Caltech

CS 142: Lecture 5.1
Mutual Exclusion

Richard M. Murray
28 October 2019

Goals:

- Introduce the concept of mutual exclusion (in distributed setting)
- Talk about how to share a variable between distributed processes

Reading:

- P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 7
- M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994. (Chapter 6: Distributed Mutual Exclusion)

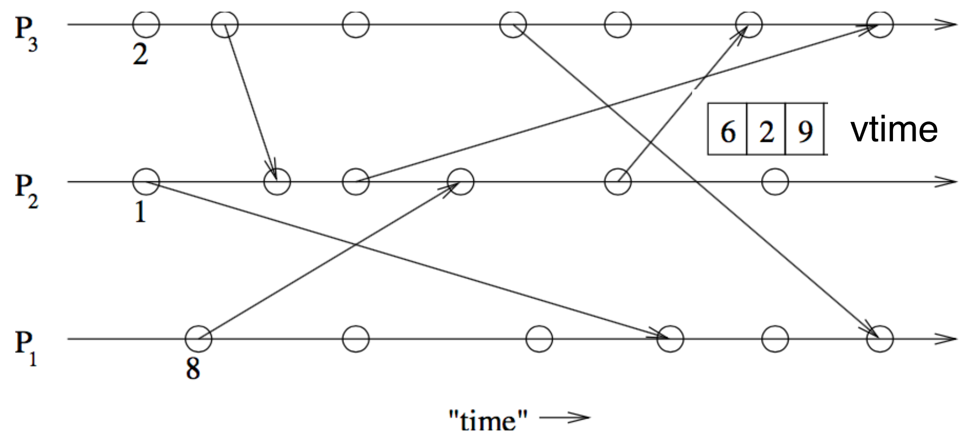
Summary: Time, Clocks, Synchronization

Channel model: FIFO, lossless, directed

Events, system timelines and logical time

- Can't assume process clocks agree
- Make use of "logical time"

$$A \longrightarrow B \;\Rightarrow\; time.A < time.B$$

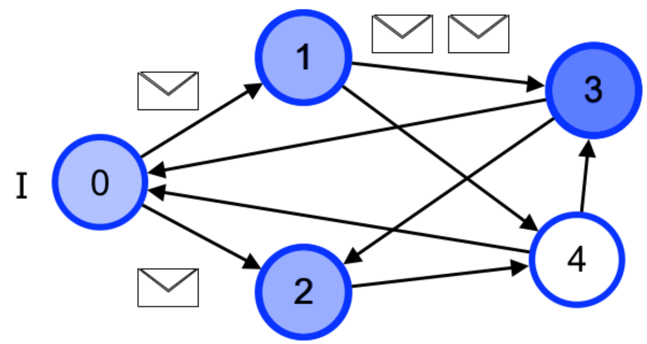


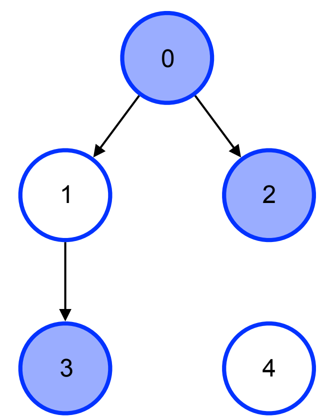
Vector clocks: $A \longrightarrow B \;\equiv\; vtime.A < vtime.B$

- Keep track of time in each process
- Order relation allows us to *know* one event occured before another

Gossip: distribute info to all nodes

- Key problem is understanding when the algorithm has terminated (all nodes idle, no information in channels)
- Use tree structure to track propagation

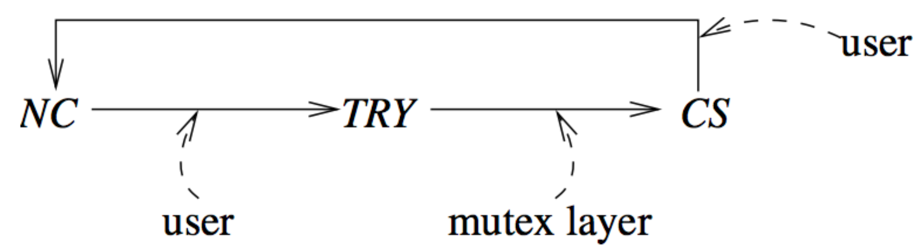




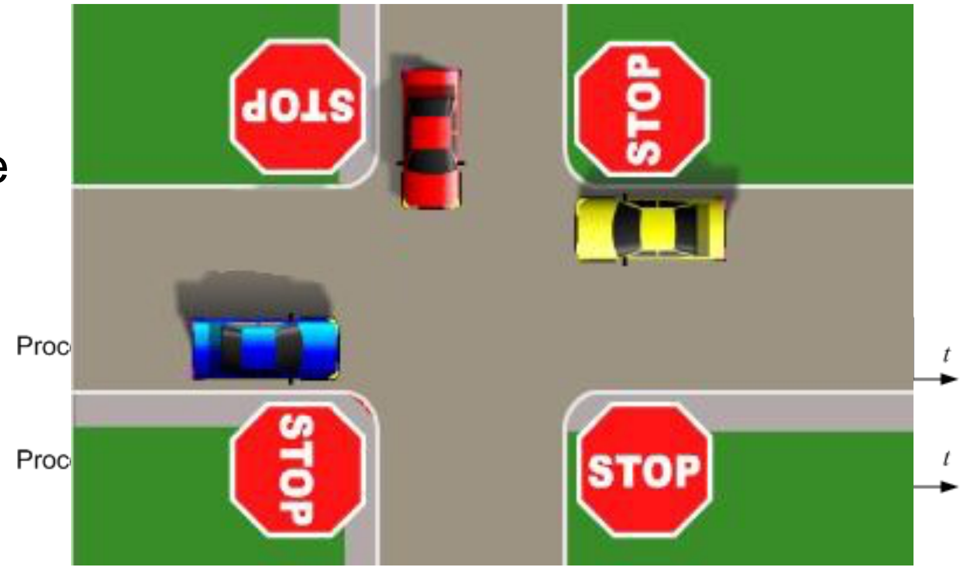
Diffusing computation properties:

- Safety: **invariant** (*claim* $\Rightarrow$ term. cond.)
- Progress: term. cond. $\rightsquigarrow$ *claim*

Today: mutual exclusion



The Mutual Exclusion Problem



Control access to a "critical section" (CS)

- Use in situations where no more than one agent can make use of a resource at a time
- Easy to implement in centralized setting
  - E.g. standard mutex libraries in Unix
- Not so easy when there is no central node and no central clock

Example: intersections for self-driving cars

- Safety: no two cars should be in the intersection at the same time
- Progress: all cars should eventually be allowed to go through the intersection

Traditional (human) protocol for mutual exclusion at intersections (4 way stop)

- First person to reach the intersection gets to go first
- If someone is already at the intersection when you arrive, they were first
- If two or more people arrive at the same time, right hand rule applies

Q1: what happens if four people arrive at the same time?

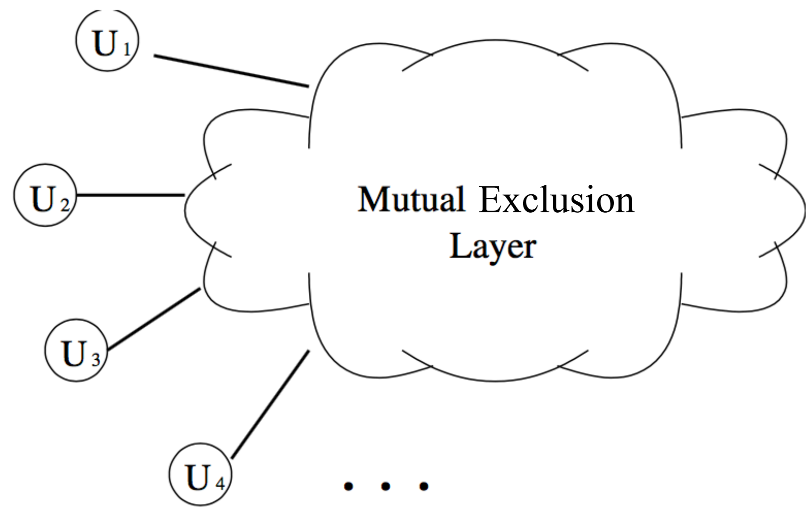
Q2: if [some] cars are self-driving, who decides who reaches intersection "first"?

- Should self-driving car give way to aggressive human? Even if they break protocol?

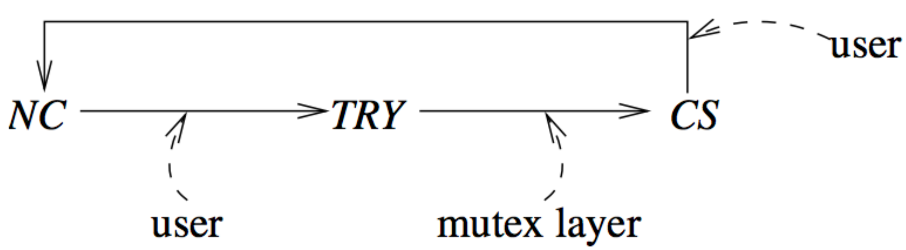
Mutual Exclusion Formal Problem Statement

Specification

- Safety: no two users (U_i) are in critical section (CS) at the same time
- Progress: strong and weak
  - Weak: some agent will eventually be allowed to enter CS
  - Strong: all agents will get a chance (as long as they keep requesting)



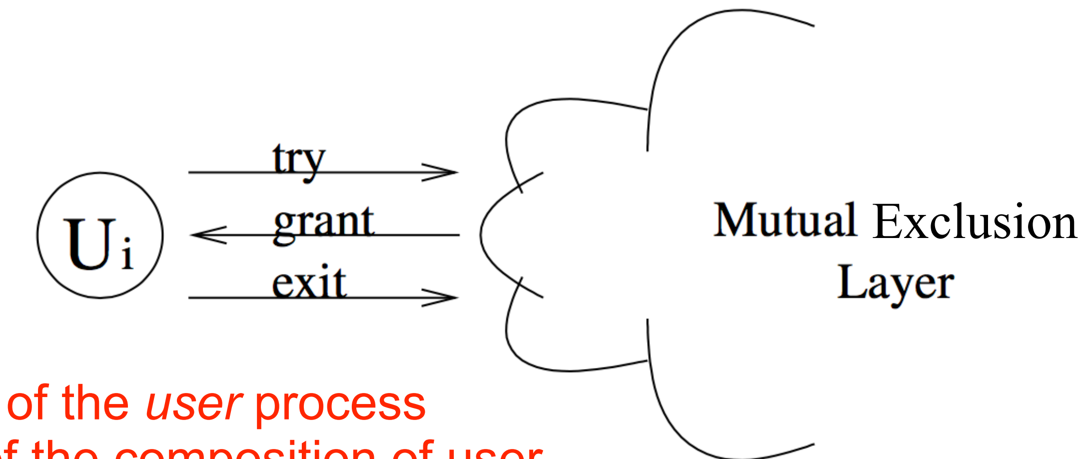
User process protocol



Composition properties:

TRY **next** TRY ∨ CS

TRY $\leadsto$ CS



User process (U_i) properties

NC **next** $NC \vee TRY$

stable.TRY

property of the *user* process but *not* of the composition of user processes & mutex layers

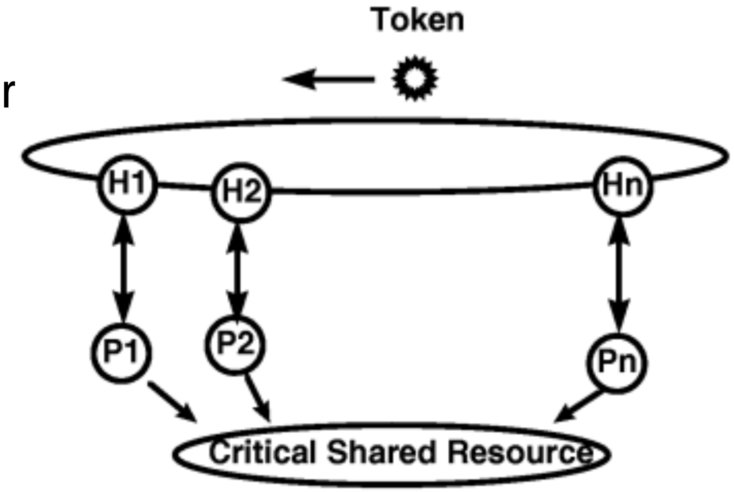
CS **next** $CS \vee NC$

transient.CS

Approaches to Mutual Exclusion

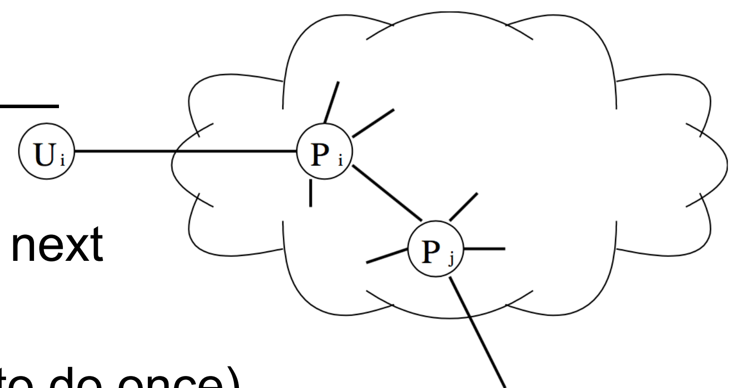
Centralized control process

- Easiest: everyone makes requests to central “allocator
- Use standard mutex at that point (eg, simple queue)
- Cons: ___________________________________



Token ring Fri

- Use an indivisible token to grant access
- Pass token around in an “efficient” way
- Pros: relatively easy to implement and verify
- Cons: ___________________________________

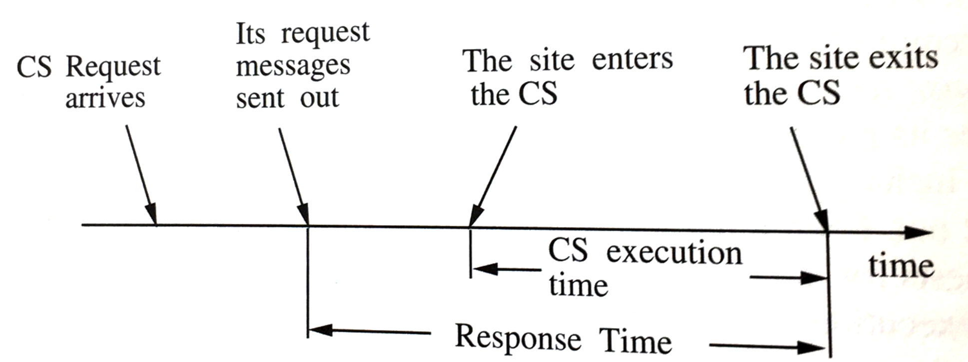


Distributed computation Today

- Create protocol by which everyone agrees on who is next
- Pros: works for arbitrary topologies
- Cons: slightly more complex to verify (but only need to do once)

Metrics for choosing an approach Fri

- Response time
- Number of messages required

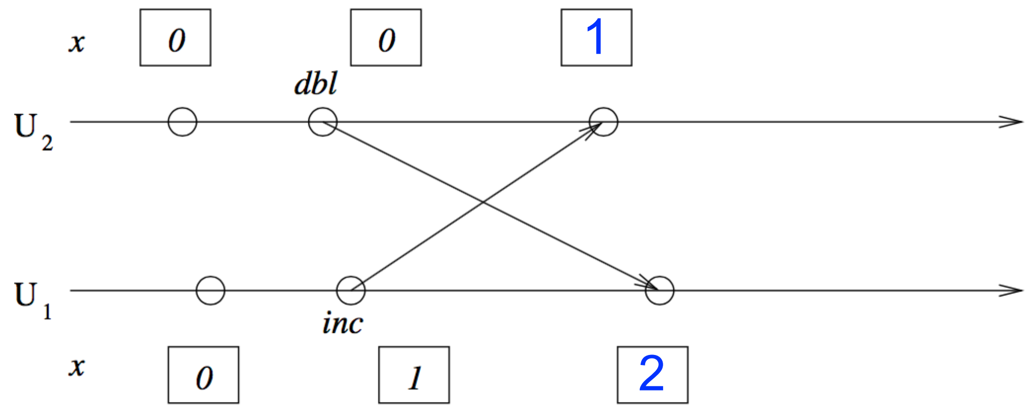


Related Problem: Distributed Atomic Variables

General question: how can we "synchronize" a variable in a distributed system?

Proposed algorithm:

- Local variables for each agent (i)
  - x = local copy of shared variable
  - ti = logical clock for agent i
  - queue of modify requests
  - list of "known times" for all other processes (why: ____________)
- Agent executes modification request when
  - request has minimum logical time
  - all known times are later than the request time

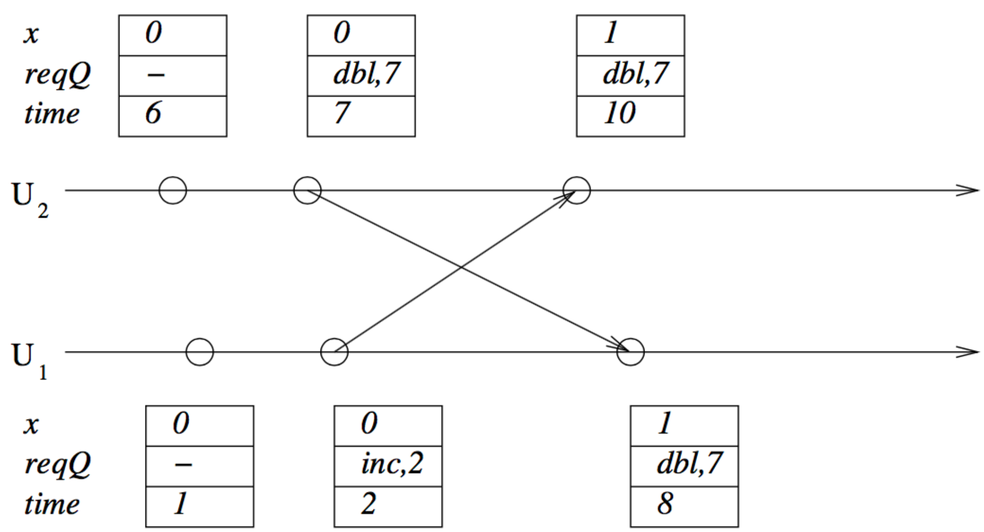


Key properties that make this work

- All agents agree on request order
- All agents know who has full information

Mutual exclusion is an example of this

- Use synchronized variable to agree on who gets to access critical section

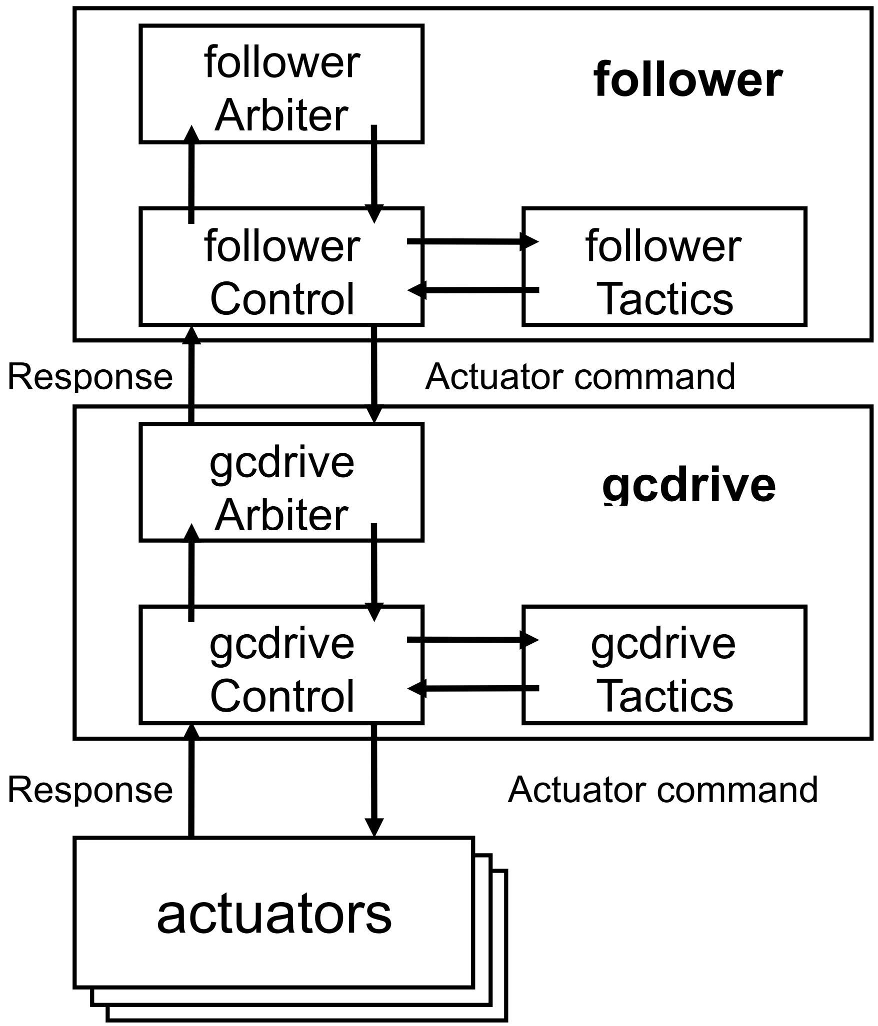


DGC Example: Changing Gear

Wongpiromsarn and M
CDC 2008

Verify that we can't drive while shifting or drive in the wrong gear

- Five components: follower Control, gcdrive Arbiter, gcdrive Control, actuators and network
- Construct temporal logic models for each component (including network)



Asynchronous operation

- Notation: $\text{Message}_{mod,dir}$ - message to/from a module; Len = length of message queue
- Verify: follower has the right knowledge of the gear that we are currently in, or it commands a full brake.
  - $\Box\, ((Len(TransResp_{f,r}) = Len(Trans_{f,s})) \wedge TransResp_{f,r}[Len(TransResp_{f,r})] = \text{COMPLETED} \Rightarrow Trans_f = Trans))$
  - $\Box\, (Trans_f = Trans \vee Acc_{f,s} = \text{-1})$
- Verify: at infinitely many instants, follower has the right knowledge of the gear that we are currently in, or we have hardware failure.
  - $\Box\Diamond\, (Trans_f = Trans = Trans_{f,s}[Len(Trans_{f,s})] \;\vee\; HW\ failure)$

Application Example: Trusted Wingman

Problem description

- UAV (unmanned aerial vehicle) flies close as long as high bandwidth link is available
- Assume low speed link is always available

Temporal logic specification

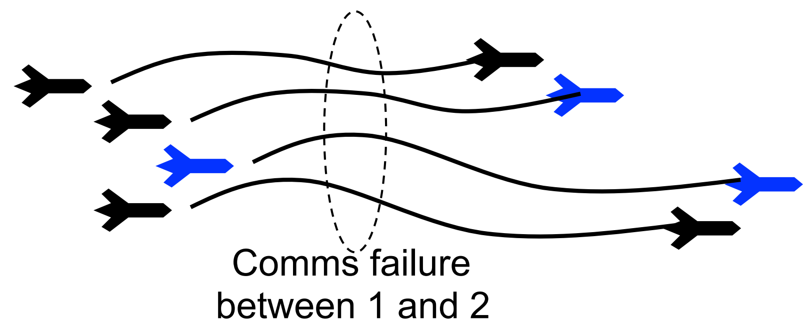
$$\text{mode} = \text{lost} \rightsquigarrow \textbf{stable}(d(x_l, x_f) > d_{\text{sep}})$$

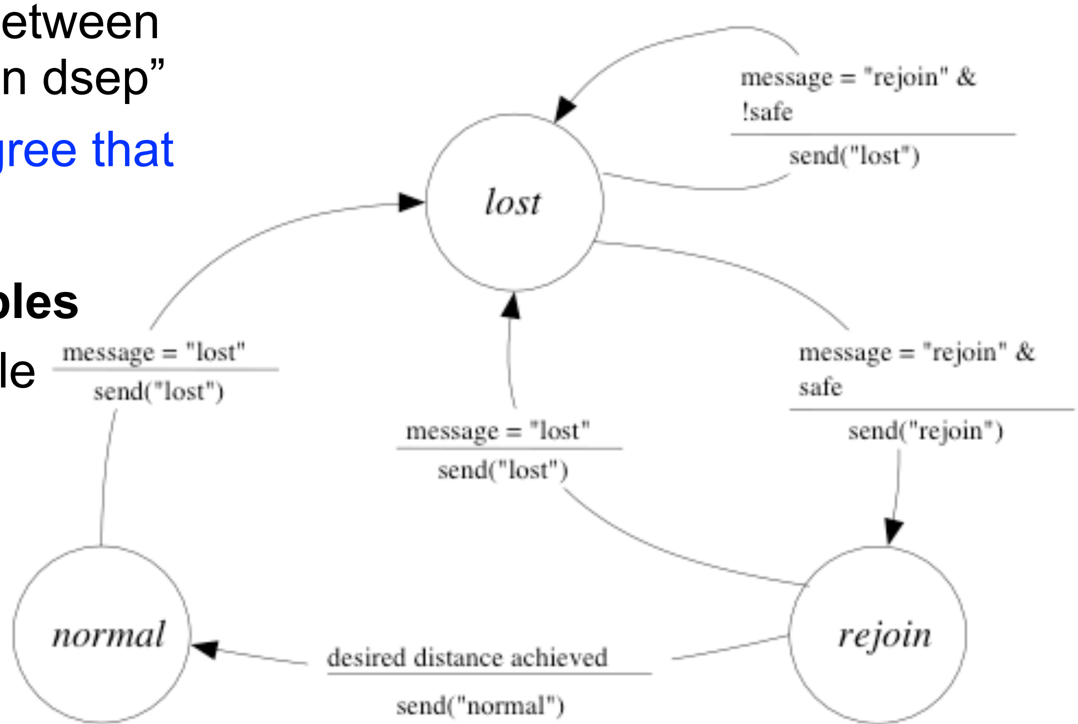
- "Lost mode leads to the distance between the aircraft always being larger than dsep"
- Need to make sure both aircraft agree that high speed link is lost

Implementation using shared variables

- Implement using distributed variable to keep track of system "mode"
- Also allows extension to multiple aircraft (eg, rest of the formation)

Lost wingman in fingertip formation





Lamport's Mutual Exclusion Algorithm

Idea: treat request queue as a distributed atomic variable

- reqQ: queue of timestamps requests for CS (sorted in ________ order)
- knownT: list of last "known times" for other processes
- Actions
  - Request entry: add to reqQ; broadcast <reqi, ti> to all other processes
  - Receive req: add to reqQ; send <acki, ti>
  - Receive ack: update knownT[j]
  - Receive release: remove Uj's request from reqQ
  - Conditions to enter CS
    - L1: req at head of reqQ
    - L2: knownT[j] > ti for all other j
  - To release CS
    - remove req from reqQ
    - broadcast <releasei> message

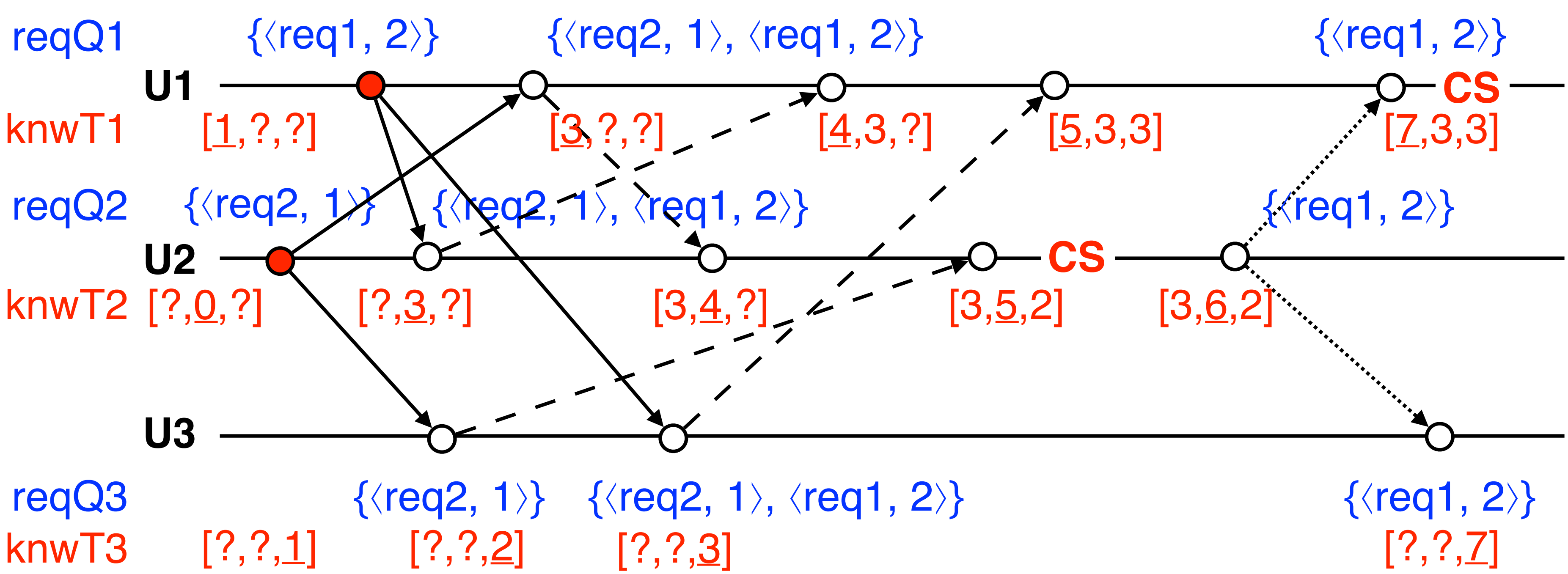
UNITY program: list of actions that can be executed by each agent (in any order)

- SendReq: $\text{mode} = \text{NC} \rightarrow \text{mode} = \text{TRY} \parallel (\forall j :: \text{send}(i, j, \langle \text{req}_i, t_i \rangle))$
- RecvReq: $(\exists j :: \text{recv}(i, j) = \langle \text{req}_j, t_j \rangle \rightarrow \text{recQ.push/sort}(\langle \text{req}_j, t_j \rangle) \parallel \text{send}(i, j, \langle \text{ack}_i, t_i \rangle))$
- RecvAck: $(\exists j :: \text{recv}(i, j) = \langle \text{ack}_j, t_j \rangle \rightarrow \text{knownT}[j] := t_j)$
- EnterCS: $\text{mode} = \text{TRY} \wedge \text{recQ[head]} = \langle \text{req}_i, t_i \rangle \wedge (\forall j :: \text{knownT}[j] > ti) \rightarrow \text{mode} = \text{CS};$
- ReleaseCS: $\text{mode} = \text{CS} \rightarrow \text{mode} = \text{NC} \parallel \text{reqQ.pop}(\langle \text{req}_i, t_i \rangle \parallel (\forall j :: \text{send}(i, j, \langle \text{rel}_i, t_i \rangle))$
- RecvRel: $(\exists j :: \text{recv}(i, j) = \langle \text{rel}_j, t_j \rangle \rightarrow \text{reqQ.pop}(\langle \text{rel}_j, t_j \rangle)$

Sample Execution

- {SendReq:} mode = NC → mode = TRY || (∀j :: send(i, j, ⟨req_i, t_i⟩))
- {RecvReq:} (∃j :: recv(i, j) = ⟨req_j, t_j⟩ → recQ.push/sort(⟨req_j, t_j⟩) || send(i, j, ⟨ack_i, t_i⟩))
- {RecvAck:} (∃j :: recv(i, j) = ⟨ack_j, t_j⟩ → knownT[j] := t_j)
- {EnterCS:} mode = TRY ^ recQ[head] = ⟨req_i, t_i⟩ ^ (∀j :: knownT[j] > ti) → mode = CS;
- {ReleaseCS:} mode = CS → mode = NC || reqQ.pop(⟨rel\q, t_i⟩ || (∀j :: send(i, j, ⟨rel_i, t_i⟩))
- {RecvRel:} (∃j :: recv(i, j) = ⟨rel_j, t_j⟩ → reqQ.pop(⟨ack_j, t_j⟩)

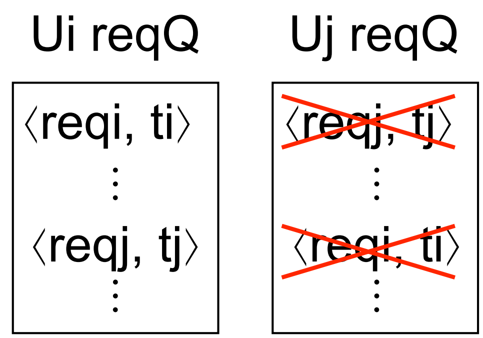
recQ: {⟨reqj, tj⟩, …}
knwT1: [log, kT2, kT3]



Proof of Correctness

Safety: need to show that no two processes are in CS at the same time

- Assume the converse: Ui and Uj are both in CS
- Both Ui and Uj must have their own requests at head of queue
- Head of Ui: <reqi, ti>
- Head of Uj: <reqj, tj>
- Assume WLOG ti < tj (if not, switch the argument)
- Since Uj is in its CS, then we must have tj < Uj.knownT[i]
  $\Longrightarrow$ <reqi, ti> must be in Uj.reqQ (since messages are FIFO)
- ti < tj $\Longrightarrow$ reqj can't be at the head of Uj.reqQ
- $\rightarrow\leftarrow$ (contradiction)



Progress: need to show that eventually every request is eventually processed

- Approach: find a metric that is guaranteed to decrease (or increase)
- One metric: number of entries in Ui.knownT that are less than its request time (ti)
  - Represents number of agents who might not have received our request
- Is this a good metric? Check conditions that are needed for induction:
  - Bounded below by zero and if at zero then we eventually enter our critical section
  - Must always decrease as other processes enter their critical section (and *someone* will execute their CS at some point in time)

Summary: Mutual Exclusion

Key ideas:

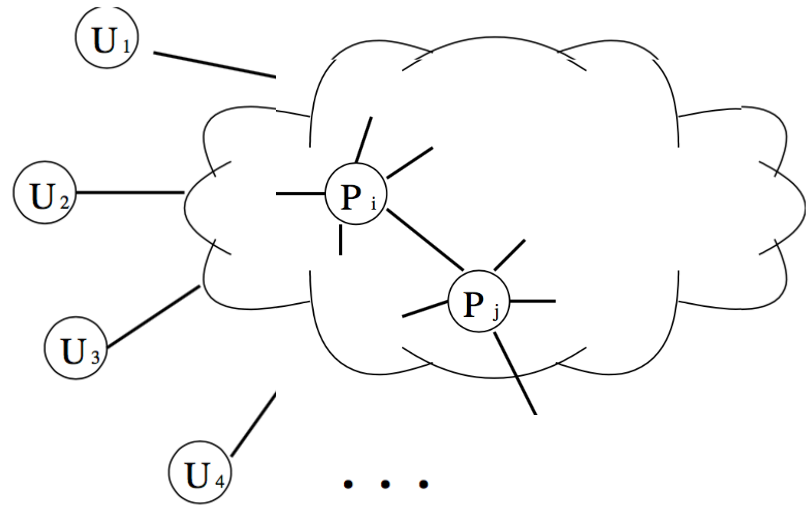
- Distributed protocol for allow access to a shared resource (“critical section”)
- Can treat as special case of distributed atomic variables
- *User* process specifications:

$$NC \ \mathbf{next} \ NC \vee TRY$$

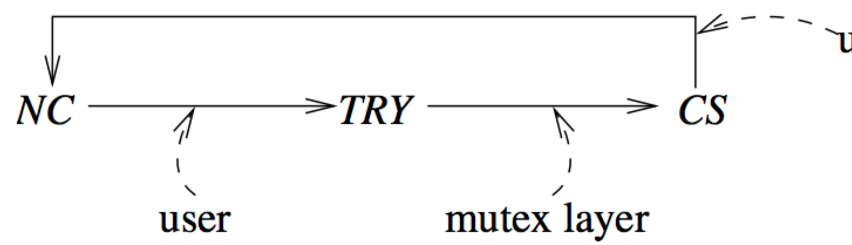
$$\mathbf{stable}.TRY$$

$$CS \ \mathbf{next} \ CS \vee NC$$

$$\mathbf{transient}.CS$$



- System specifications:
  - Safety: no two users (Ui) are in critical section (CS) at the same time
  - Progress: all agents will get a chance (as long as they keep requesting): TRY $\rightsquigarrow$ CS



TRY **next** TRY $\vee$ CS

Good example of *composition* between user and system processes and specs

Friday: optimizations + token-based algorithms