Caltech

CS 142: Lecture 5.2
Mutual Exclusion

Richard M. Murray

1 November 2019

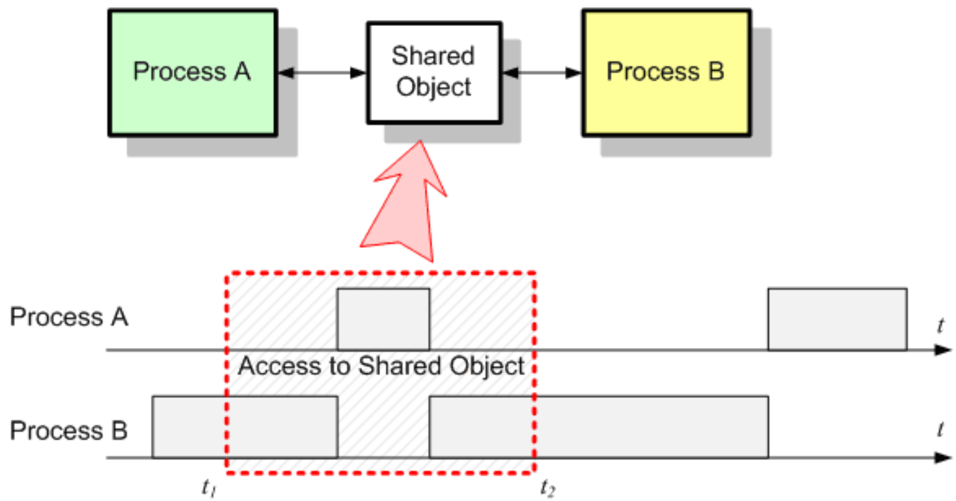
Goals:

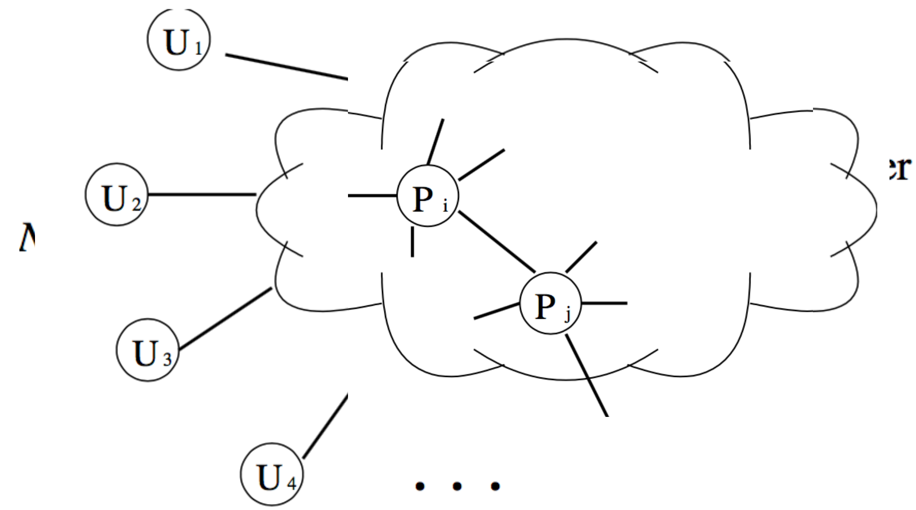
- Review Lamport's algorithm (from Mon) and prove correctness
- Describe token-based algorithms for mutual exclusion (ring + tree)
- If time: performance comparison

Reading:

- P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 7
- [SS94] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994. (Chapter 6: Distributed Mutual Exclusion)

Lamport's Mutual Exclusion Algorithm





Idea: treat request queue as a distributed atomic variable

- reqQ: queue of timestamps requests for CS (sorted in increasing order)
- knownT: list of last "known times" for other processes

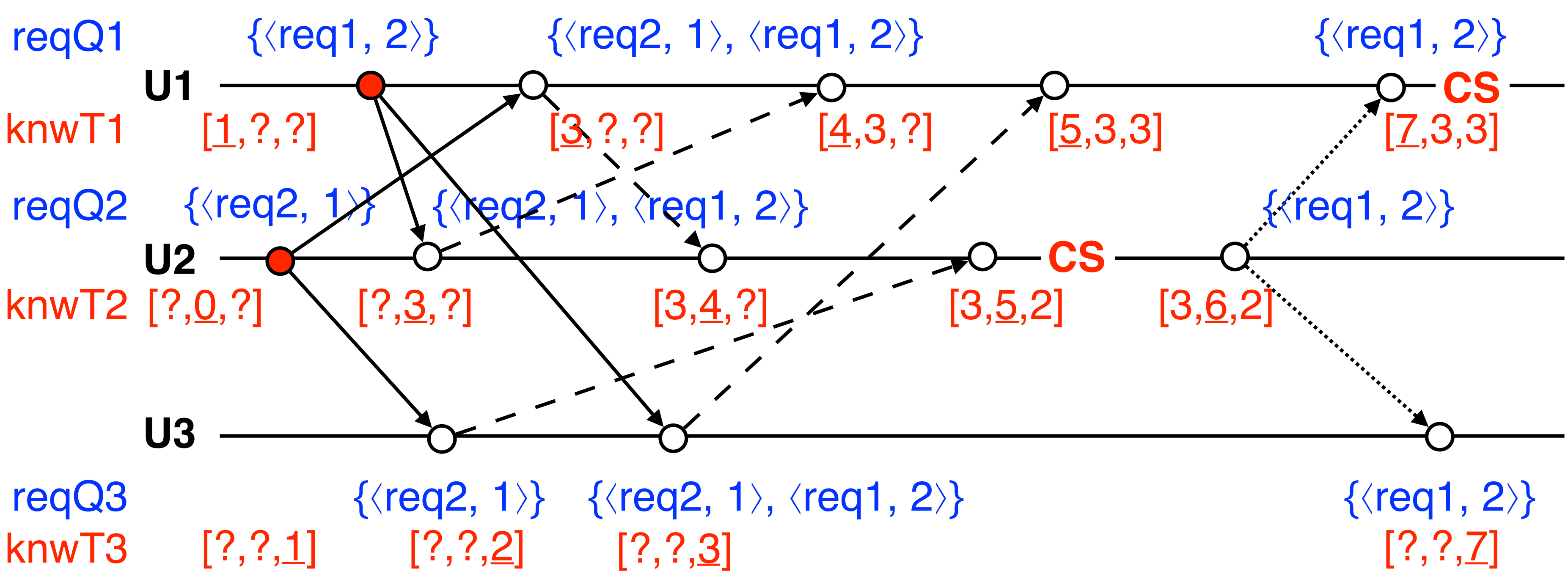
UNITY program: list of actions that can be executed by each agent (in any order)

- SendReq: mode = NC → mode := TRY || (∀j :: send(i, j, ⟨req_i, t_i⟩))
- EnterCS: mode = TRY ^ recQ[head] = ⟨req_i, t_i⟩ ^ (∀j :: knownT[j] > ti) → mode := CS;
- ReleaseCS: mode = CS → mode := NC || reqQ.pop(⟨req_i, t_i⟩ || (∀j :: send(i, j, ⟨rel_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)
- RecvRel: (∃j :: recv(i, j) = ⟨rel_j, t_j⟩ → reqQ.pop(⟨rel_j, t_j⟩)

Sample Execution

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

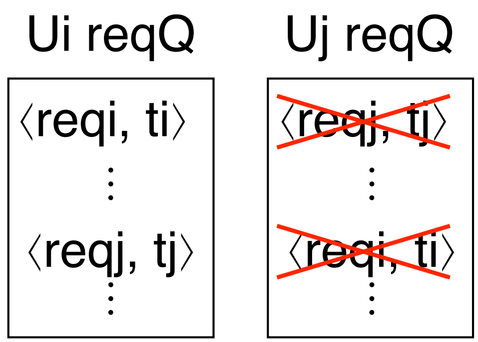
recQ: {⟨reqj, tj⟩, …}
knwT1: [log, kT2, kT3]



Proof of Correctness

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

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



$t_i < t_j < U_j.knownT[i] \Rightarrow U_j$ has acknowledged req_i

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 $U_i.knownT$ that are less than its request time (t_i)
  - Represents number of agents who might not have received our request
- Is this a good metric?
  - 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)

Proof of Correctness (Progress)

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

Proof steps

- Metric: number of entries in U_i.knownT that are less than its request time (t_i)
- Need to show that this is guaranteed to decrease $\Rightarrow$ eventually U_i can enter CS
- Consider an agent U_j with with an entry less than U_i's request time:

  U_i.knownT[j] < t_i (where $\langle req_i, t_i \rangle$ is the request from U_i)

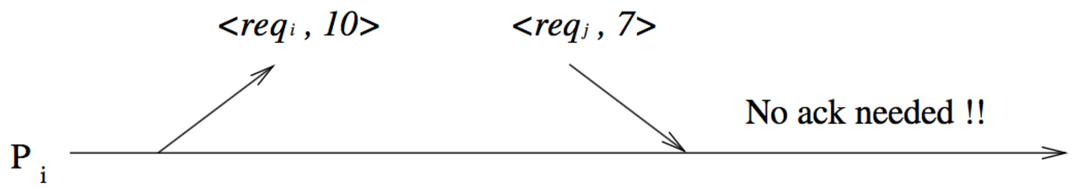
- Agent U_j's logical time is guaranteed to increase when $\langle req_i, t_i \rangle$ is received by U_j
- Agent U_j will send U_i an acknowledgement with $t_j > t_i$ (logical clock property) => metric will decrease by 1

Additional steps for complete proof:

- ______________________________

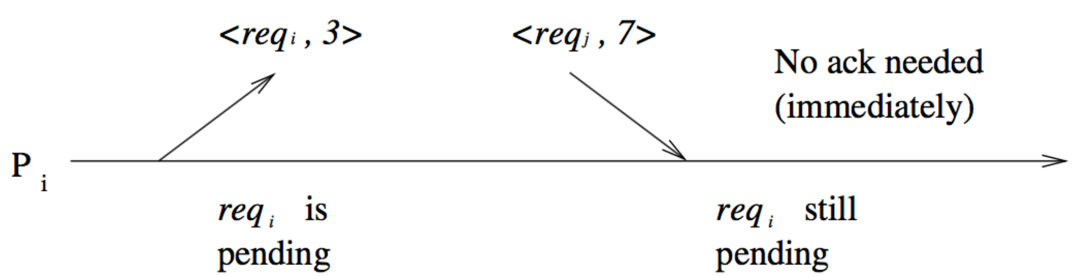
Optimizations on Lamport's Algorithm

Optimization #1 - if request sent with later timestamp than received, no ack needed



Optimization #2 (Ricart-Agrawala): merge release messages with replies

- Receive req: add to reqQ || send <ack$_i$, t$_i$> to U_j *only in certain conditions*:
  - if not requesting access to CS nor in CS
  - if requesting CS and U_j timestamp is smaller than Ui request
- When U_i exits CS, send release (clears all deferred acknowledgements)
- Idea: eventually, the pending request will be granted and we will send a <release> message, which will serve as the acknowledgement



Optimizations on Lamport's Algorithm

Optimization #1 - if request sent with later timestamp than received, no ack needed

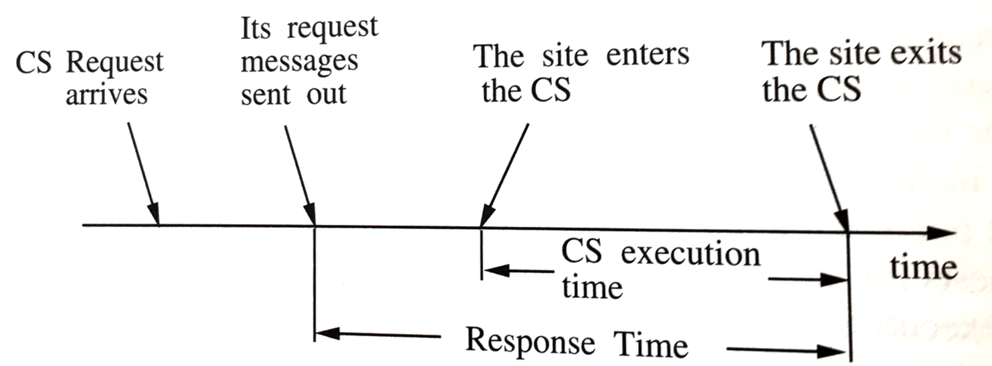
Optimization #2 (Ricart-Agrawala): merge release messages with replies

Optimization #3 (Maekawa): request permission from a subset of agents [SS94]

- U_i only sends requests to a subset of sites R_i, chosen such that $(\forall i,j : R_i \cap R_j \neq \{\})$
- Rough idea: every pair of agent has an agent that "mediates" between the pair
- Each agent sends only one ack at time; wait until a release is received => mediating agents can only grant permission if they haven't granted permission to another agent
- Proof is more complicated, but many fewer messages required

Timing analysis [SS94]

- Response time = amount of time to execute critical section (if no queue)
- Sync delay: U_i exits CS to U_j enters CS
- T = transport time: E = execution time



NON-TOKEN	Resp. time (*ll*)	Sync Delay	Messages (*ll*)	Messages (*hl*)
Lamport	$2T+E$	T	$3(N-1)$	$3(N-1)$
Ricart-Agrawala	$2T+E$	T	$2(N-1)$	$2(N-1)$
Maekawa	$2T+E$	$2T$	$3\sqrt{N}$	$5\sqrt{N}$

ll = low load condition (one req at a time)
hl = high load cond. (U_i seldom in NC)

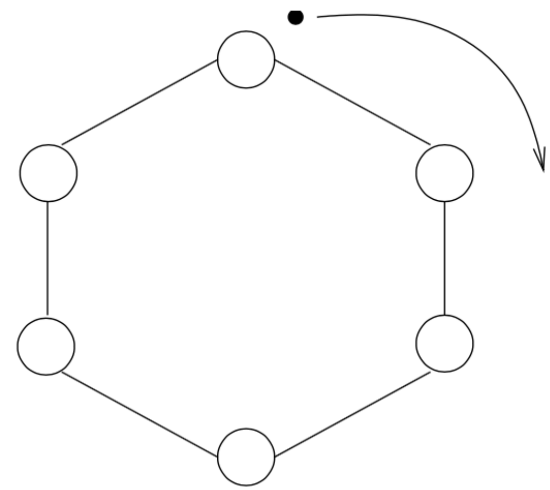
Token-Based Approaches to Mutual Exclusion

Simple token ring: send token clockwise (CW) around ring

- Problem: potentially long sync delays (especially in low load)

Token ring with requests: add request message type

- Minimize sync delay by waiting for requests to arrive (CCW)



SendReq (Try → CS)

if $holder \neq self$
  $hungry$:= **true**
  if $\neg asked$
    send $request$ (CCW)
    $asked$:= **true**
  wait until $using$
else
  $using$:= **true**
[use the resource]
$using$:= **false**
if $pending\_requests$
  send token on (CW)
  $pending\_requests$:= **false**

- Both approaches require existence of ring topology

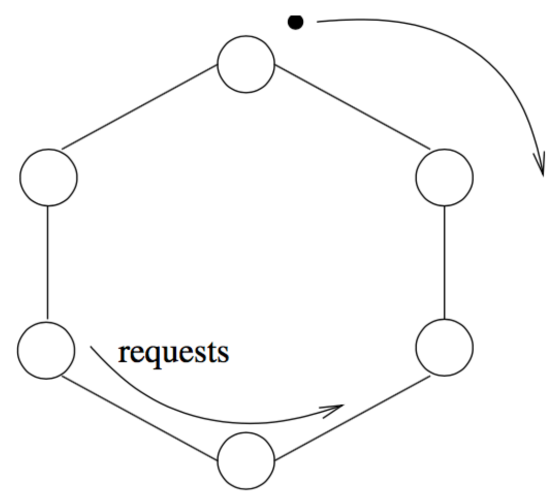
RecvReq

if $holder = self \ \wedge \ \neg using$
  send token on (CW)
else
  $pending\_requests$:= **true**
  if $holder \neq self \ \wedge \ \neg asked$
    send request (CCW)
    $asked$:= **true**

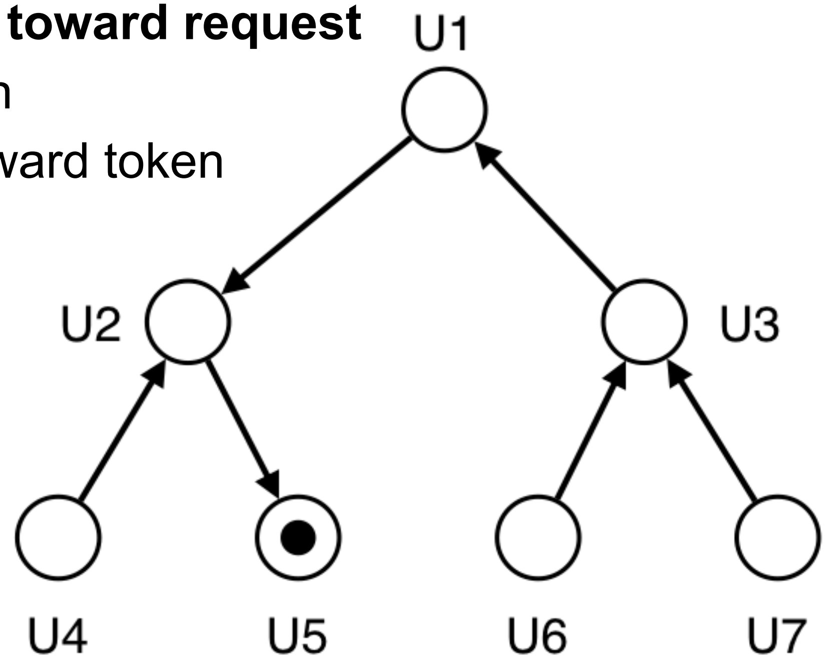
RecvToken

$asked$:= **false**
if $hungry$
  $using$:= **true**
  $hungry$:= **false**
else
  send token on (CW)
  $pending\_requests$:= **false**

hungry = Ui wants resource
asked = token requested
holder = token is with Ui
using = Ui in critical section



Token Tree Algorithm (Raymond)

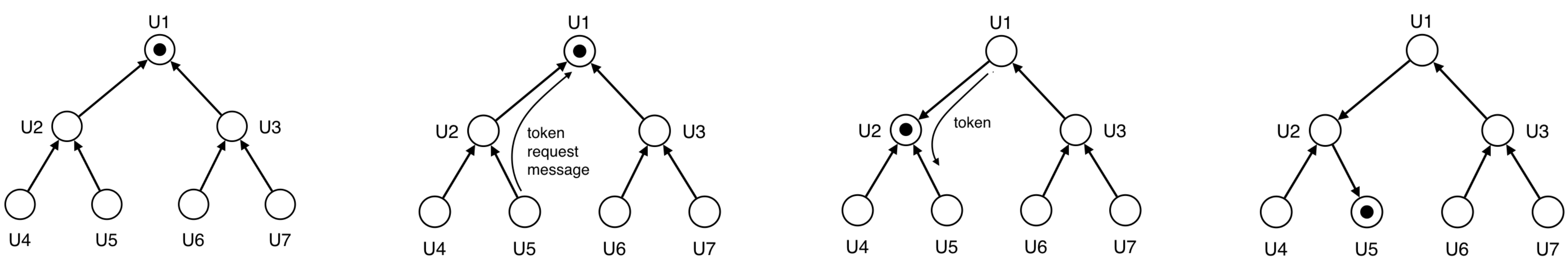


Basic idea: pass request toward token & token toward request

- Maintain a tree with root being agent with token
- Each agent gets requests and passes them toward token
- Agent with token either
  - enters CS (if earliest request)
  - passes token to node w/ earlier request
- Passing the token also updates the direction of links (so that root stays with the token)
- Invariant: graph is (rooted) tree, root at token

Question: how do we *prove* this works?

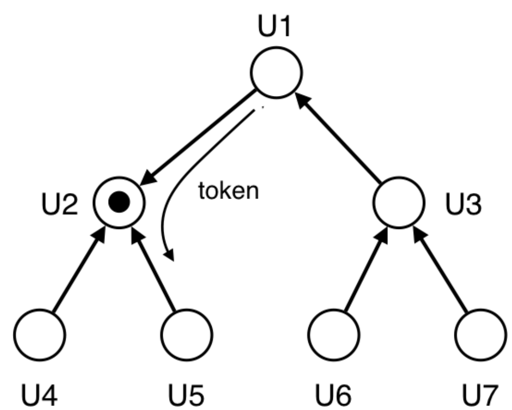
- Safety: no two nodes are in CS at same time (easy)
- Progress: need to show that all requesting nodes eventually get access
  - Trick (as usual): figure out a metric that captures this



Remarks on Token Tree Algorithm

Algorithm

1. If node wants CS, doesn't hold token and requestQ is empty, send request toward the root (= token location) and adds request to its requestQ
2. If agent receives request, place on requestQ and pass message toward the token
3. When root receives request, send token toward request and update parent link
4. When agent receives the token, remove top entry in requestQ, send token to that entry, and update parent link. If requestQ is non-empty, send request to (new) parent
   - Necessary step since we need token back to send to next entry in queue
5. Agent enters critical section when it has the token and its entry is at top of requestQ
6. After site has finished execution in critical session, got to step 4



Basic idea behind the proof [SS94]

- Agents execute requests in first come first serve (FCFS) order
- Let Ui = agent requesting access, Uh = agent holding the token
  - Always exists path Ui, Ui_1, Ui_2, … Ui_{k-1}, Ui_k, Uh
  - When Uh gets request from Ui_k, there are two possibilities
    - Ui_{k-1} is at the top of Ui_k's requisite => send token there => chain gets shorter
    - Some other agent Uj is at top queue => send token there first
    - Can show Ui_k will eventually get token back => will eventually get to Ui_{k-1} ($\leadsto$)

Comparisons between different algorithms

Timing analysis [SS94]

- Response time = amount of time to execute critical section (if no queue)
- Sync delay: Ui exits CS to Uj enters CS
- T = transport time: E = execution time
- ll = low load condition (one req at a time)
- hl = high load condition (Ui seldom in NC)

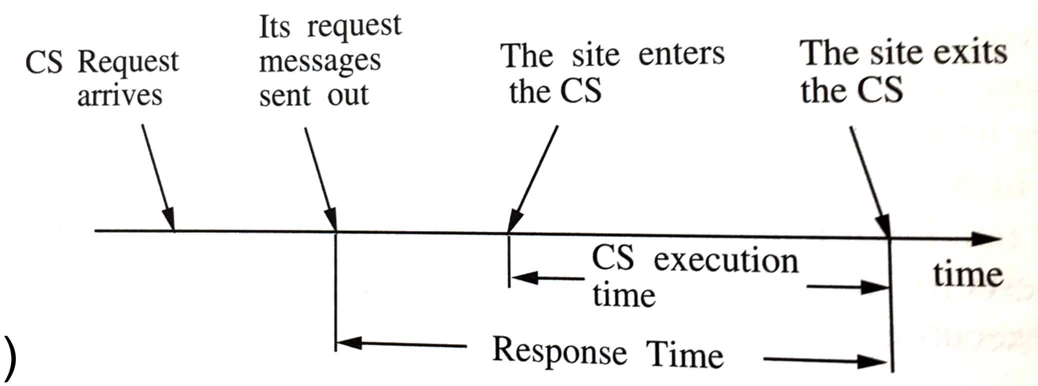


TABLE 6.1
A comparison of performance (*ll* = light load, *hl* = heavy load)

NON-TOKEN	Resp. time (*ll*)	Sync Delay	Messages (*ll*)	Messages (*hl*)
Lamport	$2T+E$	T	$3(N-1)$	$3(N-1)$
Ricart-Agrawala	$2T+E$	T	$2(N-1)$	$2(N-1)$
Maekawa	$2T+E$	$2T$	$3\sqrt{N}$	$5\sqrt{N}$
TOKEN	**Resp. time (*ll*)**	**Sync Delay**	**Messages (*ll*)**	**Messages (*hl*)**
Suzuki and Kasami	$2T+E$	T	N	N
Singhal's Heuristic	$2T+E$	T	$N/2$	N
Raymond	$T(\log N)+E$	$T\log(N)/2$	$\log(N)$	4

Summary: Mutual Exclusion

Key ideas:

- Distributed protocol for allow access to a shared resource ("critical section")
- Two approaches: distributed atomic variables (Lamport + variants) or token-based
- *User* process specifications:

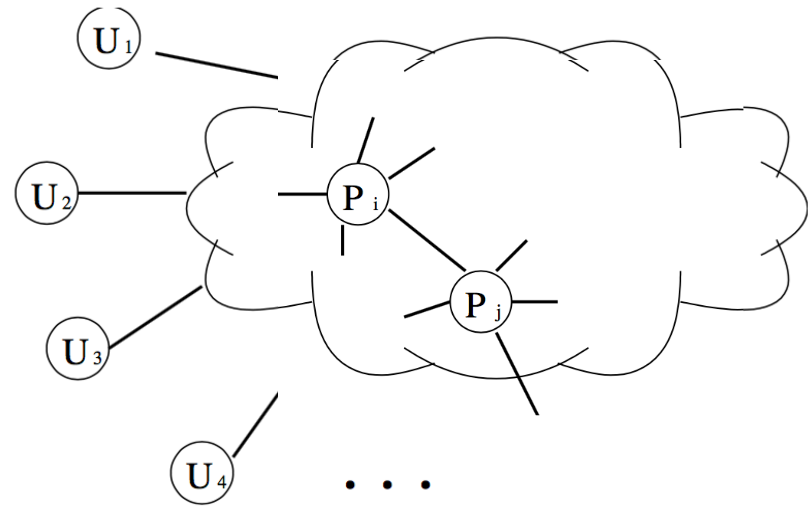
$$NC \ \mathbf{next} \ NC \vee TRY$$

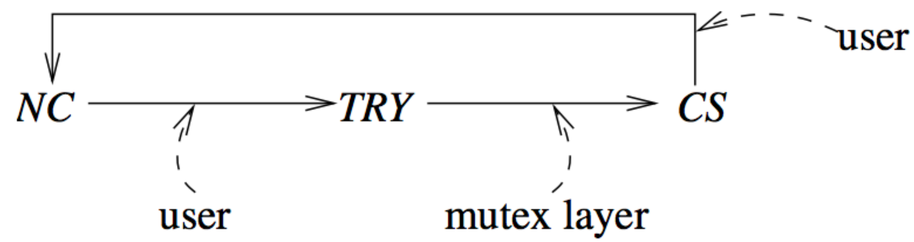
$$\mathbf{stable}.TRY$$

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

$$\mathbf{transient}.CS$$

- 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)





Composition properties: TRY **next** TRY ∨ CS

TRY $\leadsto$ CS

Friday: problem solving session (board talk)

Next week: specification refinement, conflict resolution (dining philosophers)