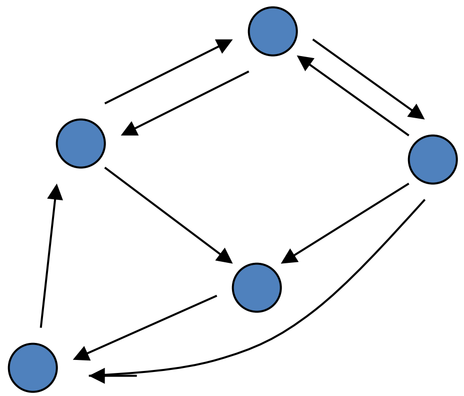
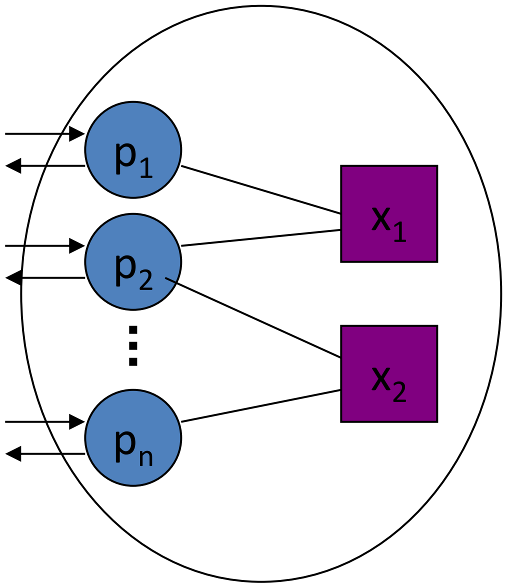
# 6.852: Distributed Algorithms
# Fall, 2015

Instructor: Nancy Lynch
TAs: Cameron Musco, Katerina Sotiraki
Course Secretary: Joanne Hanley

# What are Distributed Algorithms?



- Algorithms that run on networked processors, or on multiprocessors that share memory.
- They solve many kinds of problems:
  - Communication
  - Data management
  - Resource management
  - Synchronization
  - Reaching consensus
  - Etc.
- They work in difficult settings:
  - Concurrent activity at many processing locations
  - Uncertainty of timing, order of events, inputs
  - Failure and recovery of processors, of channels.
- So they can be complicated:
  - Hard to design
  - Hard to prove correct
  - Hard to analyze



# This course

- A theoretical CS course.
- Takes a mathematical approach to studying distributed algorithms.
- Q: What does that mean?
- Approach:
  - Define models of distributed computing platforms.
  - Define abstract problems to solve in distributed systems.
  - Develop algorithms that solve the problems on the platforms.
  - Analyze their complexity.
  - Try to improve (“optimize”) them.
  - Identify inherent limitations, prove lower bounds and impossibility results.
- Like theory for sequential algs, but more complicated.

# Distributed algorithms research

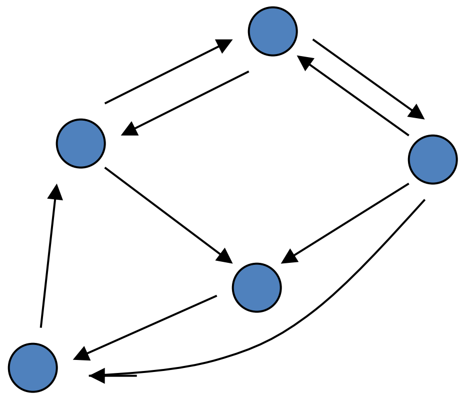
- $>$ 45 years, starting with Dijkstra and Lamport
- PODC, DISC, SPAA, OPODIS; also ICDCS, STOC, FOCS, SODA,…
- Abstract problems derived from practice, in networking and multiprocessor programming.
- Static theory:
  - Assumes fixed network or shared-memory setting.
  - Participants, and their configuration, may be generally known.
- Dynamic theory:
  - Client/server, peer-to-peer, cloud, wireless, mobile ad hoc, robot swarms
  - Participants may join, leave, move.
- Theory for modern multiprocessors:
  - Multicore processors
  - Transactional memory

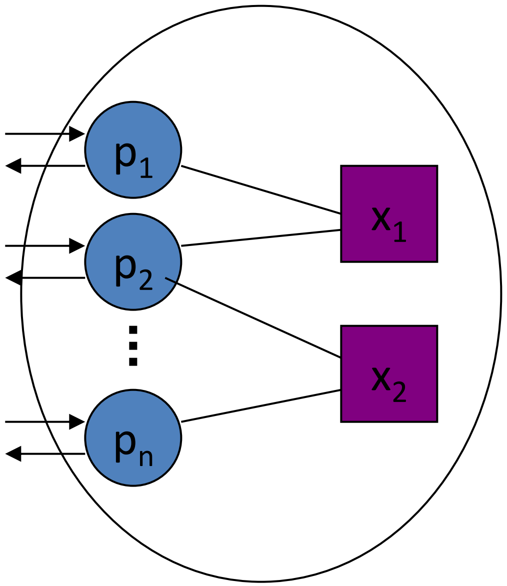
# Administrative info (Handout 1)

- People and places
- What is the course about?
- Prerequisites
  - Math, systems, algorithms
  - Formal models of computation, e.g., automata:
    - Some courses (6.045, 6.840) study general theory of automata.
    - In 6.852, automata are tools, to model algorithms and systems.
    - Necessary, because the algorithms are complicated.
- Source material
  - Books, papers
- Course requirements
  - Readings
  - Problem sets
    - Given out every week, due every two weeks
    - Collaboration policy
  - Grading
  - Term projects:
    - Reading, theoretical research, or experimental research

# Topics (Handout 2)

- Many different model assumptions:
  - Inter-process communication method:
    - Message-passing, shared memory.
  - Timing assumptions:
    - Synchronous (rounds)
    - Asynchronous (arbitrary speeds)
    - Partially synchronous (some timing assumptions, e.g., bounds on message delay, processor speeds, clock rates)
  - Failures:
    - Processor: Stopping, Byzantine
    - Communication: Message loss, duplication; Channel failures, recovery
    - Total system state corruption
- Main organization: By timing model.





# Topics

- Synchronous model: Classes 1-8.
  - Basic, easy to use for designing algorithms.
  - Not realistic, but sometimes can be emulated in worse-behaved networks.
  - Impossibility results for synchronous networks carry over to worse networks.
- Asynchronous: Classes 9-22.
  - More realistic, but harder to cope with.
- Partially synchronous: In between. Probably won't have time this year...

- Special topics at the end: Failure detectors, self-stabilization, biological distributed algorithms.

# In more detail…

- Synchronous networks:
  - Model
  - Leader election (symmetry-breaking)
  - Network searching, spanning trees, Minimum Spanning Trees (MST's)
  - Maximal Independent Sets (MIS's), Coloring
  - Processor failures: Stopping and Byzantine failures
  - Fault-tolerant consensus: Algorithms and lower bounds
  - Other problems: Commit, approximate agreement, k-agreement
- Modeling asynchronous systems (I/O automata)
- Asynchronous networks, no failures:
  - Models and proofs
  - Leader election, network searching, spanning trees, revisited.
  - Synchronizers (for emulating synchronous algorithms in asynchronous networks)
  - Logical time, replicated state machines.
  - Stable property detection (termination, deadlock, snapshots).

# In more detail…

- Asynchronous shared-memory systems, no failures:
  - Models
  - Mutual exclusion algorithms and lower bounds
  - Resource allocation, Dining Philosophers
- Asynchronous shared-memory systems, with failures:
  - Impossibility of consensus
  - Atomic (linearizable) objects, atomic read/write registers, atomic snapshots
  - Wait-free computability; wait-free consensus hierarchy; wait-free vs. $f$-fault-tolerant objects
- Asynchronous networks, with failures:
  - Asynchronous networks vs. asynchronous shared-memory systems
  - Impossibility of consensus, revisited
  - Paxos consensus algorithm

# In more detail…

- Failure detectors
- Self-stabilizing algorithms
- (Partially-synchronous systems and timing-based algorithms:
  - Models and proofs, timed I/O automata
  - Mutual exclusion, consensus
  - Clock synchronization)
- (Distributed algorithms for dynamic networks:
  - Atomic memory
  - Virtual Nodes
  - Computing functions in dynamic networks)
- Biological distributed algorithms
  - Social insect colony algorithms:  Foraging, task-allocation, house-hunting

# Supplementary Readings (on line)

- Other books:
  - [Attiya, Welch], general distributed algorithms
  - [Dolev], self-stabilization
  - [Peleg], local network computation
  - [Kaynar, Lynch, Segala, Vaandrager], interacting automata modeling for distributed algorithms/systems
  - Morgan Claypool monograph series on Distributed Computing Theory
- Dijkstra Prize papers, 2000-2015
- A variety of other interesting papers
- Also check out proceedings for PODC, DISC, etc.

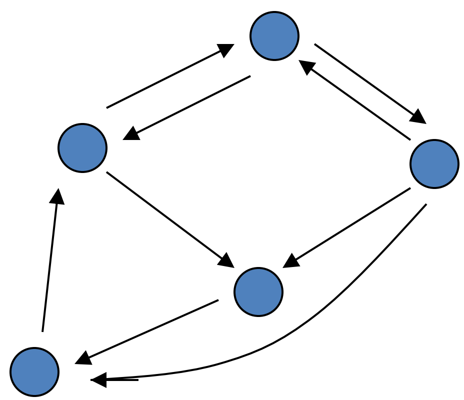
# Now start the actual course…

- Rest of today:
  - Synchronous network model
  - Leader election problem, in simple ring networks
- Reading: Chapter 2; Sections 3.1-3.5.
- Next Tuesday: Sections 3.6, 4.1-4.3

- Questions?

# Synchronous network model

- Processes at nodes of a digraph, communicate using messages.
- Digraph: $G = (V, E)$, $n = |V|$
  - $outnbrs_i, innbrs_i$
  - $distance(i, j)$, number of hops on shortest path from $i$ to $j$.
  - $diam = max_{ij} distance(i, j)$
- $M$: Message alphabet, plus $\perp$ placeholder
- For each $i \in V$, a process consisting of :
  - $states_i$, a nonempty, not necessarily finite, set of states
  - $start_i$, a nonempty subset of $states_i$
  - $msgs_i$: $states_i \times outnbrs_i \rightarrow M \cup \{\perp\}$
  - $trans_i$: $states_i \times$ (vectors of $M \cup \{\perp\}$) $\rightarrow states_i$
- Executes in rounds:
  - Apply $msgs_i$ to determine messages to send,
  - Send and collect messages,
  - Apply $trans_i$ to determine the new state.



# Remarks

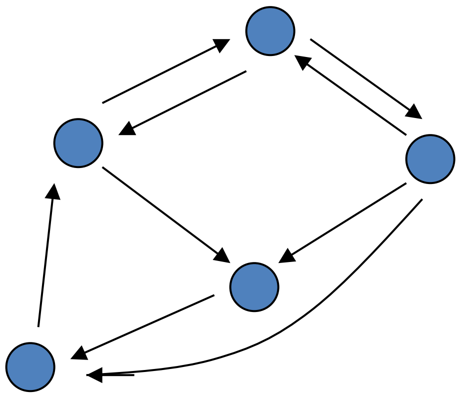
- No restriction on amount of local computation.
- Deterministic (a simplification).
- Later, we will consider some complications:
  - Variable start times
  - Failures
  - Random choices
- Can define “halting states”, but not used as accepting states as in traditional automata theory.
- Inputs and outputs: Can encode in the states, e.g., in special input and output variables.

# Executions

- An execution is a mathematical notion used to describe how an algorithm operates.
- Definition (p. 20):
  - State assignment: Mapping from process indices to states.
  - Message assignment: Mapping from ordered pairs of process indices to $M \cup \{\perp\}$.
  - Execution: $C_0, M_1, N_1, C_1, \mathrm{M}_2, \mathrm{N}_2, \mathrm{C}_2, \ldots,$
    - $C$'s are state assignments.
    - $M$'s are message assignments representing messages sent.
    - $N$'s are message assignments representing messages received.
    - Infinite sequence, in general.

# Leader election

- Network of processes.
- Want to distinguish exactly one, as the leader.
- Formally: Eventually, exactly one process should output "leader" (set special $status$ variable to "leader").

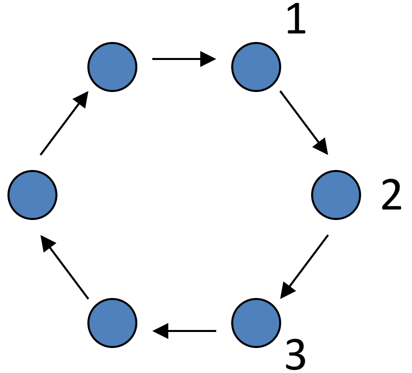


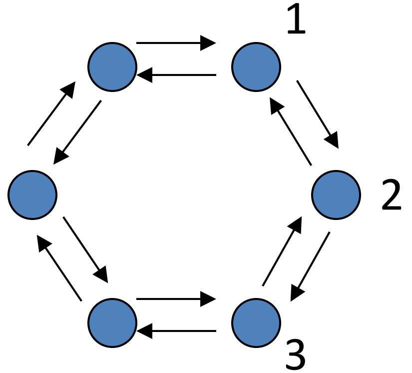
- Motivation: Leader can take charge of:
  - Communication
  - Coordinating data processing
  - Allocating resources
  - Scheduling tasks
  - Coordinating consensus protocols
  - …

# Simple case:  Ring network

- Variations:
  - Unidirectional or bidirectional
  - Ring size $n$ can be known or unknown
- Numbered clockwise
- Processes don't know the numbers; know neighbors by the names "clockwise" and "counterclockwise".

- Theorem 1:  Suppose all processes are identical (same sets of states, transition functions, etc.).  Then it's impossible to elect a leader, even under the most favorable  assumptions (bidirectional communication, ring size $n$ known to all).



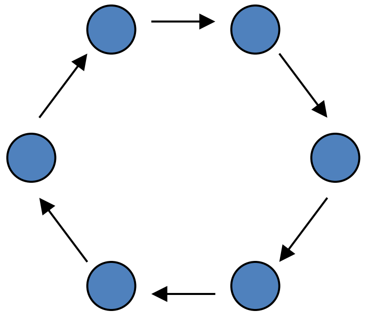
# Proof of Theorem 1

- By contradiction. Assume an algorithm that solves the problem.
- Assume WLOG that each process has exactly one start state (if more, we could choose same one for all processes).
- Then there is exactly one execution, say:
  - $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$
- Prove by induction on the number $r$ of completed rounds that all processes are in identical states after $r$ rounds.
  - Generate the same messages to corresponding neighbors.
  - Receive the same messages.
  - Make the same state changes.
- Since the algorithm solves the leader election problem, someone eventually gets elected.
- Then everyone gets elected, contradiction.

# So we need something more…

- To solve the problem at all, we need something more---some way of distinguishing the processes.
- E.g., assume processes have unique identifiers (UIDs), which they “know”.
  - Formally, each process starts with its own UID in a special state variable.
- UIDs are elements of some data type, with specified operations, e.g.:
  - Abstract totally-ordered set with just $(<, =, >)$ comparisons.
  - Integers with full arithmetic.
- Different UIDs can appear anywhere in the ring, but each can appear only once.

# A basic algorithm
## [LeLann] [Chang, Roberts]



- Assumes:
  - Unidirectional communication (clockwise)
  - Processes don't know $n$
  - UIDs, comparisons only
- Idea:
  - Each process sends its UID in a message, to be relayed step-by-step around the ring.
  - When process receives a UID, it compares it with its own.
  - If the incoming UID is:
    - Bigger, pass it on.
    - Smaller, discard.
    - Equal, the process declares itself the leader.
  - This algorithm elects the process with the largest UID.

# In terms of our formal model:

- $M$, the message alphabet: The set of UIDs
- $states_i$: Consists of values for three state variables:
  - $u$, holds its own UID
  - $send$, a UID or $\bot$, initially its own UID
  - $status$, one of {? , leader}, initially ?
- $start_i$: Defined by the initializations.
- $msgs_i$: Send contents of $send$ variable, to clockwise neighbor.
- $trans_i$:
  - Defined by pseudocode (p. 28):

    if incoming $=$ $v$, a UID, then
    
    case
    
    $v > u$: $send := v$
    
    $v = u$: $status :=$ leader
    
    $v < u$: Do nothing.
    
    endcase
  - Entire block of code is treated as atomic (performed instantaneously).

# Correctness proof

- Prove that exactly one process ever gets elected leader.
- More strongly:
  - Let $i_{max}$ be the process with the max UID, $u_{max}$.
  - Prove:
    - $i_{max}$ outputs “leader” by the end of round $n$.
    - No other process ever outputs “leader”.

# $i_{max}$ outputs "leader" after $n$ rounds

- Prove using induction on the number of rounds?
- Requires strengthening the statement, to say something about the situation after $r$ rounds, $0 \leq r < n$.

- Lemma 2: For $0 \leq r \leq n-1$, after $r$ rounds, the $send$ variable at process $(i_{max} + r) \; mod \; n$ contains $u_{max}$.
- That is, $u_{max}$ makes its way systematically around the ring.
- Proof :
  - Induction on $r$.
  - Base: By the initialization.
  - Inductive step: Because everyone else lets $u_{max}$ pass through.

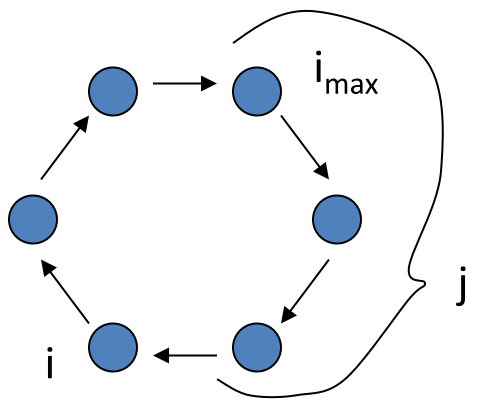
- Use Lemma 2 for $r = n-1$, and a little argument about the $n^{th}$ round to show that the correct output happens.
  - When $u_{max}$ arrives at $i_{max}$, $i_{max}$ sets its $status$ to leader.

# Uniqueness

- No one except $i_{max}$ ever outputs "leader".
- Again, strengthen claim:
- Lemma 3: For any $r \geq 0$, after $r$ rounds, if $i \neq i_{max}$ and $j$ is any process in the interval $[i_{max}, i)$, then $j$'s $send$ doesn't contain $u_i$.
- Thus, $u_i$ doesn't get past $i_{max}$ when moving around the ring.

- Proof:
  - Induction on $r$.
  - Key fact: $i_{max}$ discards $u_i$ (if it hasn't already been discarded).

- Use Lemma 3 to show that no one except $i_{max}$ ever receives its own UID, so no one else ever elects itself.



# Invariant proofs

- Lemmas 2 and 3 are examples of invariants---properties that are true in all reachable states.
- Another invariant: If $r = n$ then the $status$ variable of $i_{max} =$ leader.
- Usually proved by induction on the number of steps in an execution.
  - Usually need to strengthen them, to prove them by induction.
  - Inductive step requires case analysis.
- In this class:
  - We'll outline key steps of invariant proofs, not present all details.
  - We'll assume you could fill in the details if necessary.
  - You should work out at least a few examples in detail.
- Invariant proofs are overkill for this simple example, but:
  - Similar proofs work for much harder synchronous algorithms.
  - Also for asynchronous algorithms, and partially synchronous algorithms.
  - The properties, and proofs, are more subtle in those settings.
- Invariants are the most useful tool for proving properties of distributed algorithms.

# Complexity bounds

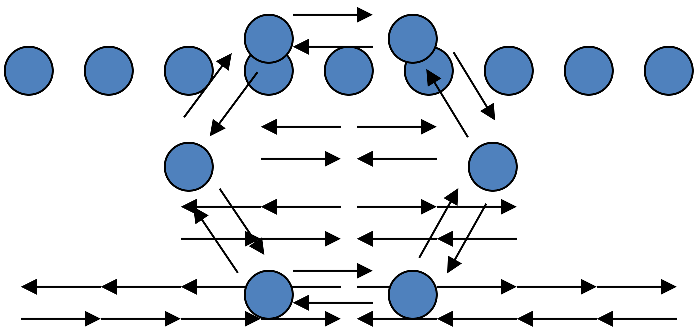
- What to measure?
  - Time = number of rounds until “leader”: $n$
  - Communication = number of single-hop messages: $\leq n^2$
- Variations:
  - Non-leaders announce “non-leader”:
    - Any process announces “non-leader” as soon as it sees a UID higher than its own.
    - No extra costs.
  - Everyone announces who the leader is:
    - At end of $n$ rounds, everyone knows the max.
      - No extra costs.
      - Relies on synchrony and knowledge of $n$.
    - Or, leader sends a special “report” message around the ring.
      - Total time: $\leq 2n$
      - Communication: $\leq n^2 + n$
      - Doesn’t rely on synchrony or knowledge of $n$.

# Halting

- Formally: Add halt states, special “looping” states from which all transitions leave the state unchanged, and that generate no messages.
- For all problem variations:
  - Can halt after $n$ rounds.
    - Depends on synchrony and knowledge of $n$.
  - Or, halt after receiving leader’s “report” message.
    - Does not depend on synchrony or knowledge of $n$
- Q: Can a process just halt (for the basic problem) after it sees and relays some UID larger than its own?
- No---it must stay alive to relay later messages.

# Reducing the communication complexity [Hirschberg, Sinclair]

- $O(n \log n)$, rather than $O(n^2)$
- Assumptions:
  - Bidirectional communication
  - Ring size not known.
  - UIDs with comparisons only



- Idea:
  - Successive doubling strategy
    - Used in many distributed algorithms where network size is unknown.
  - Each process sends a UID token in both directions, to successively greater distances (double each time).
  - Going outbound: Token is discarded if it reaches a node whose UID is bigger.
  - Going inbound: Everyone passes the token back.
  - Process begins next phase only if/when it gets both its tokens back.
  - Process that gets its own token in outbound direction, elects itself the leader.

# In terms of formal model:

- Needs local process description.
- Involves bookkeeping, with hop counts.
- LTTR (p. 33)

# Complexity bounds

- Time:
  - Worse than [LCR] but still $O(n)$.
  - Time for each phase is twice the previous, so total time is dominated by last complete phase (geometric series).
  - Last phase is $O(n)$, so total is also.

# Communication bound: $O(n \log n)$

- $1 + \lceil \log n \rceil$ phases, numbered $0,1,2,\ldots$
- Phase $0$: All send messages one hop both ways, $\le 4n$ messages.
- Phase $k > 0$:
  - Within any block of $2^{k-1} + 1$ consecutive processes, at most one is still alive at the start of phase $k$.
    - Others' tokens discarded in earlier phases, stop participating.
  - So at most $\lfloor n / (2^{k-1} + 1) \rfloor$ start phase $k$.
  - Total number of messages at phase $k \le 4\ (2^k \lfloor n / (2^{k-1} + 1) \rfloor\ ) \le 8n$

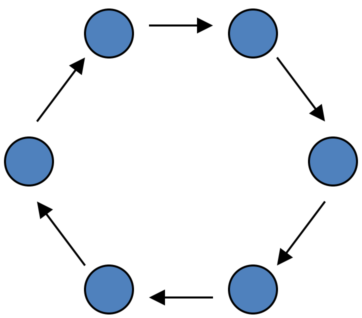
Out and back, both directions

Distance

- So total communication $\le 8\,n\,(1 + \lceil \log n \rceil) = O(n \log n)$

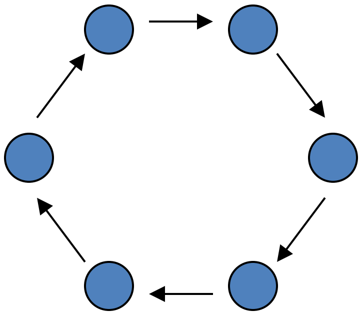
# Non-comparison-based algorithms

- Q: Can we improve on worst-case $O(n \log n)$ messages to elect a leader in a ring, if UIDs can be manipulated using arithmetic?
- Yes, easily!
- Consider case where:
  - $n$ is known,
  - Ring is unidirectional,
  - UIDs are positive integers, allowing arithmetic.



- Algorithm:
  - Phases $1,2,3, \ldots,$ each consisting of $n$ rounds
  - Phase $k$
    - Devoted to UID $k$.
    - If process has UID $k$, circulates it at beginning of phase $k$.
    - Others who receive it pass it on, then become passive (or halt).
- Elects min.

# Complexity bounds



- Communication:
  - Just $n$ (one-hop) messages
- Time:
  - $u_{min} n$
  - Practical only if the UIDs are small integers.

- Q: What if $n$ is unknown?
- Can still get $O(n)$ messages, though now the time is even worse: $O(2^{umin} n)$.
  - VariableSpeeds algorithm, Section 3.5.2.
  - Different UIDs travel around the ring at different speeds, smaller UIDs traveling faster.
  - UID $u$ moves one hop every $2^u$ rounds.
  - Smallest UID gets all the way around before next smallest has gone half-way, etc.

# Lower bound

- Q: Can we get smaller message complexity for comparison-based algorithms?
- $\Omega(n \log n)$ lower bound (next time).
- Assumptions
  - Comparison-based algorithm,
  - Deterministic,
  - Unique start state (except for UID).

# Comparison-based algorithms

- All decisions determined only by relative order of UIDs:
  - Identical start states, except for UID.
  - Manipulate UIDs only by copying, sending, receiving, and comparing them ($<, =, >$).
  - Can use results of comparisons to decide what to do:
    - State transition
    - What (if anything) to send to neighbors
    - Whether to elect self leader

# Lower bound proof: Overview

- For any $n$, there is a ring $R_n$ of size $n$ in which any leader election algorithm has:
  - $\Omega(n)$ "active" rounds (in which messages are sent).
  - $\Omega(n\ /\ i)$ messages sent in the $i^{th}$ active round.
  - Therefore, $\Omega(n \log n)$ messages total.
- Choose ring $R_n$ with a great deal of symmetry in ordering pattern of UIDs.
- Key lemma: Processes whose neighborhoods have the same ordering pattern act the same, until information from outside their neighborhoods reaches them.
  - Need many active rounds to break symmetry.
  - During those rounds, symmetric processes send together.
- Details next time (tricky, read ahead, Section 3.6).

# Next time…

- Lower bound on communication for comparison-based leader election algorithms in rings, in detail.
- Basic computational tasks in general synchronous networks:
  - Leader election, breadth-first search, shortest paths, broadcast and convergecast.
- Readings:
  - Sections 3.6, 4.1-4.3.