and go out

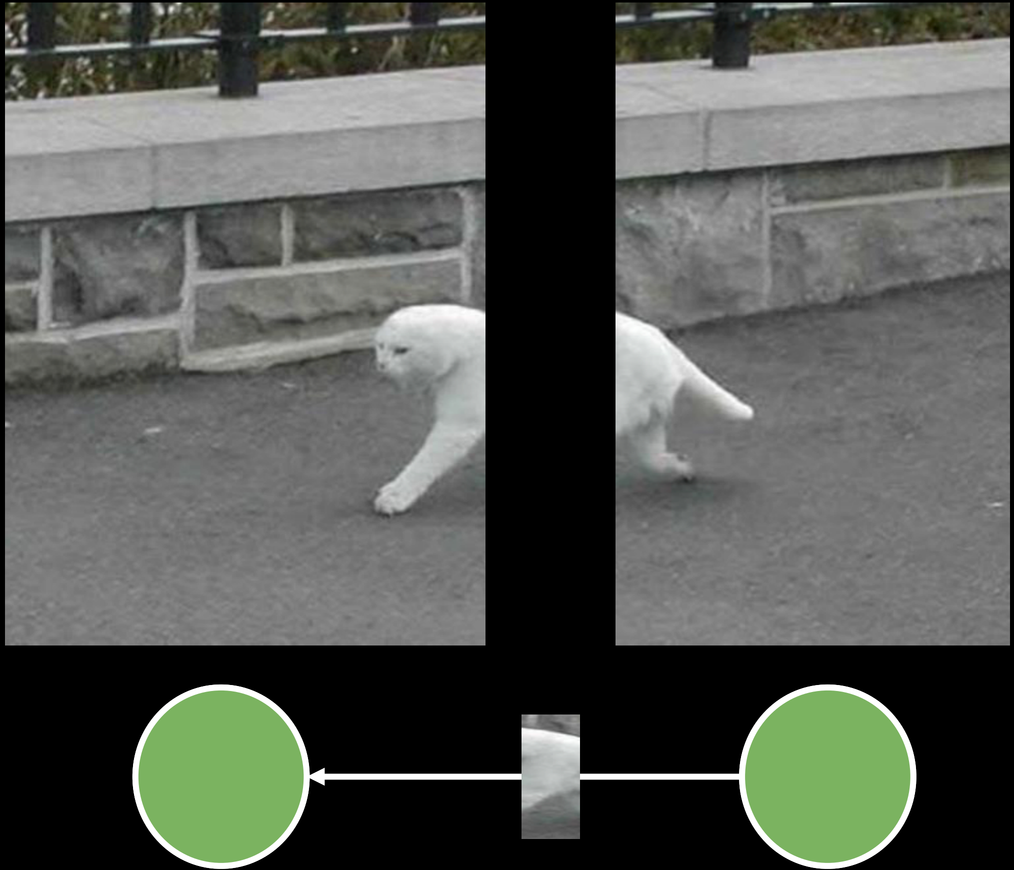
Always ask yourself: is this channel buffered?

In general, don’t use locks or atomic operations with channels (awkward)

Try not to nest goroutines (hard to reason about)

Need synchronization





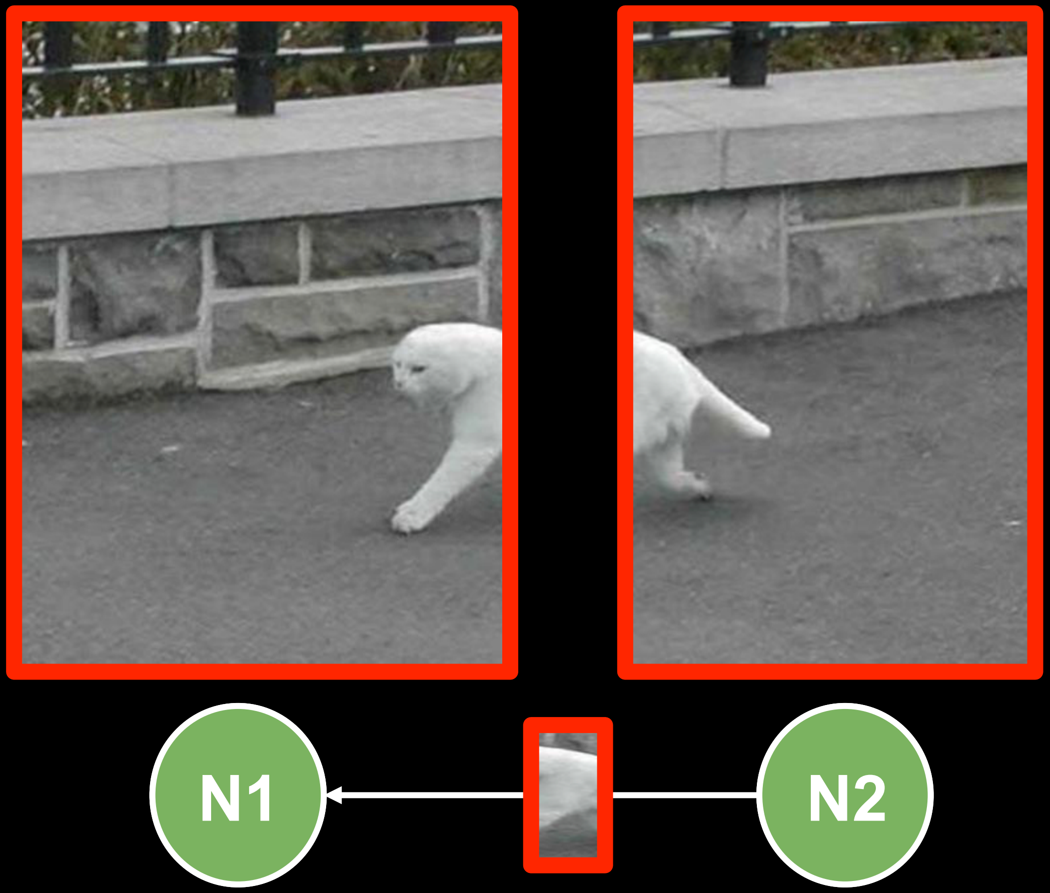
Distributed snapshots are easy to screw up

Must ensure state is not duplicated across the cluster

Must ensure state is not lost across the cluster

Messages in flight must also be recorded

But which ones?



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?

Chandy-Lamport snapshot algorithm

Key idea: Servers send marker messages to each other

Marker messages...

...mark the beginning of the snapshot process on the server

...act as a barrier (stopper) for recording messages

Refresher: system model

- N processes in the system with no process failures
  - Each process tracks some state
- Two FIFO unidirectional channels between every process pair P and Q
  - Channel also has state: the set of messages in the channel
  - All messages sent on channels arrive intact, unduplicated, in order

Chandy-Lamport snapshot algorithm

Starting the snapshot procedure on a server:

- Record local state
- Send marker messages on all outbound interfaces

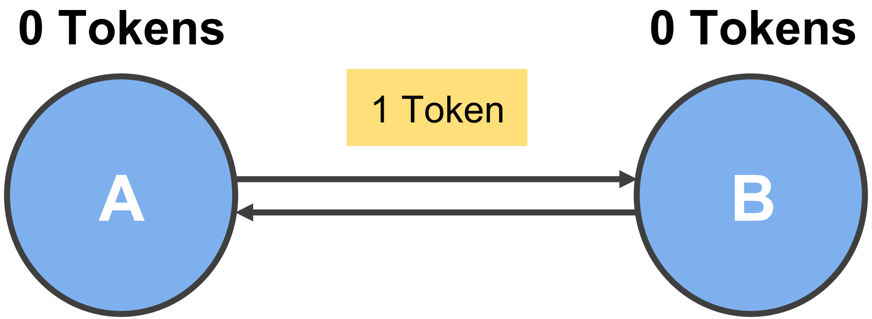
When you receive a marker message on an interface:

- If you haven't started the snapshot procedure yet, record your local state and send marker messages on all outbound interfaces
- Stop recording messages you receive on *this* interface
- Start recording messages you receive on all *other* interfaces

Terminate when all servers have received marker messages on all interfaces

Token passing example 1

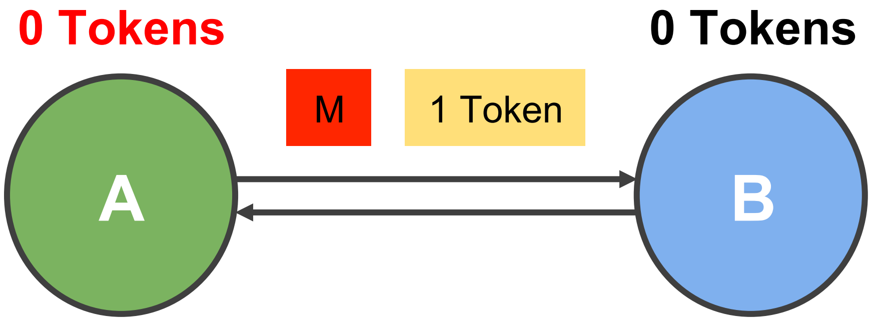
Token passing example 1



Event order:

1. *A* sends 1 token

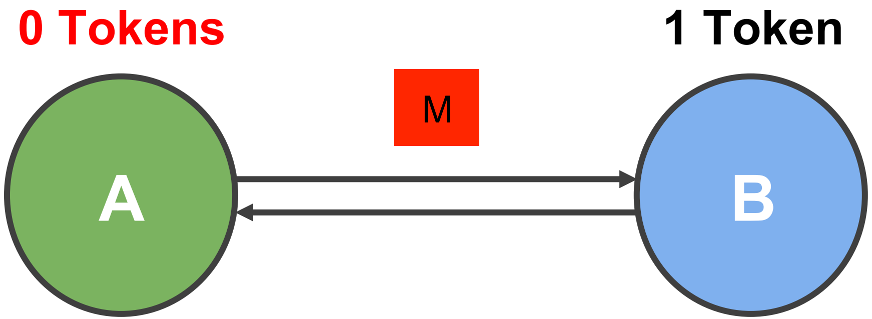
Token passing example 1



Event order:

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

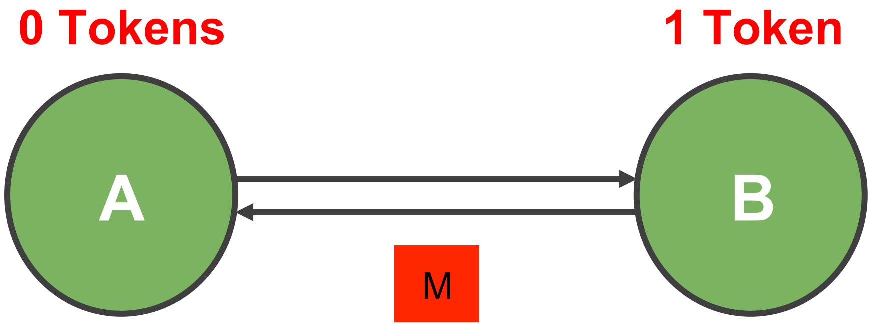
Token passing example 1



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:

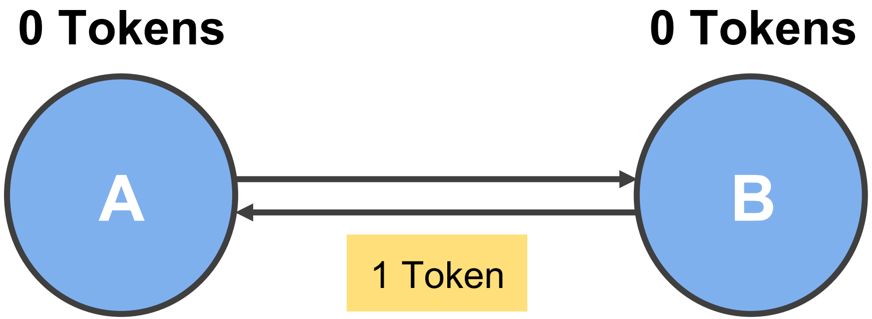
1. *A* sends 1 token
2. *A* starts snapshot, sends marker
3. *B* receives 1 token
4. *B* receives marker, starts snapshot
5. *A* receives marker, ends snapshot

Token passing example 2

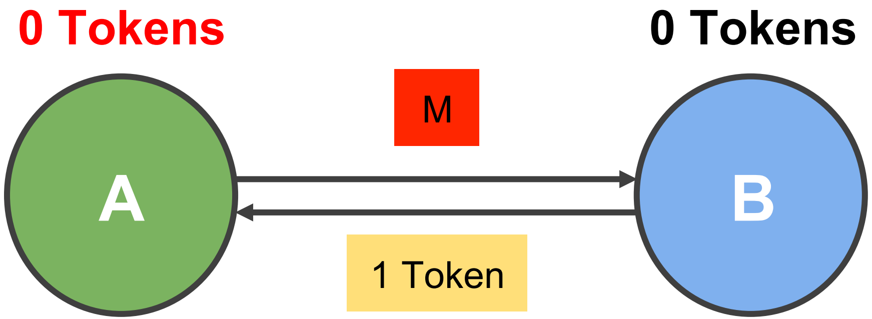
Token passing example 2

Event order:

1. *B* sends 1 token



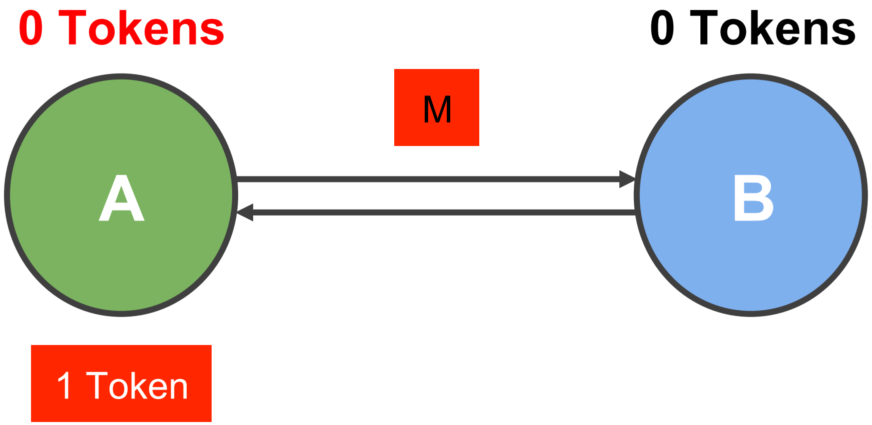
Token passing example 2



Event order:

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

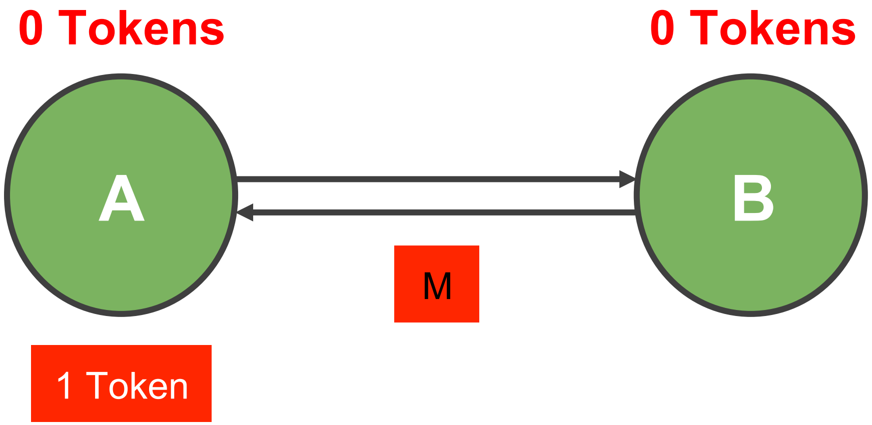
Token passing example 2



Event order:

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

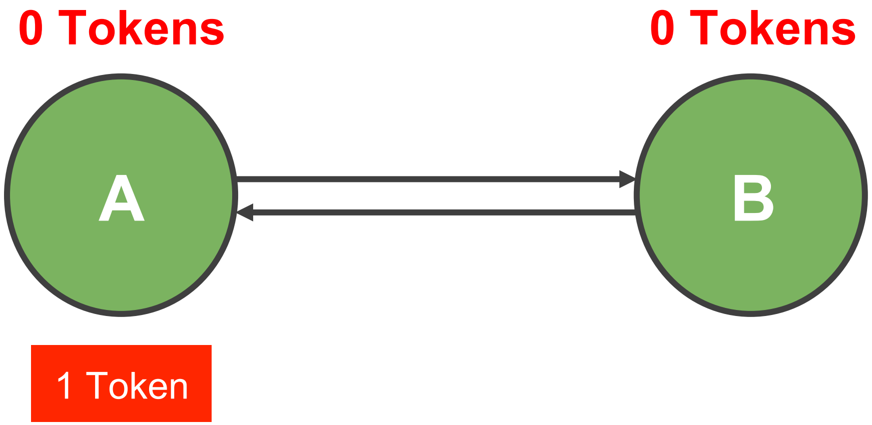
Token passing example 2



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

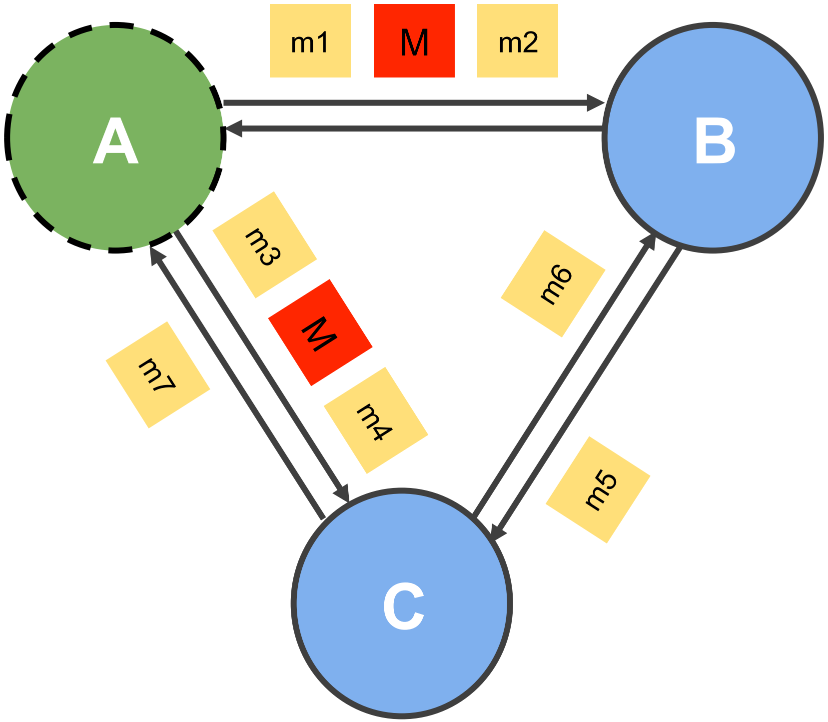


Event order:

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

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

Token passing example 3



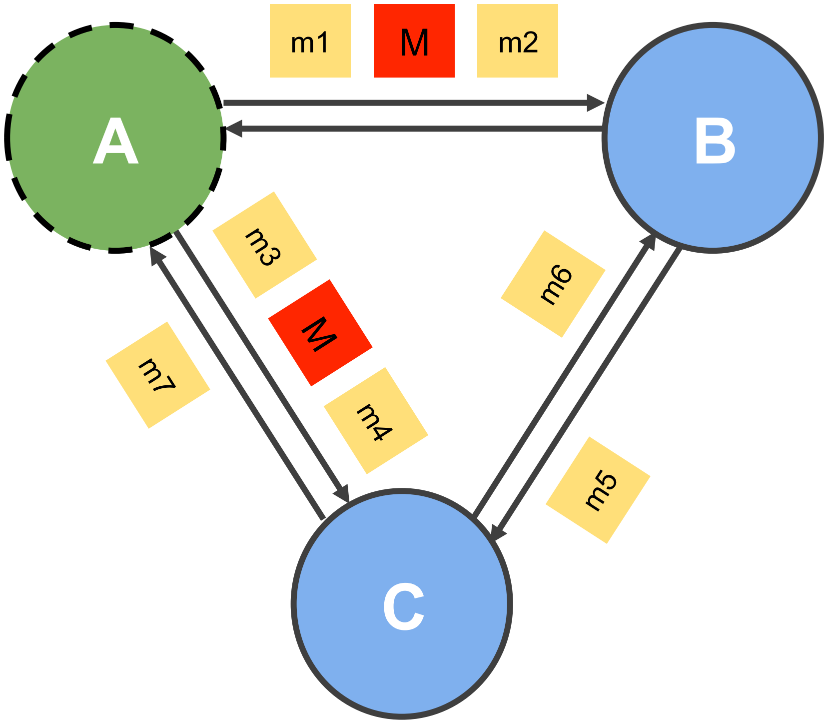
Which messages are definitely recorded*?

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

Assignment 2

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

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)
```

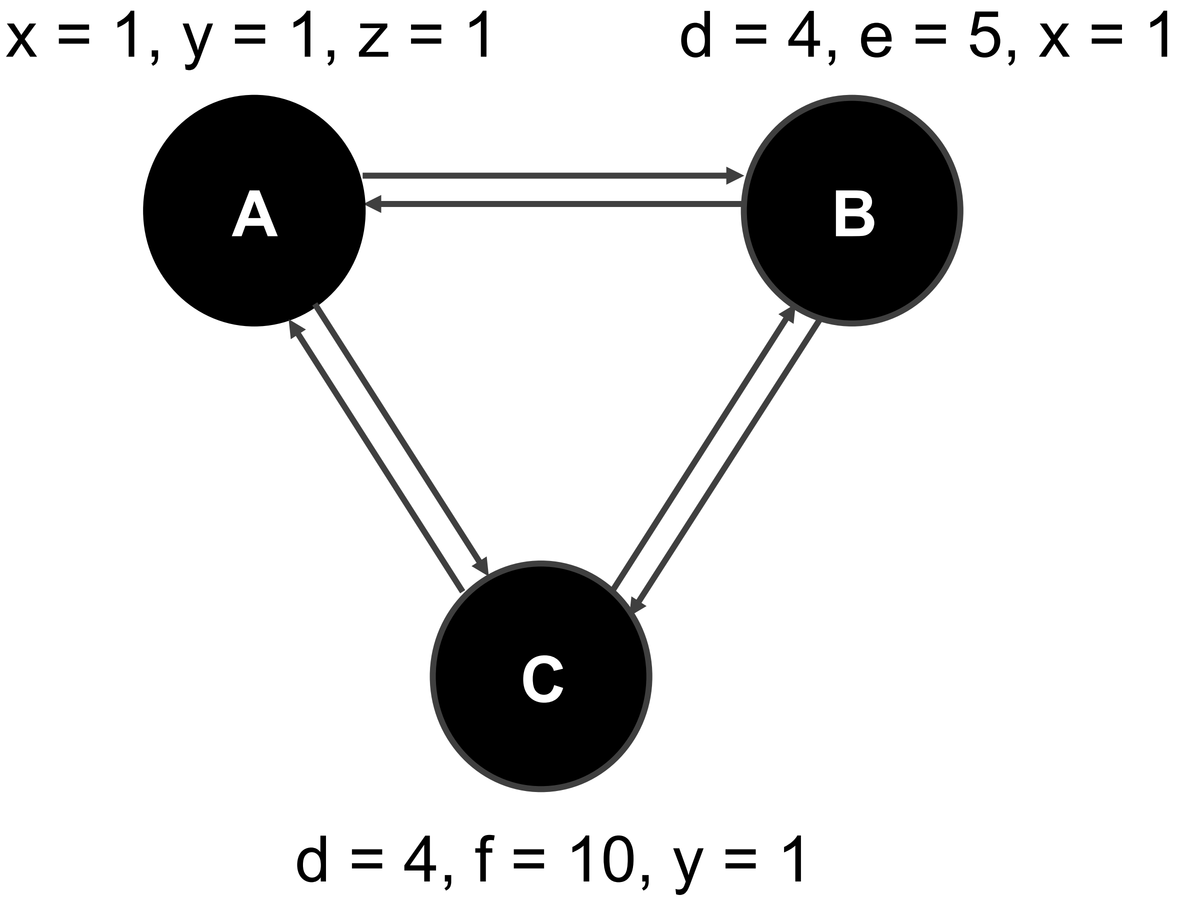
What kind of state does the server need to keep track of?

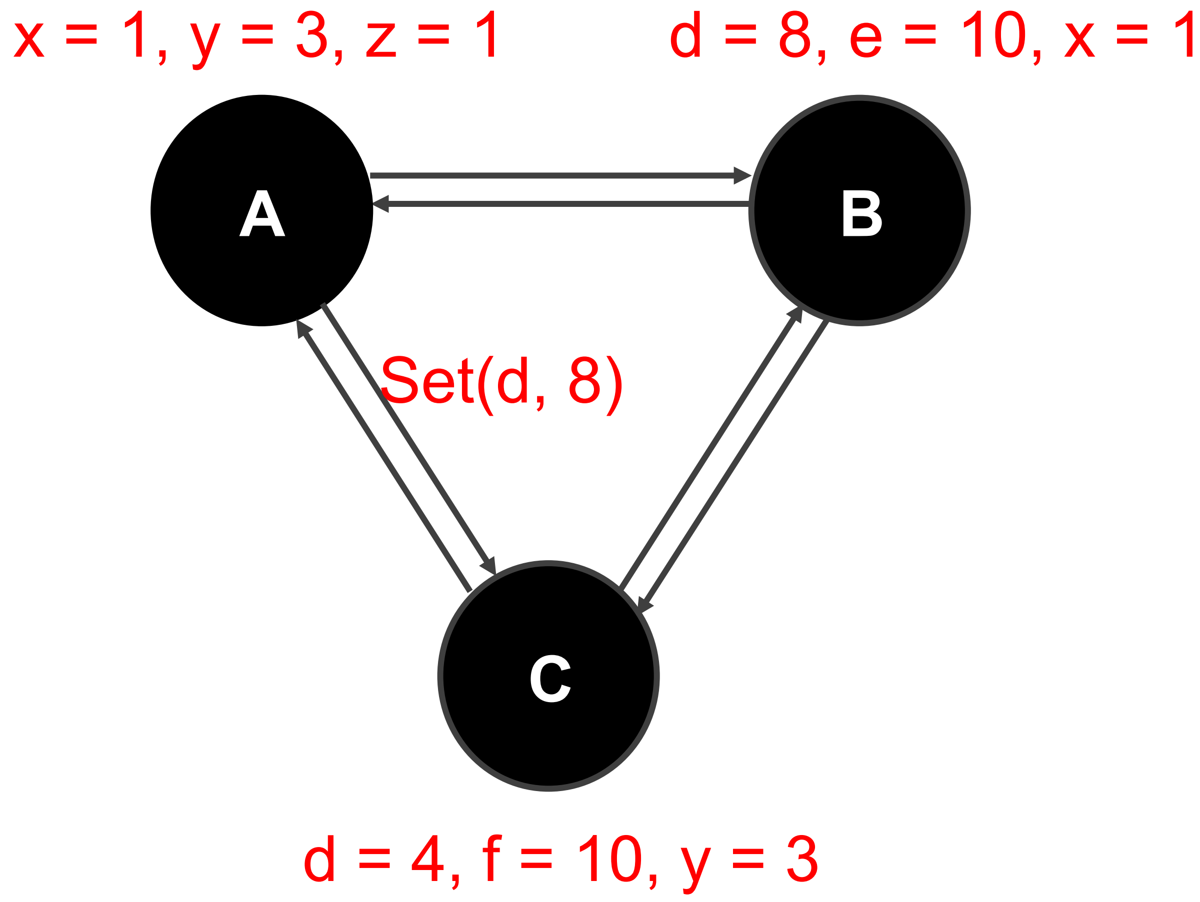
Local state, neighbors, which interfaces received markers, recorded messages

Assignment 2

Due 10/17 (Thursday) at 11:59pm!

Distributed database exercise


x = 1, y = 1, z = 1
d = 4, e = 5, x = 1
A
B
C
d = 4, f = 10, y = 1


x = 1, y = 3, z = 1
d = 8, e = 10, x = 1
A
B
Set(d, 8)
C
d = 4, f = 10, y = 3