Dependable and Distributed Systems

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 5 - Leader Election

Attendance Code: 88554452

Distributed System Models

Over the next few lectures we will study various problem and their solutions in the context of a synchronous system model

For example, we will study consensus, one of the most important problem in distributed systems

We will also look at variants of consensus and associated techniques

How Will We Study Distributed Systems?

Each problem we study in distributed systems will have an enumerated set of requirements that any solution we design must satisfy

These requirements form our **problem specification**

We will study how to show whether a problem solution solves the problem

Recall - System Model Role

We have a **system model**

- How we view our system - synchronous, partially synchronous, asynchronous, etc.

Define a **fault model**

- What are the potential problems that we must consider

System has a **specification**

- Essentially an oracle for system behaviour (capturing the specification we have just mentioned)

Synchronous Network Model

Synchronous Network Model

Extent of message delays and execution speed are known and bounded

Synchronous network model

- Can be represented as directed graph $G = (V, E)$ with a fixed message alphabet M
- Each node in the graph represents a process with a unique identifier
- We have $|V| = n$, though the edges and number of nodes may or may not be know to each node
- Out-degree and in-degree of a node are the number of edges leaving and coming into the node respectively
- Distance between two nodes is the length of the shortest path between those two nodes
- Diameter of a graph is the maximum distance between any two nodes in the graph

Synchronous Network Model

For each node $i \in V$ we have a process, which formally consists of:

- $states_i$ - Set of states (not necessarily finite)
- $start_i$ - Non-empty subset of states, know as the start states
- $msgs_i$ - Message generation function, mapping $states_i \times out\text{-}nbrs_i$ to elements of $M \cup \{null\}$
- $trans_i$ - State transition function, mapping $states_i$ and vectors ($in\text{-}nbrs_i$ indexed) of elements of $M \cup \{null\}$ to $states_i$

Each process has a set of states, a subset of which are considered to be start states

Message generation function specifies, for each state and outgoing neighbour, the message (if any) that process i sends that neighbour, starting from the given state

State-transition function specifies, for each state and collection of messages from all incoming neighbours, the new state to which process i should move

Synchronous Network Model

For each $(i, j) \in E$ there is a channel, which is capable of holding at most a single message of M

Execution of the system begins with all processes in arbitrary start states and all channels empty

Processes repeatedly performing the following two actions in lock-step:

1. Apply the message-generation function to the current state to generate the messages to be sent to all outgoing neighbours, putting these messages in the appropriate channels

2. Apply the state-transition function to the current state and the incoming messages to obtain the new state, removing all messages from the channels

Synchronous Network Model - Notes

Takes no account of the computation a process undertakes to compute the values of its message and state generation functions

Easy to distinguish some of the process states as halting states, specifying that no further computation should take place once such a state is entered

- Halting states do not play the same role in these systems as they do in finite-state automata

- Commensurate notion of accepting state is not normally used in distributed algorithms

Some systems require that our synchronous model allow for undirected graphs, which we can account for using bidirectional edges between all pairs of neighbours

Synchronous Network Model - What Can Go Wrong?

Can account for process failures and channel failures

Link failure - A channel can fail by losing messages

Stop failure - A process can exhibit stopping failure simply by stopping somewhere in the middle of its execution e.g., between the first and second step in a round

Byzantine failure - A process can exhibit Byzantine failure, which means that it can generate its next messages and next state in some arbitrary way (not necessarily following the rules specified by its message generation and state transition functions)

The Leader Election Problem

The Leader Election Problem

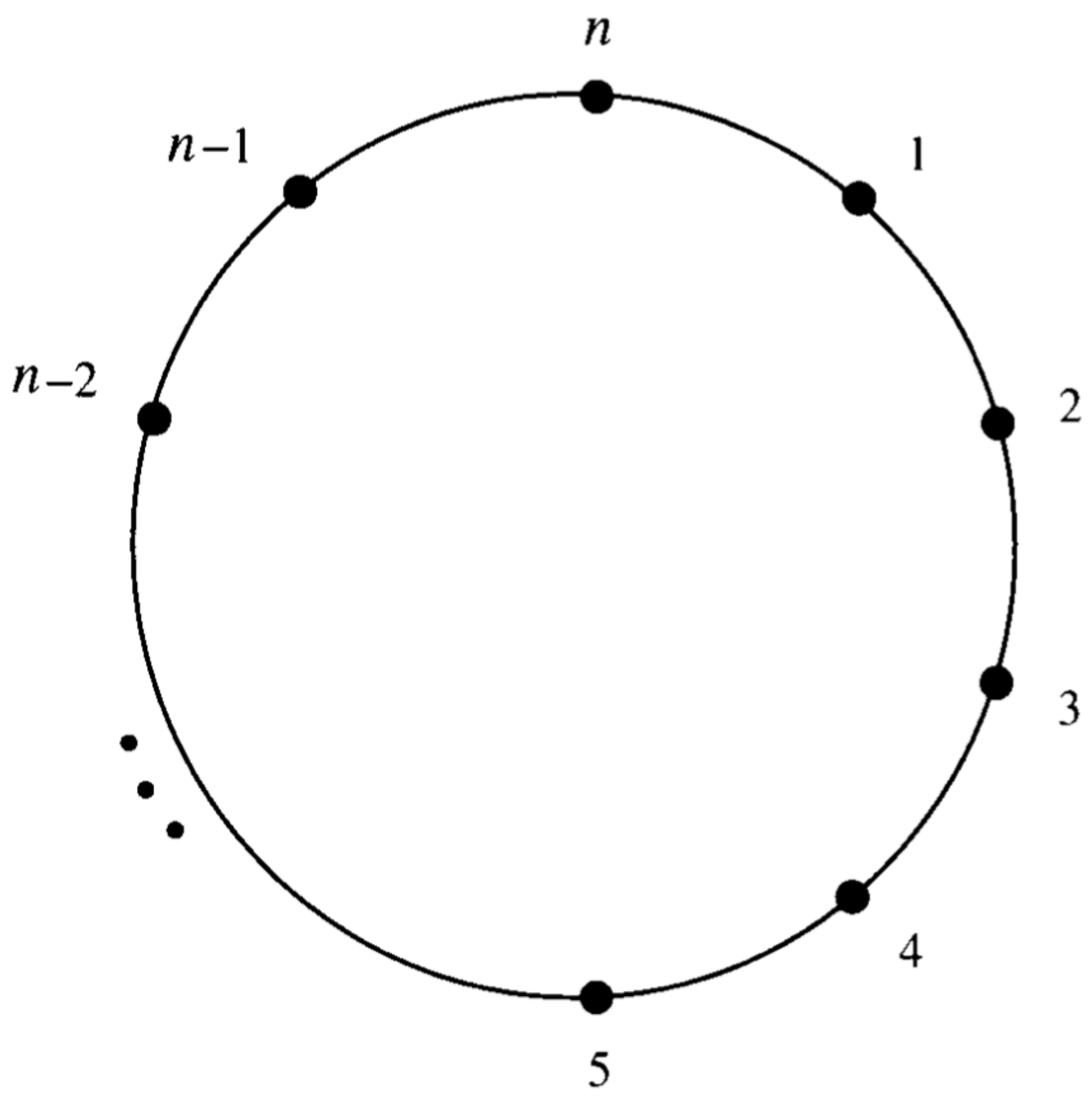
Found to be as a significant problem when studying token ring networks

A set of processes arranged in a logical ring

- In each turn, each process holds a token, and can then send messages on the network

Token can get lost, which means it must be possible to regenerate the token

- **Regeneration process amounts to electing a leader in the network**



The Leader Election Problem

There are actually several variants of the classical problem, for example:

- The ring might be unidirectional or bidirectional
- The number of processes in the ring may be known or unknown
- Non-leaders might be required to output some acknowledgement that they are not the leader
- Unique identifiers may be consecutive integers or otherwise
- Unique identifiers may admit operations other than comparison, e.g., swap, or otherwise

The Leader Election Problem

We know that a specification can be broken down into two parts

- Safety
- Liveness

Safety - “Nothing bad happens”

Liveness - “Sometimes something good will happen”

- Notion of sometimes depends on the system model. e.g., under an asynchronous system model sometimes means eventually

The Leader Election Problem

Safety for leader election

At any one time, there is at most one leader in the system (if a node is elected leader then all previous leaders have crashed)

Liveness for leader election

Sometimes, there exists a leader in the system (either no correct node exists or a node is eventually elected leader)

In a synchronous system, we can take sometimes to mean within a time bound

In a worse case, we can take it to mean eventually

Leader Election In A Token Ring

The Leader Election Problem - Our System Model

Network is a connected ring

Nodes have consecutive unique identifiers $0,1, \ldots, N-1$

System synchronous model (with rounds operating in the way we have described)

Each node knows its right and left neighbour

Addition is module-N

Unidirectional ring of unknown size

Leader Election - LCR Algorithm

Le Lann, Chang, and Roberts (LCR) first proposed the essential algorithm

Each process sends its identifier around the ring

When a process receives an incoming identifier, it compares that identifier to its own

If the incoming identifier is:

- greater than its own, it keeps passing the identifier;
- less than its own, it discards the incoming identifier;
- equal to its own, the process declares itself the leader

Leader Election - LCR Algorithm

Each node p sends its identifier, id_p , around the ring

When p receives an identifier, id_q , it can do one of three things:

- If $id_p = id_q$ then n is elected leader
- If $id_p > id_q$ then do nothing
- If $id_p < id_q$ then send id_q to neighbour

Process with highest identifier is elected leader

Leader Election - LCR Algorithm Formally

M: The set of unique identifiers

For each i, the state in $states_i$ are:

- u, a unique identifier that is initially i's unique identifier
- $send$, a unique identifier or $null$ that is initially i's unique identifier
- $status$, with values in $\{unknown, leader\}$ that is initially $unknown$

For each i, the start states $start_i$ is:

- Defined by unique identifier allocation

Leader Election - LCR Algorithm Formally

For each i, the message generation function $msgs_i$ is:

Send the current value of *send* to process $i + 1$

For each i, the transition function $trans_i$ is:

```
$send := null$
if the incoming message is $v$, a UID, then
    case
        $v > u$: $send := v$
        $v = u$: $status := leader$
        $v < u$: do nothing
    endcase
```

Does This Algorithm Work? Can We Prove It?

We need to show that the algorithm satisfies both safety and liveness

If either of these is violated, the algorithm does not solve the leader election problem

Very often we can base our proof on the use of **invariant assertions**

Assertions generally proved by induction on the number of steps in execution

Proving Liveness

Liveness for Leader Election - Either no correct process exists or a process is eventually elected leader

Trivial to show that, if no correct process exists, there will be no leader

What we need to show: Process i_{max} outputs leader by the end of round n

> **Proof:** We know u_{max} is the initial value of variable $u_{i_{max}}$ (the variable u at process i_{max}) by initialisation. We can see that the values of the u variables never change (by inspection / code), they are all distinct (by assumption), and i_{max} has the largest u value (by definition of i_{max})
>
> Thus it suffices to show that after n rounds $status_{i_{max}} = leader$

Proving Liveness - Invariant Assertions

Invariant Assertion 1: After n rounds $status_{i_{max}} = leader$

Prove with induction on the number of rounds, with an invariant on small numbers of rounds

Invariant Assertion 2: For $0 \leq r \leq n - 1$ after r rounds, $send_{i_{max}+r} = u_{max}$

For $r = 0$ the initialisation says that $send_{i_{max}} = u_{max}$ after 0 rounds, which is what we desire

Every process other than i_{max} accepts the maximum value and places it into its channel, since u_{max} is greater than all other values

Having shown Assertion 2, we use its special case for $r = n - 1$, process i_{max} accepts u_{max} as a signal to set its status to $leader$

Proving Safety

Safety for Leader Election - At any one time, there is at most one leader in the system

What we need to show: No process other than i_{max} ever outputs the value $leader$

Proof: It is enough to show that all other processes always have $status = unknown$ by, once again, relying on a stronger invariant

If i and j are any two processes such that $i \neq j$ and $[i, j)$ defines indices $\{i, i+1, \ldots, j-1\}$, we need an invariant assertion states that no identifier v can reach any $send$ variable in any position between i_{max} and v's origin i

Proving Safety - Invariant Assertions

Invariant Assertion: For any r and any i and j, after r rounds, if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then $send_j \neq u_i$

A non-maximum value can never get past i_{max}, since the i_{max} values is compared with the incoming value u_{max} and u_{max} is greater than all other identifiers

As i_{max} can only receive its own identifier, only i_{max} can output $leader$

Having demonstrated liveness and safety, we know LCR solves the leader election problem

How well does it solve the problem?

Is halting an issue?

Complexity Analysis - LCR Algorithm

Synchronous system model allows us to consider the number of rounds in relation to time complexity but it is also important to consider communication complexity

It will be n rounds before a leader is elected

The message complexity is $O(n^2)$

We look for lower bounds so the we do not keep trying to improve the complexity beyond the theoretical bound

Leader Election - LCR Halting

As we have it, almost all processes never reach a halting state

Modify the algorithm to incorporate a *finished* message that is initiated by the elected leader

- Any process receiving a *finished* message will halt after it has passed it along
- It will be $2n$ rounds before a leader is elected
- The message complexity is still $O(n^2)$

Benefit that non-leaders can declare themselves as a non-leader

Leader Election - Improving Message Complexity

LCR has linear time complexity but the communication complexity of $O(n^2)$ in undesirable

- Particularly significant in the context of concurrently running distributed algorithms

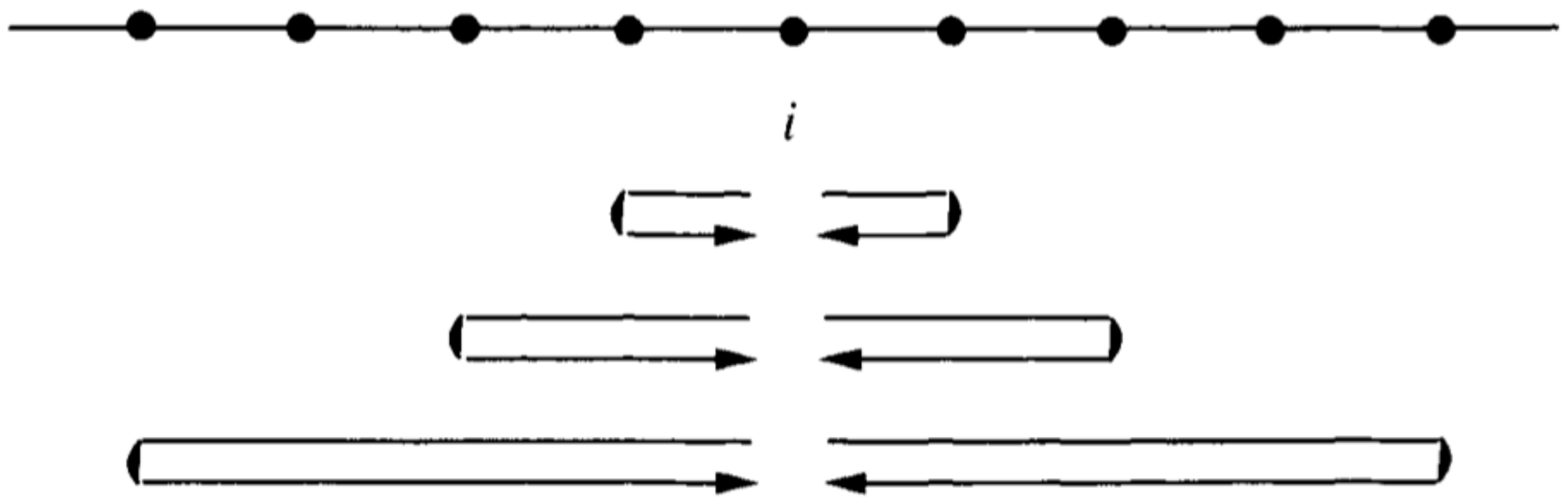
Hirschberg and Sinclair (HS) published the first algorithm with $O(n\ logn)$ communication complexity

- Ring size can be unknown
- Bidirectional communication

Leader Election - HS Algorithm

Each process i operates in phases $0,1,2,\ldots$

In each phase l, process i sends out "tokens" containing its identifier u_i in each direction



These are intended to travel distance 2^l, before returning to their origin i

If both tokens make it back safely then process i continues with the following phase (note: the implication is that tokens might not make it back safely)

Leader Election - HS Algorithm

While a u_i token is proceeding in the outbound direction, each other process j on the u_i path compares u_i with its own identifier u_j

If $u_i < u_j$ then j discards the token

f $u_i > u_j$ then j relays u_i

If $u_i = u_j$ then it means process j has received its own identifier before the token has turned around, hence j elects itself $leader$

Leader Election - HS Algorithm Formally

M: The set of triples consisting of $< identifier, direction\text{-}flag, hop\text{-}count >$

For each i, the state in $states_i$ are:

u, a unique identifier (initially i's unique identifier)
$send+$, containing either an element of M or $null$ (initially $< u_i, out, 1 >$)
$send-$, containing either an element of M or $null$ (initially $< u_i, out, 1 >$)
$status$, with values in $\{unknown, leader\}$ (initially $unknown$)
$phase$, a non-negative integer (initially 0)

For each i, the start states $start_i$ is:

Defined single state based on initialisations

Leader Election - HS Algorithm Formally

For each i, the message generation function $msgs_i$ is:

Send the current value of $send$ to process $i + 1$

Send the current value of $send$ to process $i - 1$

Leader Election - HS Algorithm Formally

For each i, the transition function $trans_i$ is:

```
$send+ := null$
$send- := null$
if the message from $i-1$ is $(v, out, h)$ then
    case
        $v > u$ and $h > 1$: $send+ := (v, out, h-1)$
        $v > u$ and $h = 1$: $send- := (v, in, 1)$
        $v = u$: $status := leader$
    endcase
if the message from $i+1$ is $(v, out, h)$ then
    case
        $v > u$ and $h > 1$: $send- := (v, out, h-1)$
        $v > u$ and $h = 1$: $send+ := (v, in, 1)$
        $v = u$: $status := leader$
    endcase
if the message from $i-1$ is $(v, in, 1)$ and $v \neq u$ then
    $send+ := (v, in, 1)$
if the message from $i+1$ is $(v, in, 1)$ and $v \neq u$ then
    $send- := (v, in, 1)$
if the messages from $i-1$ and $i+1$ are both $(u, in, 1)$ then
    $phase := phase + 1$
    $send+ := (u, out, 2^{phase})$
    $send- := (u, out, 2^{phase})$
```

Complexity Analysis - HS Algorithm

- Every process sends out a token in phase 0
    - $4n$ messages for the both initial tokens to complete outbound and inbound journeys in each direction
    - For $l > 0$ a process sends a token in phase l if it receives both its phase $l - 1$ tokens on their return
    - The occurs only when it has not been defeated by another process within distance 2^{l-1} in either direction
        - Within any group of $2^{l-1} + 1$ consecutive processes, at most one goes on to initiate tokens at phase l

Complexity Analysis - HS Algorithm

On that basis we can calculate that he number of processes that initiate tokens is at most:

$$\lfloor \frac{n}{2^{l-1}+1} \rfloor$$

This means that the total number of messages sent out at phase l is bounded by:

$$4(2^l \lfloor \frac{n}{2^{l-1}+1} \rfloor) \leq 8n$$

This is because phase 1 tokens travel distance 2^l

The factor of 4 comes from the fact that the token is sent out in both directions and that each outbound token must return

Complexity Analysis - HS Algorithm

The total number of phases that are executed before a leader is elected and all communication stops is, at most:

$$1 + \lceil log n \rceil$$

This means that the total number of messages is, at most:

$$8n(1 + \lceil log n \rceil)$$

Hence, the communication complexity is $O(n\ log n)$

Complexity Analysis - HS Algorithm

The overall time complexity for the HS algorithm is $O(n)$

To see this we can start by noting that the time for each phase l is given by the time for the tokens to go out and return, which is given by:

$2(2^l) = 2^{l+1}$

That is true apart from final phase, which takes time n because there is no return for the tokens

In the penultimate phase $l = \lceil logn \rceil - 1$, where its time complexity is at least as great as the total time complexity of all preceding phases - implying that all but the final phases is at most:

$2(2^{\lceil logn \rceil})$

Non-comparison Algorithms

The communication complexity of algorithms that only permit comparisons between process identifiers, including LCR and HS, is bounded by $\Omega(n\ logn)$

If we permit our algorithm to modify process identifiers, we can improve on this results

We can achieve $O(n)$ communication complexity at the expense of time complexity

If we permit our algorithm to modify process identifiers, we can improve on this results

The $TimeSlice$ and $VariableSpeeds$ algorithms are the two most prominent examples

Non-comparison Algorithms - TimeSlice

Computation in phases $1,2,\ldots$, where each phase consists of n consecutive rounds

Each phase is devoted to the possible circulation (around the entire ring) of a token carrying a identifier

In phase v, which consists of rounds $(v-1)n+1$, only a token carrying identifier v is allowed to circulate

If process i with identifier v exists and round $(v-1)n+1$ is reached without i having previously received a non-null message, process i elects itself $leader$ and sends its identifier around the ring

- As this token travels, all other processes see that they have received it, preventing them from electing themselves or initiating the sending of a token at any later phase

Minimum identifier U_{min} eventually gets around the ring, causing its originating process to be elected

Non-comparison Algorithms - TimeSlice Complexity

No messages are sent before round $(u_{min} - 1)n + 1$ and no messages are sent after round nu_{min}

The total number of messages send is n

Improves on $\Omega(n\ logn)$ communication complexity by allowing modification

Time complexity is approximately nu_{min}, which is unbounded even in the case of a fixed size ring

Only useful for small rings where identifiers are known to be small positive integers

Non-comparison Algorithms - VariableSpeeds

Although the algorithms achieves $O(n)$ message complexity, we will not study it in great detail because it is not practical, since its time complexity is $O(n2^{u_{min}})$

A counter example to the $\Omega(n\ logn)$ bound on message complexity for unknown ring sizes

Each process i initiates a token, which travels around the ring, carrying identifier u_i

Different tokens travel at different rates, with a token carrying identifier v travelling at a rate of one message every 2^v rounds

Each process tracks the smallest identifier it has seen and discards anything it sees that is larger

If a token returns to its origin process, that process is elected *leader*

Leader Election In A General Network

Generalising Our Approach

Strongly connected network digraph $G = (V, E)$ with $|V| = n$

In order to name processes we assign indices $1,2, \ldots, n$ but (unlike the ring's indices) these have no connection to the process' position] in the graph

The processes do not know their indices or those of their neighbours, instead referring to their neighbours by local names

> If a process i has the same process j for both an incoming and outgoing neighbour, then i knows that the two processes are the same

General Network - Does Leader Election Change?

As for a ring network, there can be different variants of the problem based on the assumptions made when formulating it

- The number of processes in the network (or an upper bound on it) may be known or unknown
- The diameter, $diam$, of the network (or an upper bound on it) may be known or unknown
- Non-leaders might be required to output some acknowledgement that they are not the leader

Leader Election In A General Network - Flooding

Every process maintains a record of the highest identifier it has seen so far

At each round, each process propagates this maximum to all neighbours

After *diam* rounds, if the maximum identifier seen is the process's own then it outputs *leader*, otherwise is outputs *non-leader*

Leader Election In A General Network - Flooding

M: The set of unique identifiers

For each i, the state in $states_i$ are:

u, a unique identifier (initially i's unique identifier)

$max\text{-}uid$, a unique identifier (initially i's unique identifier)

$status$, a value in $\{unknown, leader, non\text{-}leader\}$ (initially $unknown$)

$rounds$, an integer (initially 0)

For each i, the start states $start_i$ is:

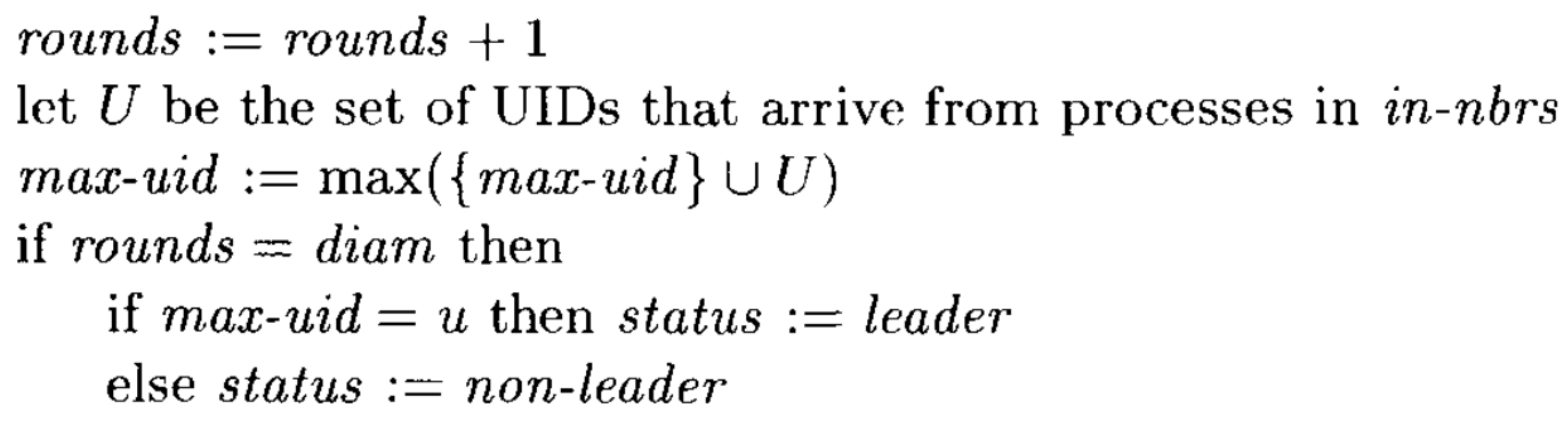
Defined by unique identifier allocation

Leader Election In A General Network - Flooding

For each i, the message generation function $msgs_i$ is:

If $rounds < diam$ then send $max\text{-}id$ to all $j \in out\text{-}nbrs$

For each i, the transition function $trans_i$ is:



```
$rounds := rounds + 1$
let $U$ be the set of UIDs that arrive from processes in $in\text{-}nbrs$
$max\text{-}uid := \max(\{max\text{-}uid\} \cup U)$
if $rounds = diam$ then
    if $max\text{-}uid = u$ then $status := leader$
    else $status := non\text{-}leader$
```

The Correctness Of Our Flooding Approach

What we need to show: Process i_{max} outputs $leader$ and each other process outputs $non\text{-}leader$ within $diam$ rounds

Translate above into corresponding assertion using code / system variables

Proof: It suffices to show that after $diam$ round, $status_{i_{max}} = leader$ and $status_j = non\text{-}leader$ for every $j \neq i_{max}$

Must demonstrate that after r rounds, the maximum identifier has reached every process within distance r of i_{max} when measured along directed paths in G

The Correctness Of Our Flooding Approach

Invariant Assertion 1: For $0 \leq r \leq diam$ and for every process j, after r rounds, if the distance from i_{max} to j is at most r, then $max\text{-}uid_j = u_{max}$

Implies that every process has the maximum identifier by the end of $diam$ rounds

Invariant Assertion 2: For every r and j, after r rounds, $rounds_j = r$

Invariant Assertion 3: For every r and j, after r rounds, $max\text{-}uid_j \leq u_{max}$

Specialising Assertions 1, 2 and 3 to $r = diam - 1$ and accounting for what happens in round $diam$ we demonstrate that after $diam$ round, $status_{i_{max}} = leader$ and $status_j = non\text{-}leader$ for every $j \neq i_{max}$

Complexity Analysis - Flooding

It will be $diam$ rounds before a leader is elected

Follows from the definition of the algorithm

The number of messages sent is $diam|E|$, where $|E|$ is the number of directed edges

A message is sent on every directed edge for each of the first $diam$ rounds

Approach still works with an upper bound on the diameter rather than the actual diameter

Can we reduce the communication complexity?

Optimisations To Flooding for Leader Election

Processes could send their $max\text{-}uid$ values only when they first learn about them

This simple optimisation will decrease the communication complexity in many cases, though the worst-case analysis remains valid

If process i receives a new maximum from a process j that is both an incoming and outgoing neighbour then i does not send a message to j in the next round

A minor improvement but does improve performance in real-world situations

Example Exam Question

Leader Election - A Very Nice Question

(a) Define the leader election problem. **[5 marks]**

(b) Develop an algorithm that solves leader election in a synchronous unidirectional ring network. State any assumptions. **[5 marks]**

(c) Analyse the time complexity of your algorithm. **[5 marks]**

(d) The network is modified so that it permits bidirectional communication. Explain the implications of the network permitting bidirectional communication for the algorithm you have developed. You should state whether your algorithm solves the leader election problem in the modified network. **[5 marks]**