Computer Networks

EDA387/DIT663

Fault-tolerant Algorithms for Computer Networks

Introduction and Leader Election (Ch.2)

http://www.ted.com/talks/danny_hillis_the_internet_could_crash_we_need_a_plan_b.html

Self-* Methods

- Computer systems can be complex due to numerous factors including scale, decentralization, heterogeneity, mobility, dynamism, bugs and failures
- Deploying, operating and maintaining such systems can be not only very difficult, but also very costly
- A flurry of recent activity has been directed at this problem whereby future computer systems are envisioned to be self-configuring, self-organizing, self-managing and self-repairing, aka, self-* properties

Self-* Methods

- Fault-tolerant computer systems that are self-stabilizing can recover after the occurrence of transient faults
  - which can cause an arbitrary corruption of the system state (so long as the program's code is still intact)
- This design criteria liberate the application designer from dealing with low-level complications, and provide an important level of abstraction
- Consequently, the application design can easily focus on its task - and knowledge-driven aspects

Goal

- We would like to understand how to design self-stabilizing network protocols
- At the end of these 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

- Task definition
- Solutions for Internet-like networks
  - How to describe such solutions?
  - How to argue about their correctness?
- Other type of networks
  - When can we not solve a task?

What is a System of Computer Networks?

- A network system consists of multiple autonomous computers that communicate through local networks
  - Communication networks
  - Multiprocessor computers
  - Multitasking single processor

- The computers interact with each other in order to achieve a common goal
  - Namely, the program *task*

- A Distributed System is modeled by a set of *n* state machines called processors that communicate with each other

The Network Model

Processor can be a computer, a router, CPU, a thread, process, etc.

Denote:

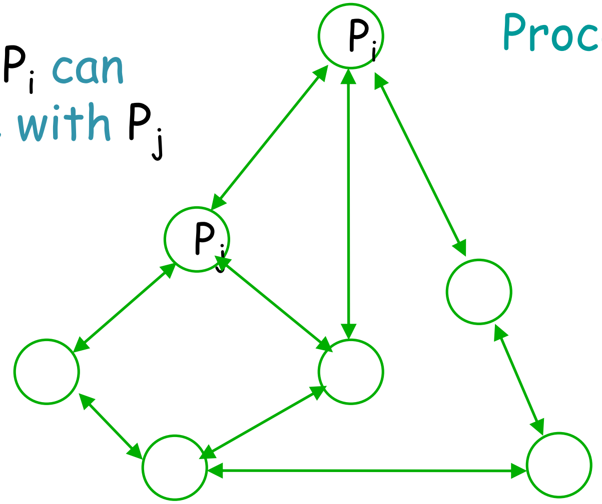
- p_i - the i^{th} processor in the set of the processors P
- neighbor of P_i - a processor that can communicate with it directly

Network Representation

How to represent the network?

Link P_i<->P_j = P_i can communicate with P_j

Node i = Processor i

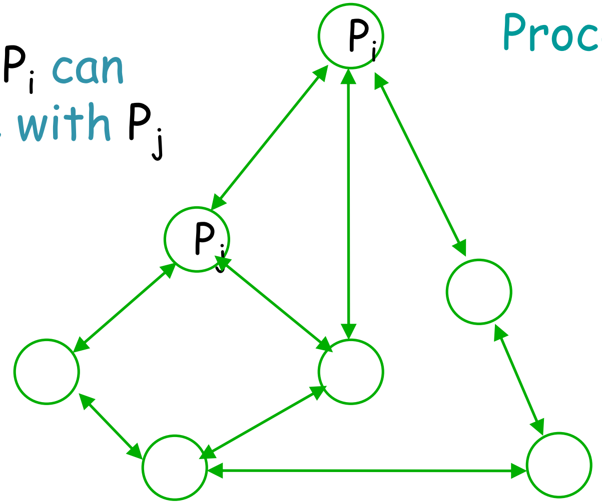


Network Representation

How to represent the network?

Link P_i<->P_j = P_i can communicate with P_j

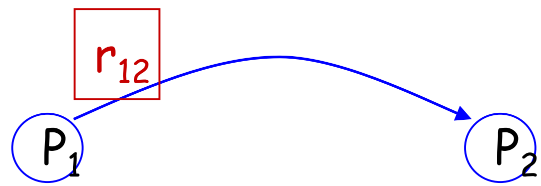
Node i = Processor i



Ways of communication

- message passing - fits communication networks and all the rest
- shared memory - fits geographically close systems (our focus today)

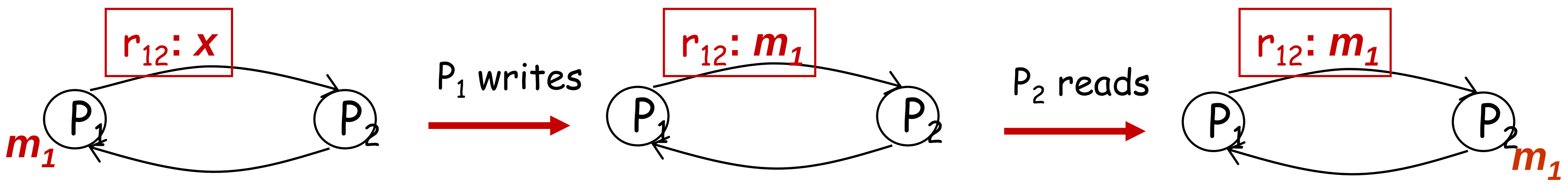
Shared Memory



- 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
  s_i = State of p_i
  r_i= Content of communication register I
- Sometime we write c_i and sometime c[i] --- they are the same

Computation Steps

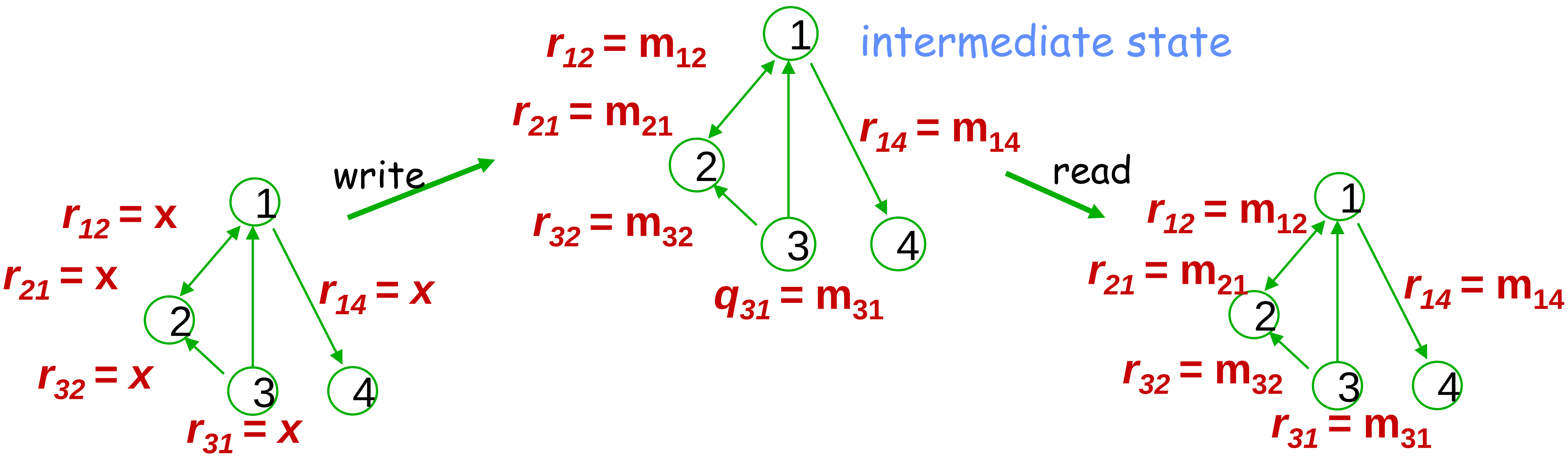
In shared memory model ...



Synchronous Computation

- Every state transition of a process 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
- Step a is applicable to configuration c iff $\exists\, c' : c \xrightarrow{a} c'$.
- Execution E = $\langle c_1, a_1, c_2, a_2, \ldots \rangle$: an alternating sequence such that $c_{i-1} \xrightarrow{a} c_i$ $(i>1)$
- 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 network is totally defined by c_1, the first configuration in E
  - Therefore, we can write E = $\langle c_1, c_2, \ldots \rangle$

Synchronous Computation



Leader Election

- Need to designate a single processor as the organizer of some (other) task among several processors
  - Simplifies many tasks by providing a single point of decision
- Leader election: (eventually) in every configuration, there is exactly one processor in the network for which the property *Leader* holds
  - Before the task is begun, all network nodes are unaware which node will serve as the "Leader," or coordinator, of the task
  - After a leader election method has been to run, however, each node throughout the network recognizes a particular, unique node as the task leader

Leader Election

- The network nodes communicate among themselves in order to decide which of them will have the “Leader” property
- For that, the processors need some method in order to break the symmetry among them
- How can they do that?

Leader Election

- The network nodes communicate among themselves in order to decide which of them will have the “Leader” property
- For that, the processors need some method in order to break the symmetry among them
- How can they do that?
- Hint: if each node has unique identity, such as IP address, then the nodes can compare their identities, and decide that the processor with the highest identity is the leader

Leader Election

1 **do** forever
2   **write** *id* **to** r_i
3   **for** m := 1 to n **do** lr_m := **read**(r_m)
4   Leader := (*id* == *maximum* $\{lr_m.\mathrm{id} \mid 1 \le m \le n\}$)
5   (* **if** *Leader* == True **then** ***act _like_a_leader() *)***

- Every processor:
  - starts by writing its unique id to its register (line 2)
  - reads the ids of its neighbors (line 3)
  - decides on the “Leader” property (line 4)

Arguing Correctness

- We need to convincingly demonstrate that some system properties and statements are necessarily true.
- The proof must demonstrate that a statement is true in all cases, without a single exception.
- The statement that is proved is often called a theorem, a lemma, or a claim.
- For example:

  Lemma (**maximum**): Let A be a set of (unique) integers, which is totally ordered by $\leq$. There is a single (unique) maximal value, $maximum(A, \leq)$.

Proof Statement

- First list all of the proof assumptions before claiming that some properties hold within a finite time
- **Leader Election:**
  - Assumptions:
    - All processors execute the same program
    - All processors have unique ids
    - The system is synchronous
      - All processors take their steps simultaneously
      - All processors start executing the program simultaneously in line 01
    - The network topology is of a fully connected graph
  - Within one complete iteration of the leader election program, exactly one processor has the property "Leader"

Convergence of Leader Election

Lemma (start of lemma statement)

- Suppose that in a synchronous network, all processors have unique ids and that they all execute the Leader Election program.
  - Namely, all processors take their steps simultaneously and start executing the program simultaneously.

Leader Election Convergence

- Moreover, suppose that the network topology is of a fully connected graph.
- Within one complete iteration of the leader election program, exactly one processor has the property “Leader”.
- Let $E=(c_0, c_1,\dots)$ be the system execution of the leader election program.
- Let c_{safe} be the first configuration after one complete iteration of the program, in which all processors execute lines 1 to 5.

Convergence of Leader Election

- In c_{safe}, exactly one processor has the property “Leader” , which is the processor with the maximal id, p_{max}.

(end of lemma statement)

Remark: We later remove the assumption that all processors start in line 01

Leader Election: Correctness

Proof: Suppose that all programs start from executing line 1. Since all processors start executing their programs simultaneously, we are going to show that c_{safe} is c_{n+1}, because n+1 steps guarantee one complete iteration.

By line 2 of the pseudo code, we have that in configuration c_1, it holds that for any processor, p_i: $r_i = id$.

By line 3, we have that in configuration $c_{(1+n)}$, it holds that for any two processors, p_i and p_j, the sets $A_i=\{lr_{mi}.\mathrm{id} \mid 1 \leq m \leq \mathrm{n}\}$ and A_j are identical, i.e., $\mathrm{A_i} = \mathrm{A_j}$.

Leader Election: Correctness

By the assumption that all processors have unique ids and the maximum lemma, we have that in configuration $c_{(n+1)}$ exactly one processor, p_{max}, has the property of being a leader. I.e., for any p_i in $P \backslash \{p_{max}\}$: $\text{Leader}_i = \text{false}$ and $\text{Leader}_{max} = \text{true}$, where P is the set of all processors in the system.

□

(The symbol □ is used to mark the end of a proof.)

Leader Election: Correctness

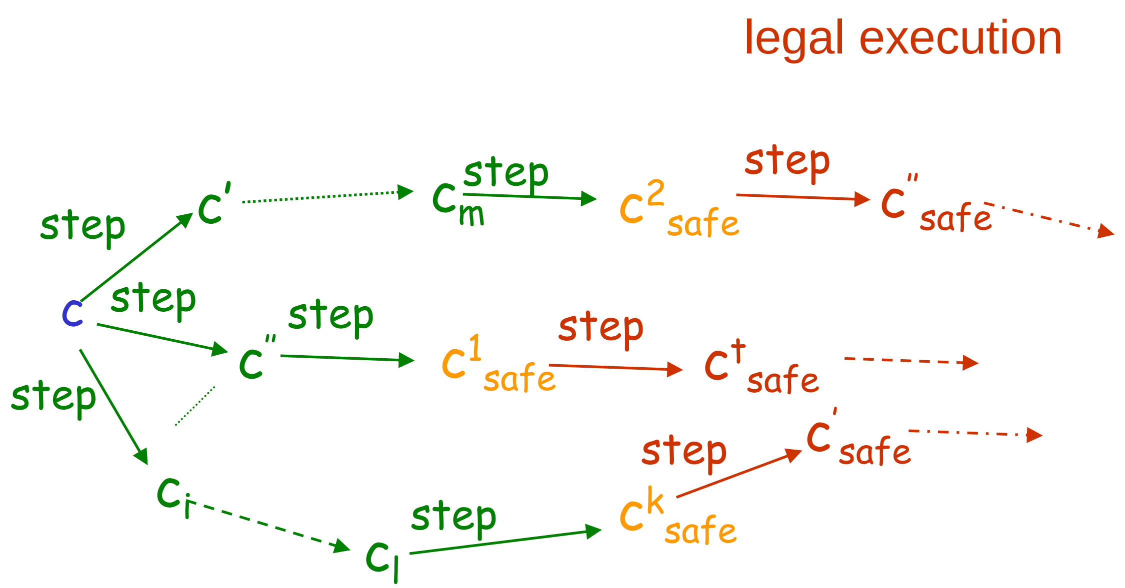
- Suppose there is no joint start in line 01?
  - Within (n+1) steps we are guaranteed that ***<u>all</u>*** processors, p_i, write their own id to their own register, r_i.
    - Once it happens, r_i's value does not change.
  - Within (n+1) additional steps we are guaranteed that all processors, p_i, read the registers, r_j, of all of their neighbors, p_j.
    - Once it happens, lr_i's value does not change.
- The proof is finished by the same arguments on A_i

Leader Election: Correctness

- We have just proved that the leader election program convergence to a configuration in which it is safe to assume that we have exactly one leader.
- Actually, we can, and should, prove more.
- Lemma (**Closure of Leader Election**) Assume the same assumption as in the Convergence of Leader Election Lemma. Moreover, suppose that in E's starting configuration, c_0, the processor with the maximal id, p_{max}. is the only processor that has the property of being "Leader". In every configuration in *E*, p_{max} is the only processor that has the property "Leader".

Legal Behavior

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



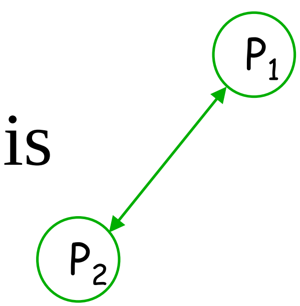
A self-stabilizing system can be started in any arbitrary configuration and will eventually exhibit a desired “legal” behavior

Legal Behavior of Leader Election

- LE_{leader}: In every configuration there is exactly one processor that has the property “Leader”
- Can we guaranteed that starting from arbitrary configuration, we reach a safe configuration with respect to LE_{leader}?
- Is there a deterministic method for leader election in uniform systems in which all processors are identical?
  - all processors are running the same program, they have no unique id’s, etc.

Legal Behavior of Leader Election

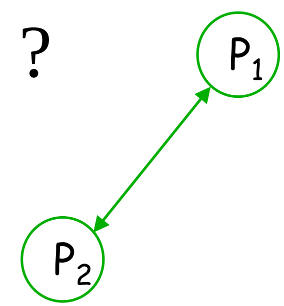
- Suppose there is such a method, then how can we deal with an execution E in a network of two processors and a starting configuration, $c[0] = \langle s_1, s_2, r_1, r_2 \rangle$, in which both p_1 and p_2 are in the same state, where $s_1 = s_2$ and $r_1 = r_2$
- Lets say that within l steps we reach a configuration in which the leader election task is achieved
  - one processor is a leader and the other one is not
  - $s_1 = s_2$ and $r_1 = r_2$ do not hold



Legal Behavior of Leader Election

- Let $c[l_{last\text{-}same}]$ and $c[l_{first\text{-}different}]$ be the last, and respectively, the first be two consecutive configurations in E, such that $s_1=s_2$ and $r_1=r_2$ do, and respectively, do not hold
  - Consecutive configurations: $l_{first\text{-}different} = l_{last\text{-}same}+1$
- The question is how a deterministic method can take a step that leads from $c[l_{last\text{-}same}]$ to $c[l_{first\text{-}different}]$?
  - Same program
  - Same state
  - Same number of neighbors
  - No ids
  - Same program counter
  - …

It cannot!!!



Leader Election: Impossibility

Lemma (**Impossibility of Leader Election in Anonymous Networks**) Let us consider an anonymous, synchronous network that has a topology of a fully connected graph. No deterministic leader election method exists for this network

Proof: Lets assume, in a way of contradiction, that a deterministic leader election method, DLE, *does* exist and E is an execution of DLE that starts in configuration, $c[0]=\langle s_1,s_2,\ldots s_i, s_j,\ldots,s_n,r_1,r_2,\ldots r_i, r_j,\ldots r_n\rangle$, where $s_i = s_j$ and $r_i = r_j$

Leader Election: Impossibility

DLE solves the leader election task by reaching a configuration in which one processor has a state that is different than all other processors, which is the leader

Therefore, in E there are two consecutive configurations $c[l_{last\text{-}same}]$ and $c[l_{first\text{-}different}]$ that are the last, and respectively, the first configurations in E for which $s_i = s_j$ and $r_i = r_j$ do, and respectively, do not hold, where $l_{first\text{-}different} = l_{last\text{-}same}+1$

Leader Election: Impossibility

Therefore, DLE causes at least one processor to take a step from $c[l_{last\text{-}same}]$ to $c[l_{first\text{-}different}]$ that is different than the step of other processors between these two configurations

This is a contradiction, because by definition any deterministic algorithm implies that if $s_i = s_j$ and $r_i = r_j$ holds for all p_i and p_j in $c[l]=<\mathrm{s}_1,\mathrm{s}_2,\ldots \mathrm{s}_\mathrm{i}, \mathrm{s}_\mathrm{j},\ldots,\mathrm{s}_\mathrm{n},\mathrm{r}_1,\mathrm{r}_2,\ldots \mathrm{r}_\mathrm{i}, \mathrm{r}_\mathrm{j},\ldots \mathrm{r}_\mathrm{n}>$, then both p_i and p_j take identical steps

□

Summary

- Presented the model of share memory
- Presented the self-stabilization design criteria
- Defined the leader election task
- Presented a solution and proved its correctness

Review Questions

1. Write the proof of the Closure for the Leader Election task