Computer Networks

EDA387/DIT663

Fault-tolerant Algorithms for Computer Networks

Review and Dijkstra's algorithm (Ch.2)

Goal

- We would like to understand how to design self-stabilizing network protocols
- At the end of these three lectures and after the home assignments you should be able to:
  - Define network tasks
    - leader election, token circulation, spanning tree construction (BFS) and network topology update (routing)
  - Propose methods for solving such tasks
    - with an emphasis on self-stabilizing methods
  - Argue about the correctness of their proposals

Today

- Defining system settings and tasks
- Solution for token circulation and spanning tree construction

What is a Distributed System?

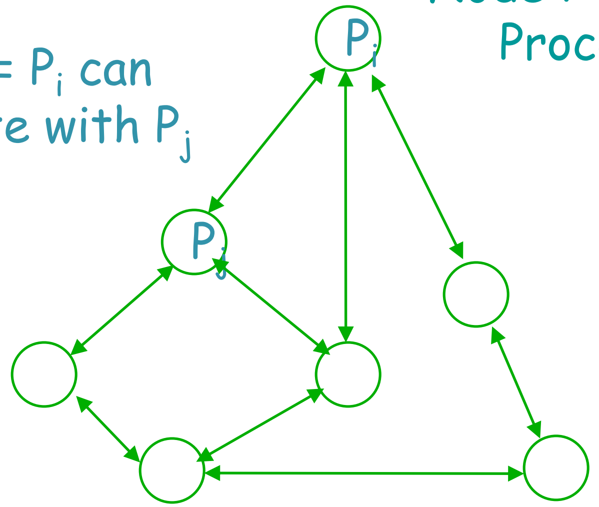
- Communication networks
- Multiprocessor computers
- Multitasking single processor

A computer network is modeled by a set of n finite state machines. We call these automata processors, and say that they communicate with each other

The Distributed System Model

Denote :

- P_i - the i^{th} processor
- neighbor of P_i - a processor that can communicate with directly with P_i



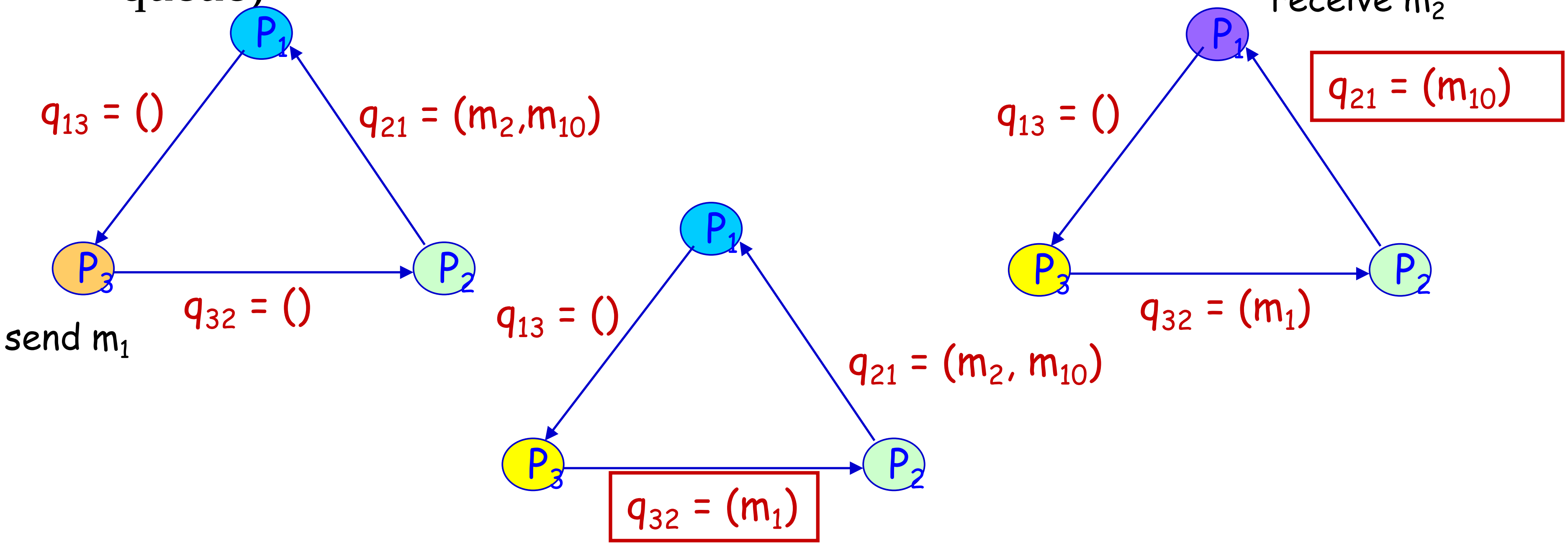
How to Represent the Model?

Ways of communication

- message passing - fits communication networks and all the rest
- shared memory - fits geographically close systems

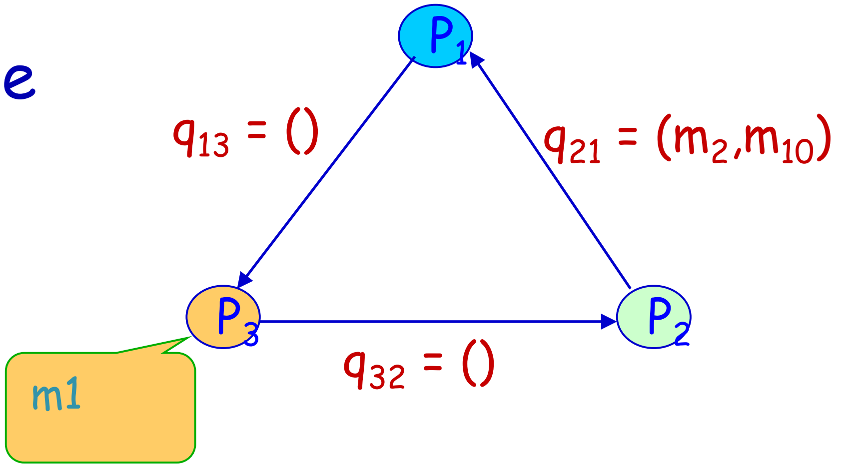
Asynchronous Distributed Systems – Message passing

- A communication link, which is unidirectional from P_i to P_j, transfers message from P_i to P_j
- For a unidirectional link we will use the abstract q_{ij} (a FIFO queue)

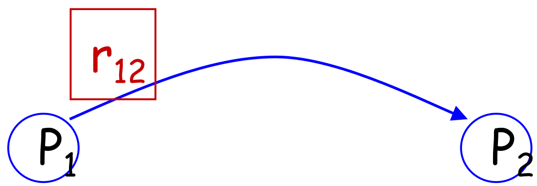


Asynchronous Distributed Systems - Message passing

- System configuration (configuration) : Description of a distributed system at a particular time.
- A configuration will be denoted by
  $C = (s_1, s_2, \ldots, s_n, q_{1,2}, q_{1,3}, \ldots, q_{i,j}, \ldots, q_{n,n-1})$, where
  s_i = State of P_i
  $q_{i,j}$ $(i \neq j)$ the message queue



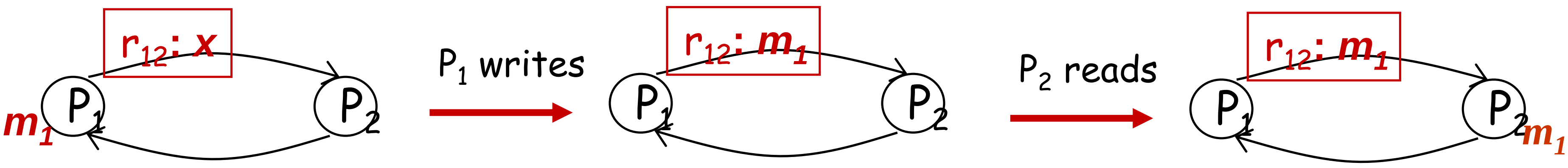
Asynchronous Distributed Systems – Shared Memory



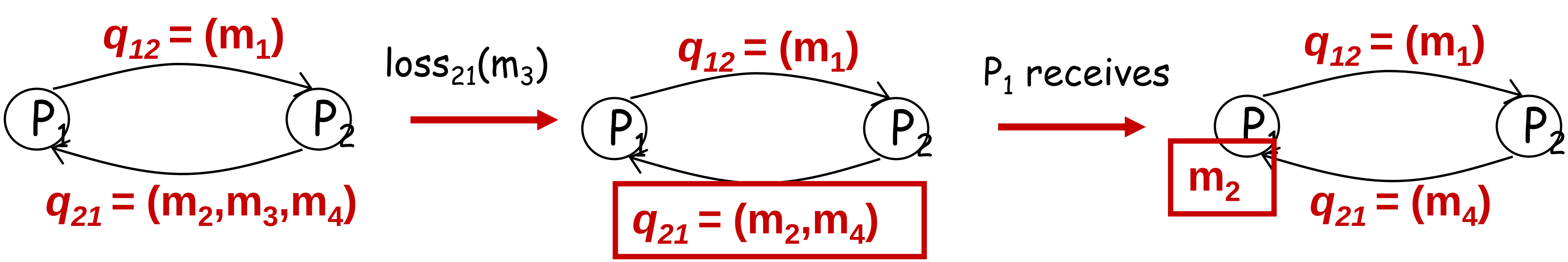
- Processors communicate by the use of shared communication registers
- The configuration will be denoted by
  $c = (s_1,s_2,\ldots,s_n,r_{1,2},r_{1,3},\ldots r_{i,j},\ldots r_{n,n-1})$ where
  si = State of P_i
  r_i= Content of communication register I
- Sometime we write c_i and sometime c[i] --- they are the same

The distributed System –
A Computation Step

In shared memory model …

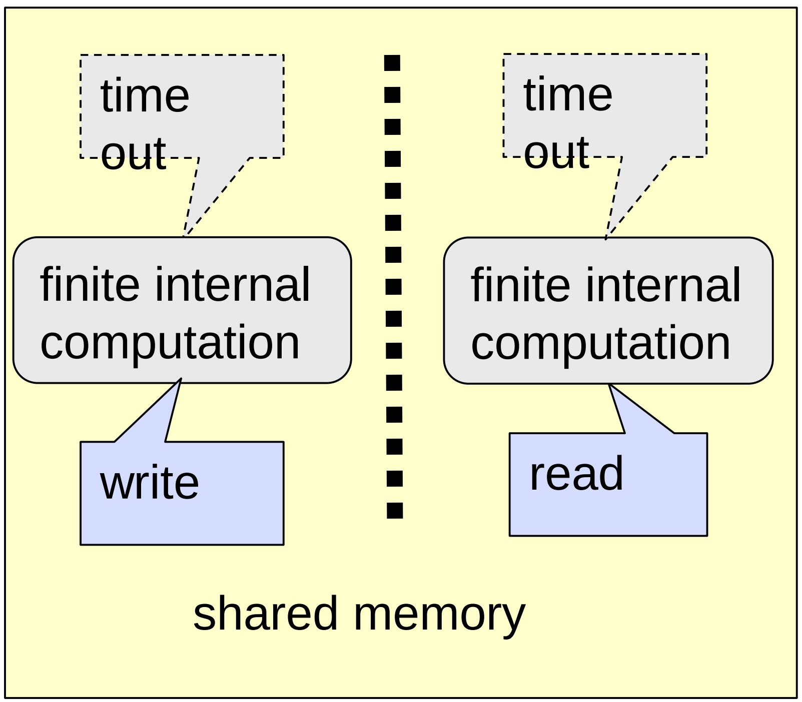


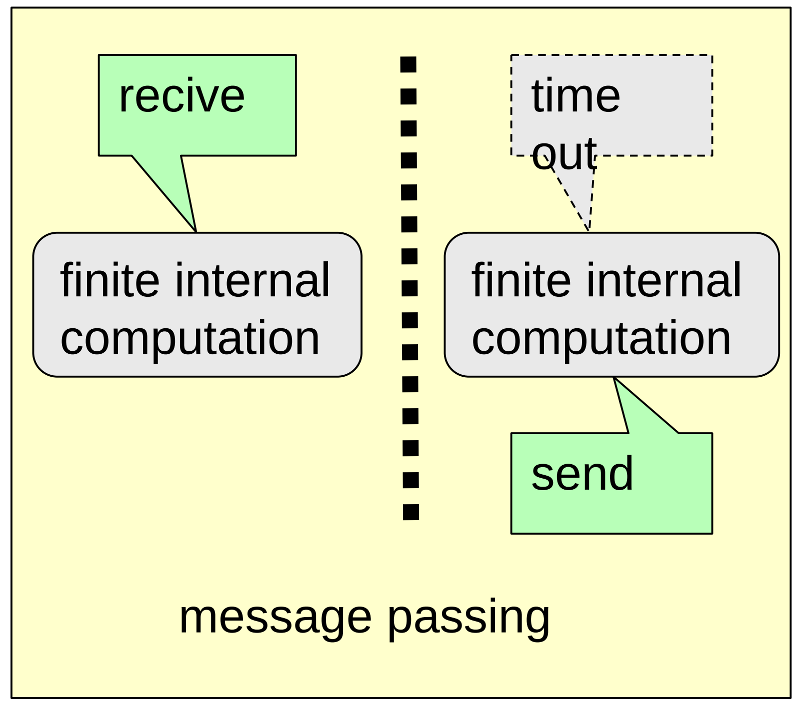
And in message passing model …



The distributed System – Communication Steps

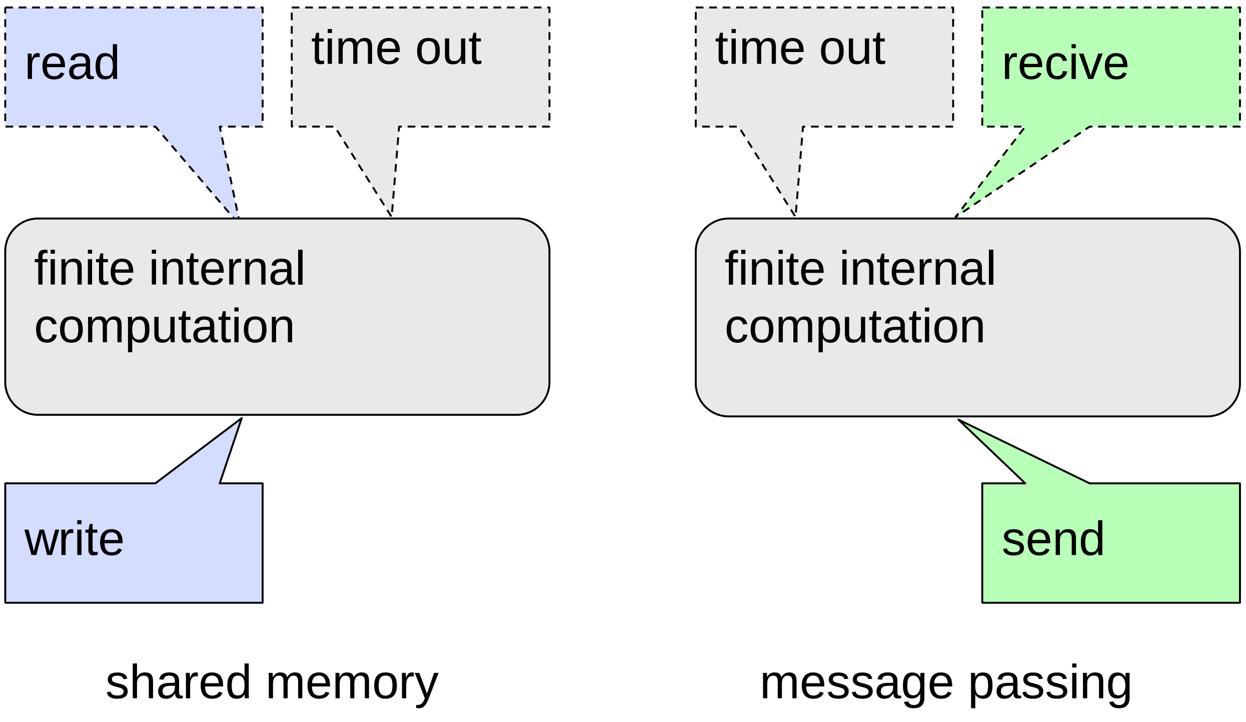
- The book considers computation steps with a single communication operation





The distributed System – Atomic Steps

- We often assume that steps are atomic and that they have the following form



The Interleaving model

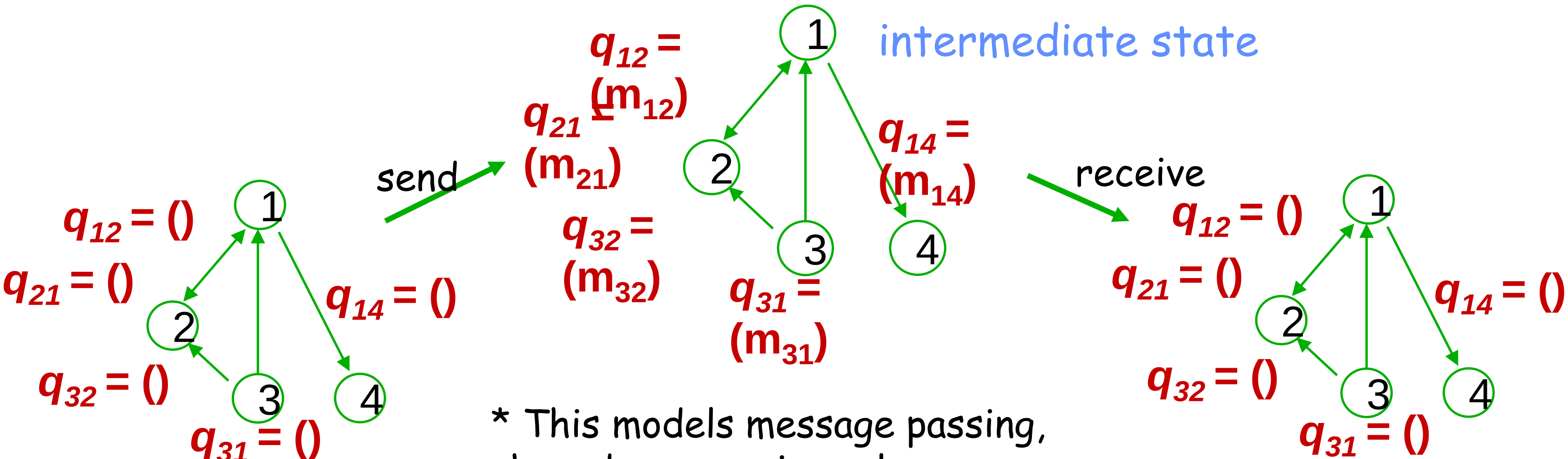
- Scheduling of events in a distributed system influences the transition made by the processors
- The interleaving model - at each given time only a single processor executes a computation step
- Every state transition of a processor is due to communication-step execution
- A step will be denoted by a
- $c_1 \xrightarrow{a} c_2$ denotes the fact that c_2 can be reached from c_1 by a single step a

The distributed System – more definitions

- Step a is applicable to configuration c iff $\exists\, c' : c \xrightarrow{a} c'$.
- An execution $E = (c_0, a_0, c_1, a_1, \ldots)$, an alternating sequence such that $c_{i-1} \xrightarrow{a} c_i$ $(i>1)$
- A fair execution - every step that is applicable infinitely often is executed infinitely often
- A communication channel is fair (in message passing) when the fact that a message is sent infinitely often implies that this message is it received infinitely often

Synchronous Distributed Systems

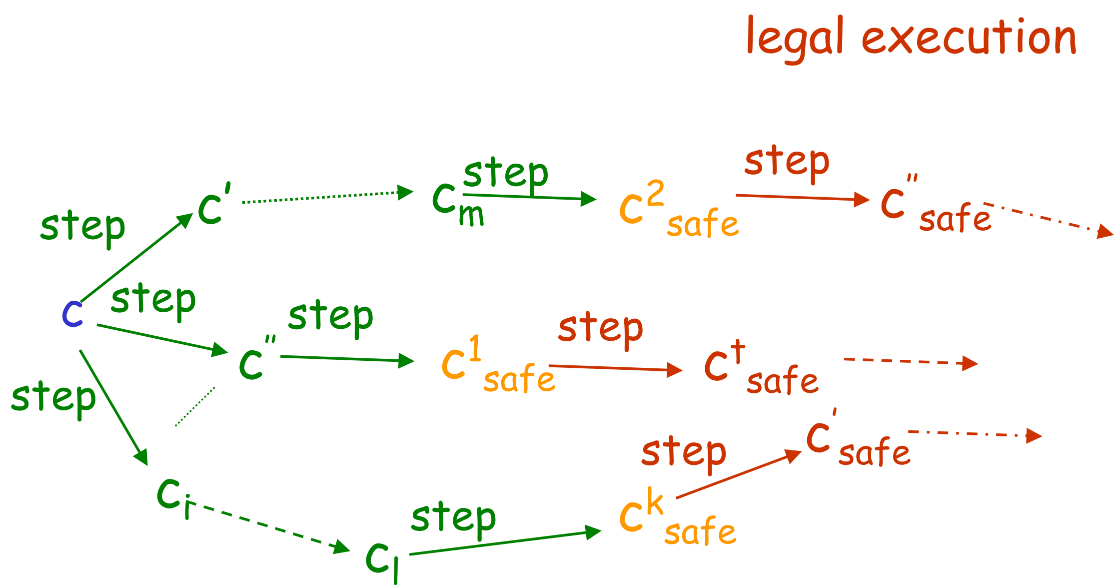
- A global clock pulse (pulse) triggers a simultaneous step of every processor in the system
- Fits multiprocessor systems in which the processors are located close to one another
- The execution of a synchronous system E = (c1,c2,...) is totally defined by c1, the first configuration in E



* This models message passing, shared memory is analogous with write -> read

Legal Behavior

- A desired legal behavior is a set of legal executions denoted LE



A self-stabilizing system can be started in any arbitrary configuration and will eventually exhibit a desired "legal" behavior

Self-stabilizing Systems

- Note that we define LE for a particular system and a particular task
- An execution of a self-stabilizing system has a suffix that appears in LE
- We say that configuration c is safe with regard to task LE and system, if every fair execution of the algorithm that starts from c belongs to LE
- An algorithm is self-stabilizing for a task LE if every fair execution of the algorithm reaches a safe configuration with relation to LE

Time complexity

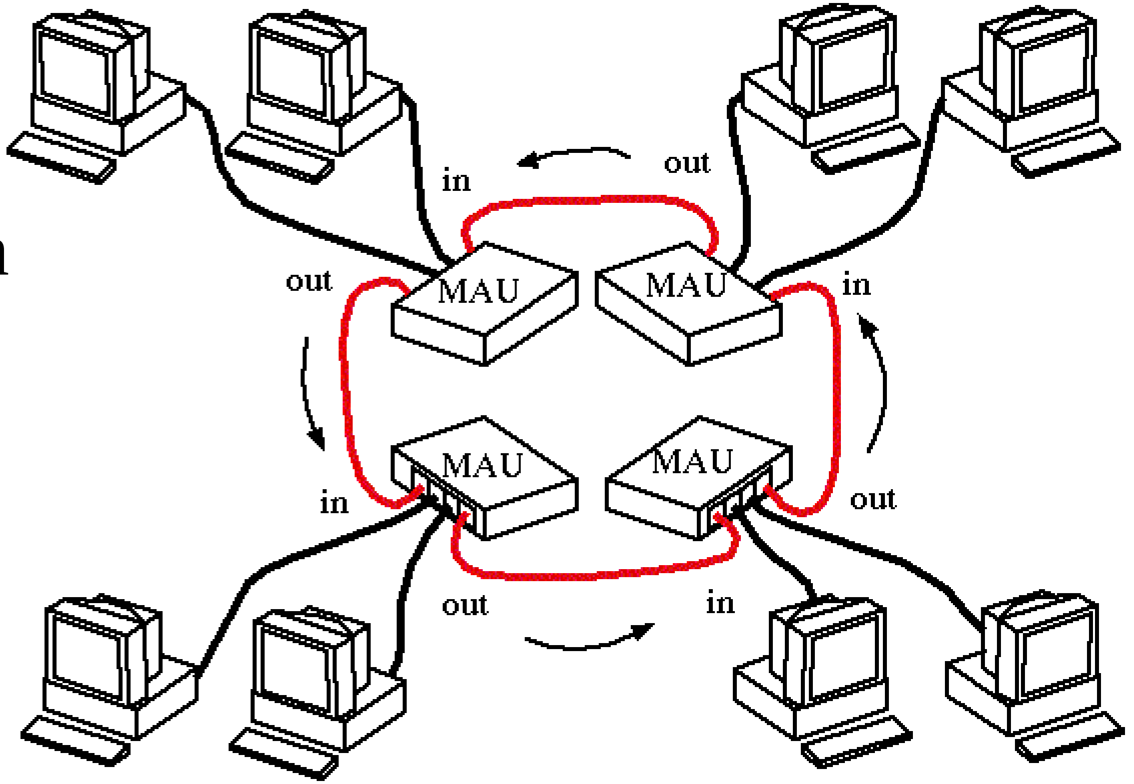
- The first asynchronous round (round) in an execution E is the shortest prefix E' of E such that each processor executes at least one step in E', where E=E'E''.
- The number of rounds = time complexity
- A Self-Stabilizing algorithm is usually a do forever loop
- The number of steps required to execute a single iteration of such a loop is $O(\Delta)$, where Δ is an upper bound on the number of neighbors of P_i
- Asynchronous cycle (cycle) the first cycle in an execution E is the shortest prefix E' of E such that each processor executes at least one complete iteration of it's do forever loop in E', E=E'E''.
- Note : each cycle spans $O(\Delta)$ rounds
- The time complexity of synchronous algorithm is the number of pulses in the execution

Space complexity

- The space complexity of an algorithm is the total number of (local and shared) memory bits used to implement the algorithm

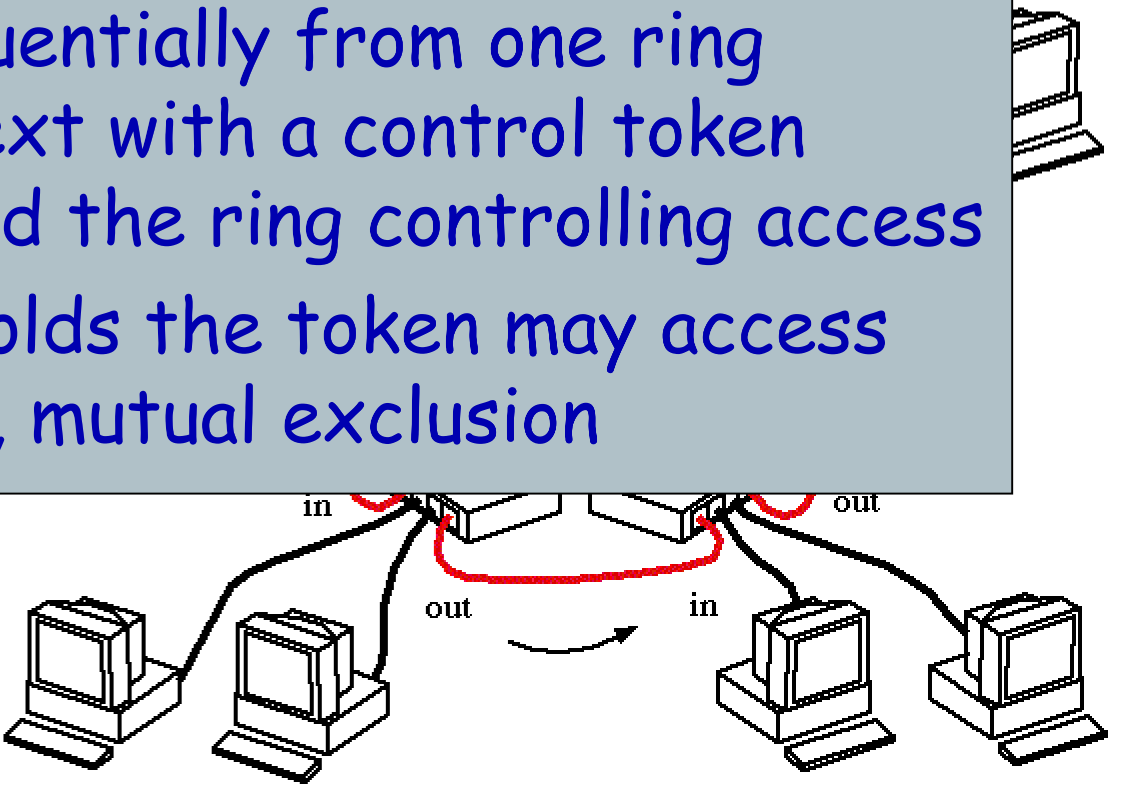
Dijkstra’s method

- The task of Mutual Exclusion has “other names” in message passing systems:
  - token ring
  - token passing
  - token circulation

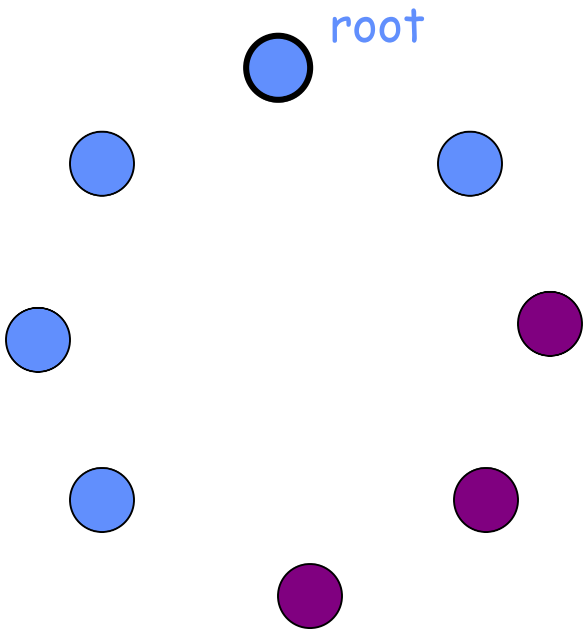


Dijkstra's method

- Stations on a token ring LAN are logically organized in a ring topology with data being transmitted sequentially from one ring station to the next with a control token circulating around the ring controlling access
- A station that holds the token may access the network, i.e., mutual exclusion



Mutual Exclusion



The root changes it's state if equal to it's neighbor

The rest – each processor copies it's neighbor's state if it is different

Legal Behavior of Mutual Exclusion

- The set LE_{ME} includes all the executions in which each processor may access the critical section infinitely often
  - Namely, let p_i be a processor in the network, E be an execution in LE_{ME} and E' be a suffix of E. Then, p_i may access the critical section infinitely many times
- Moreover, while p_i is accessing the critical section no other processor may concurrently access that critical section

Dijkstra's method

P_1: **do** forever

if $x_1 = x_n$ **then**

$x_1 := (x_1 + 1) \mathrm{mod} (n+1)$

$P_i (i \neq 1)$: **do** forever

if $x_i \neq x_{i-1}$ **then**

$x_i := x_{i-1}$

Dijkstra’s alg. is Self-Stabilizing

- A configuration of the system is a vector of n integer values (the processors in the system)
- The task ME:
  - exactly one processor can change its state in any configuration,
  - every processor can change its state in infinitely many configurations in every sequence in ME
- One of the safe configurations in ME and Dijkstra’s algorithm is a configuration in which all the x variables have the same value

Dijkstra's alg. is Self-Stabilizing

- A configuration in which all x variables are equal, is a safe configuration for ME (Lemma 2.2)
- For every configuration there exists at least one integer j such that for every i $x_i \neq j$ (Lemma 2.3)
- For every configuration c, in every fair execution that starts in c, P_1 changes the value of x_1 at least once in every n rounds (Lemma 2.4)
- For every possible configuration c, every fair execution that starts in c reaches a safe configuration with relation to ME within $O(n^2)$ rounds (Theorem 2.1)

Lemma 2.4

For every configuration c, in every fair execution that starts in c, processor p_1 changes the value of x_1 at least once in every n rounds

- Assume, in a way of a proof by contradiction, that there exists a configuration c and a fair execution that starts in c and in which p_1 does not change that value of x_1 during the first n rounds.
- Lets c_2 be the configuration that immediately follows the first time p_2 executes an atopic step during the first round.

Lemma 2.4

- Clearly, $x_1 = x_2$ in c_2 and in every configuration that follows c_2 in the next n-1 rounds. Let c_3 be the configuration that immediately follows the first time p_3 executes a computation step during the second round.
- It holds in c_3 that $x_1 = x_2 = x_3$. The same arguments repeats itself until we arrive at the configuration c_n, which is reached in the (n-1)-th round and in which $x_1 = x_2 = \ldots = x_n$.
- The n-th round, p_1 is activated and changes the value of x_1 , a contradiction. □

Theorem 2.1

For every possible configuration c, every fair execution that starts in c reaches a safe configuration with relation to ME within $O(n^2)$ rounds

- In accordance with Lemma 2.3 for every possible configuration c there exists at least one integer $0 \leq j \leq n$ such that, for every $1 \leq i \leq n$ $x_i \neq j$ in c.

Theorem 2.1

- In accordance with lemma 2.4, for every possible configuration *c*, in every fair execution that starts in *c*, the special processor p_1 changes the value of x_1 in every *n* rounds.
- Every time p_1 changes the value of x_1, p_1 increments the values of x_1 module $n+1$.
- Thus, it must hold that every possible value, and in particular the value *j*, is assigned to x_1 during any fair execution that starts in *c*.

Theorem 2.1

- Let c_j be the configuration that immediately follows the first assignment of j in x_1.
- Every processor p_i, $2 \leq i \leq n$ copies the value x_{i-1} to x_i.
- Thus, it holds for $1 \leq i \leq n$ that $x_i \neq j$ in every configuration that follows c and precedes c_j; it also holds that in c_j, that the only x variables to hold the value j is x_1.
- Processor p_1 does not change the value of x_1 until $x_n=j$.

Theorem 2.1

- The only possible sequence of changes of the values of the *x* variables to the value of *j* is:
  - p_2 changes x_2 to the values of x_1 (which is *j*), then p_3 changes the value of x_3 to *j* and so on until p_n changes the value of x_n to *j*.
- Only at this stage is p_1 able to change value again (following c_j).
- Let c_n be the configuration reached following the assignment of $x_n := j$. c_n is a safe configuration.

Theorem 2.1

- In accordance with Lemma 2.4, p_1 must assign j to x_1 in every n^2 rounds.
- Thus a safe configuration must be reached in n^2+n rounds. □

Summary

- Revised our system settings
  - Added message passing
  - Added asynchrony
- Presented solutions and proved for mutual exclusion
- And also mutual exclusion for general communication graphs via fair composition

Review Questions

1. Rewrite the proof of the convergence and closure of the Leader Election program in the <u>message passing</u> model
2. Show that in a synchronous uniform ring, there is no deterministic self-stabilizing method for token circulating
   – Hint: The answer is similar to the leader election impossibility lemma