# CSc72010

Leader Election

Reading: Chapters 14 (Asynchronous Network Model), 15.1 (Leader Election in a Ring)

## Importance of leader election algorithms

The basic solution to cycles in an Ethernet LAN is for the switches to run the spanning tree protocol (STP), which determines a spanning tree for the LAN. The switches then forwards frames only through ports that are part of a spanning tree.

Reliable servers are implemented by having multiple reliable servers; one of them may be chosen as primary (or primary for some subset of the requests, in the case of load-balancing servers).

### *Introduction*

Model:

Network of identical processes
Algorithm to pick a leader
Motivation: see above; token ring; commit coordinator; resource allocator

Requirement: Sometime after we start the algorithm, exactly one process outputs "leader."

Folk theorem: cannot use identical processes
Proof: By induction on number of rounds
Idea: If all processes are in the same state at round r, they must be in the same state at round r+1 (otherwise, they are not identical).
Therefore, if any process declares itself leader, they all do

Can we get away with just an ID distinguishing the processes? Each process knows its own UID and can distinguish its neighbors.

### *Leader election in a ring*

Special case: Network is ring – still has some of the key difficulties

```
We draw a ring 1->2->3->4
               ^        |
               |-----------
```

but the processes don't know the numbers; all they can do is distinguish clockwise and counter-clockwise neighbors. Note that there's also an assumption that the neighbors are fixed. This may not be true in a wireless LAN, especially an *ad hoc* network.

**Lelann-Chan-Roberts (LCR)**

Assume unidirectional ring (note that for learning bridges, we must have bidirectional).
Processes don't know network size

Processes can tell if two UID's are the same or different; no arithmetic allowed.

Informally: each process sends its own identifier to its clockwise neighbor. A process forwards a received UID if it is larger than its own; drops it if smaller; declares itself leader if it receives its own UID.

**Correctness**
Eventually, one (and only one) node outputs "leader"

Let n be the size of the ring (number of processes)
Define $i_{max}$ to be the number of the process with the maximum UID
Define $u_{max}$ to be the maximum UID

## ***Leader***

Note: old version of TIOA!! Must be updated to run in tempo

```
type Status = enumeration of leader, unknown

automaton Process (pos : Int, prev : Int, next : Int,  rank: Int)
  signature
    input RECEIVE(m: Int , const prev, const pos : Int)
    output SEND (m: Int , const pos , const next : Int)
    output leader(const rank)

  states
    pending : Int := rank,        % largest ID seen so far
    status : Status := unknown,   % may become leader or non-leader
    ready : Bool := true          % ready to send a message if true

  transitions
      % to illustrate "where" clause; not a great way to use it
     input RECEIVE(m, j, i) where m > pending
       eff pending := m; ready := true

     input RECEIVE(m, j, i) where m < pending

     input RECEIVE(m, j, i) where m = rank
       eff status := leader

     output SEND (m, i, j)
       pre status ~= leader /\ m = pending /\ ready
       eff ready := false

     output leader(rank)
       pre status = leader
```

Representation of state: <4, T, unknown>, <7,F,leader>
That is, <*pending*, *ready*, *status*>

## ***Channel***

Reliable channels

```
automaton Channel(i:Int , j:Int)
  signature
    input SEND (m: Int , const i : Int, const j : Int)
    output RECEIVE(m : Int, const i : Int ,const j : Int)

  states
    queue : Seq[Int] := {}

  transitions
    input SEND (m, i, j)
      eff queue := queue |- m

    output RECEIVE(m,i,j)
      pre head(queue) = m
      eff queue := tail(queue)
```

Representaton of state: {}, {4}, {4,7}
That is, *queue*

## Composition

```
automaton LCR
  components

   P1: Process(1, 4, 2, 7);
   P2: Process(2, 1, 3, 3);
   P3: Process(3, 2, 4, 2);
   P4: Process(4, 3, 1, 5);
   C1: Channel(1, 2);
   C2: Channel(2, 3);
   C3: Channel(3, 4);
   C4: Channel(4, 1)
```

## Schedule

```
schedule
  do
    while(true) do
      fire output P1.SEND(P1.pending,1,2);
      fire output P2.SEND(P2.pending,2,3);
      fire output P3.SEND(P3.pending,3,4);
      fire output P4.SEND(P4.pending,4,1);

      fire output C1.RECEIVE(C1.toSend,1,2);
      fire output C2.RECEIVE(C2.toSend,2,3);
      fire output C3.RECEIVE(C3.toSend,3,4);
      fire output C4.RECEIVE(C4.toSend,4,1)


    od
  od
```

## A sample execution

[<7,T,unknown>, {}, <3,T,unknown>, {}, <2,T,unknown>,{},<5,T,unknown>,{}],
SEND(2,3,4),
[<7,T,unknown>,{},<3,T,unknown>,{},<2,F,unknown>,{2},<5,T,unknown>,{}],
SEND(3,2,3),
[<7,T,unknown>,{},<3,F,unknown>,{3},<2,F,unknown>,{2},<5,T,unknown>,{}],

RECEIVE(3,2,3),
[<7,T,unknown>,{},<3,F,unknown>,{},<3,T,unknown>,{2},<5,T,unknown>,{}],
SEND(7,1,2),
[<7,F,unknown>,{7},<3,F,unknown>,{},<3,T,unknown>,{2},<5,T,,unknown>,{}],
SEND(3,3,4),
[<7,F,unknown>,{7},<3,F,unknown>,{},<3,F,unknown>,{2,3},<5,T,unknown>,{}],
RECEIVE(7,1,2),
[<7,F,unknown>,{},<7,T,unknown>,{},<3,F,unknown>,{2,3},<5,T,unknown>,{}],
SEND(7,2,3),
[<7,F,unknown>,{},<7,F,unknown>,{7},<3,F,unknown>,{2,3},<5,T,unknown>,{}],
RECEIVE(7,2,3),
[<7,F,unknown>,{},<7,F,unknown>,{},<7,T,unknown>,{2,3},<5,T,unknown>,{}],
SEND(7,3,4),
[<7,F,unknown>,{},<7,F,unknown>,{},<7,F,unknown>,{2,3,7},<5,T,unknown>,{}

Prove:
1) $i_{max}$ outputs leader by the end of round n
2) No other process outputs leader

Assume nodes are numbered 0,…,n-1

We use invariants to prove safety properties – such as no process except the leader outputs leader. These are properties that are true in all reachable states.
Standard proof technique:
1.Define an invariant
2.Prove by induction. The induction step examines case-by-case what the transition function does.

Invariant (Assertion 15.1.1 from book, page 478)

## *Variants*

**Processes stop when the leader has been output:**

Leader must circulate new message

**Non-leader announcements**

When a process receives a larger UID

Number of messages is n^2 and time is roughly the size of the ring; we will learn how to prove this later, when we learn how to prove that a leader is eventually elected.

**Reduced complexity**

Can we reduce messages below n^2?
Yes: Hirschberg-Sinclair

Assumptions

Bidirectional

Don't know ring size
Comparisons only for UID's

Informally: Send the UID both directions to successively greater distances (double the distance on each pass.
Outbound: Pass on the UID if it's larger than your own, swallow it otherwise.
Inbound: Pass everything on.
If you get your own UID inbound, proceed to the next round
Details in Section 15.1.2

Number of messages is O(n log n).

## Composition

### *Properties*

We would like composition to behave appropriately, i.e., we can recover the original automata from the composition.

**Definition**. For an execution _ and an automaton $A_i$, define the restriction of _ to $A_i$ as _$|A_i$ = the same execution with all actions not in $A_i$ (and the subsequent states) removed, and with states projected onto states of $A_i$.

Restriction onto Process(1,4,2,7):
For a trace _ and an automaton $A_i$, define the restriction of _ to $A_i$ as _$|A_i$ = the same trace with all actions not in $A_i$ removed.

Example: see red, above, for restriction to P(3,2,4,2). Actions of P(3,2,4,2) are:
RECEIVE(m, 2, 3)
SEND(m, 3, 4)
leader(2)

**Theorem 8.1**. Projection.
1. If $\alpha \in$ execs(A) then $\alpha|A_i \in$ execs($A_i$) for every i.
2. If $\alpha \in$ traces(A) then $\alpha|A_i \in$ traces($A_i$) for every i.

**Theorem 8.2**. Pasting.
1. If $\alpha_i \in$ execs($A_i$) for all i, $\beta$ is a sequence of actions in ext(A) such that $\beta|A_i$ = trace($\alpha_i$) for all i, then there is an execution $\alpha$ of A such that $\beta$ = trace($\alpha$) and $\alpha_i = \alpha|A_i$ for all i.
2. If $\alpha$ is a sequence of actions in ext(A) and $\alpha|A_i \in$ traces($A_i$) for all i then $\alpha \in$ traces(A).

Consider _ =SEND(7,1,2), SEND(3,2,3), SEND(2,3,4), SEND(5,4,1), RECEIVE(7,1,2), RECEIVE(3,2,3), RECEIVE(2,3,4), RECEIVE(5,4,1), SEND(7,2,3), SEND(3,3,4), RECEIVE(7,2,3), RECEIVE(3,3,4), SEND(7,3,4), RECEIVE(7,3,4), SEND(7,4,1), RECEIVE(7,4,1)

## Implementation

A crucial definiton is the following: We say that A *implements* A' if every trace of A is a trace of A'. Since TIOA serve as specifications, we need a criterion for saying when an implementation conforms to the specification. This is the criterion.

Consider the reliable FIFO channel. We can show that it implements a reliable channel that doesn't guarantee in-order delivery, which in turn implements a channel that delivers every message at least once, but possibly out of order, which in turn implements a best-effort channel. (This should be intuitively obvious.)

So, if an algorithm works with an unreliable channel, for example, we don't have to prove it again for a reliable channel. This relies on the following substitutivity result:

Use above theorems to prove:
**Theorem 3.** Substitutivity

Suppose A and A' have the same external signature and traces(A) $\subseteq$ traces(A'). Similarly for B and B'. Then traces(AxB) $\subseteq$ traces(A'xB').

Proof: Let $\alpha' \in$ traces(A$\times$B). Then by Theorem 8.1 $\alpha'|A \in$ traces(A) and $\alpha'|B \in$ traces(B) and so $\alpha'|A' \in$ traces(A') and $\alpha'|B' \in$ traces(B'). Then (using part 2 of Theorem 8.3) implies that $\alpha' \in$ traces(A'$\times$B').

## Fairness

For liveness, we need to know that each task keeps getting turns to do steps (even if no steps are enabled).

### *Definition*

Formally, an execution fragment $\alpha$ is defined to be fair if for all C $\in$ tasks(A), one of the following holds:

1) $\alpha$ is finite and no action of C is enabled in the final state of $\alpha$.
2) $\alpha$ is infinite and contains infinitely many steps with actions in C.
3) $\alpha$ is infinite and contains infinitely many states in which C is not enabled.

fairexecs(A) is the set of fair executions
fairtraces(A) is the set of fair traces (i.e., traces of fair executions)

### *Examples*

#### Universal reliable FIFO channel

`{}, send(i,j,a), {}|-a, receive(i,j,a), {}, send(i,j,b), {b}, receive(i,j,b), {}` is fair

`{}, send(i,j,a), {}|-a, receive(i,j,a), {}, send(i,j,b), {b}` is not fair, because receive is enabled

`{}, send(i,j,a), {}|-a, send(i,j,a), {}|-a|-a, send(i,j,a), {}|-a|-a|-a, ..., send(i,j,a), {}|-a|-a...|-a, ...`
is unfair because one of j's tasks (the set of receive actions) never gets to execute.

What are the fair sequences?

Finite: must end in empty queue
Infinite: Every message sent is received (note this is part of the definition of a reliable channel – without the fairness condition, we can't prove reliability, because the channel could just stop otherwise.)

## Clock

Clock, p. 213 (actually, this is a translation to the language)

```
automaton Clock

signature
  input request
  internal tick
  output clock(t:Int)

states
  counter:Int := 0,
  flag:Bool := false

transitions
input request
eff
  flag := true

output clock(t:Int)
pre
  flag = true /\ counter = t
eff
  flag := false

internal tick
pre
  true
eff
  counter := counter+1

tasks
  { tick };
  { clock(t:Int) }
```

What are the fair executions?

No finite sequence of ticks because tick is always enabled
Every request for time must be answered in the execution.

## *Properties*

**Theorem 8.4**: Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \cup_{i \in I} A_i$.

1. If $\alpha \in$ fairexecs(A), then $\alpha | A_i \in$ fairexecs($A_i$), for every $i \in I$.
2. If $\alpha \in$ fairtraces(A), then $\alpha | A_i \in$ fairtraces($A_i$), for every $i \in I$.

The following are analogous to pasting for arbitrary executions:

**Theorem 8.5**: Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \cup_{i \in I} A_i$. Suppose $\alpha_i$ is a fair execution of $A_i$ for every $i \in I$, and suppose $\beta$ is a sequence of actions in ext(A) such that $\beta | A_i$ = trace($\alpha_i$) for every $i \in I$. Then there is a fair execution $\alpha$ of A such that $\beta$ = trace($\alpha$) and $\alpha_i = \beta | A_i$ for every $i \in I$.

Theorem 8.5 means that we can paste together fair executions of compatible automata to get fair executions of the composition.

**Theorem 8.6**: Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \cup_{i \in I} A_i$.
Suppose $\beta$ is a sequence of actions in ext(A). If $\beta | A_i \in$ fairtraces($A_i$) for every $i \in I$, then $\beta \in$ fairtraces(A).

Theorem 8.6 means that we can paste together fair traces of component automata to get a fair trace of the composition.

**Theorem 8.7**: Every finite execution (or finite trace) can be extended to a fair execution (or fair trace). [See technical details in book.]
.
Theorem 8.7 says that we can extend finite executions/traces to fair executions.