

6.852: Distributed Algorithms

Fall, 2009

Instructor: Nancy Lynch

Guest Instructor: Victor Luchangco

Distributed algorithms

- Algorithms that are supposed to work in distributed networks, or on multiprocessors.
- Accomplish tasks like:
 - Communication
 - Data management
 - Resource management
 - Synchronization
 - Consensus
- Must work in difficult settings:
 - Concurrent activity at many locations
 - Uncertainty of timing, order of events, inputs
 - Failure and recovery of machines/processors, of communication channels.
- So the algorithms can be complicated:
 - Hard to design
 - Hard to prove correct, analyze.

This course

- Theoretical, mathematical viewpoint
- Approach:
 - Define distributed computing environments.
 - Define abstract problems.
 - Describe algorithms that solve the problems.
 - Analyze complexity.
 - Identify inherent limitations, prove lower bounds and other impossibility results.
- Like sequential algorithms (6.046), but more complicated.

Distributed algorithms research

- Active area for 30+ years.
- PODC, DISC, SPAA, OPODIS; also ICDCS, STOC, FOCS,...
- Abstract problems derived from practice, in networking and multiprocessor programming.
- Static theory:
 - Assumes fixed network or shared-memory setting.
 - Set of participants, configuration, may be generally known.
 - In [\[Lynch text\]](#), [\[Attiya, Welch text\]](#), [\[Supplementary course papers\]](#)
- Dynamic theory:
 - Client/server, peer-to-peer, wireless, mobile ad hoc
 - Participants may join, leave, move.
 - Next year's special-topics course?
- Theory for modern multiprocessors:
 - Multicore processors
 - Transactional memory
 - New material for this term [\[Herlihy, Shavit book\]](#)
- All are active research areas!

Administrative info (Handout 1)

- People and places
- What the course is about
- Prerequisites
 - Automata:
 - Traditional courses (6.045, 6.840) study general theory of automata.
 - We use automata as tools, to model processes, channels, ...
 - Important for clear understanding, because the algorithms are complicated.
- Source material
- Course requirements
 - Readings
 - Problem sets
 - Given out every week, due every two weeks
 - Tempo language, for writing pseudocode
 - Code should (at least) syntax-check!
 - Grading

Topic overview (Handout 2)

- Variations in model assumptions:
 - IPC method: Shared memory, message-passing
 - Timing:
 - Synchronous (rounds)
 - Asynchronous (arbitrary speeds)
 - Partially synchronous (some timing assumptions, e.g., bounds on message delay, processor speeds, clock rates)
 - Failures:
 - Processor: Stopping, Byzantine, occasional total state corruption
 - Communication: Message loss, duplication; Channel failures, recovery
- Top-level organization is by the timing model.
- **Synchronous model:** Classes 1-6.
 - Basic, easy to program.
 - Not realistic, but sometimes can emulate in worse-behaved networks.
 - Impossibility results for synch networks carry over to worse networks.
- **Asynchronous:** Most of the rest of the course
 - Realistic, but hard to cope with.
- **Partially synchronous:** Last few classes.
 - Somewhere in between.

Detailed topic overview

- Synchronous networks:
 - Model
 - Leader election (symmetry-breaking)
 - Network searching, spanning trees
 - Processor failures: Stopping and Byzantine
 - Fault-tolerant consensus: Algorithms and lower bounds
 - Other problems: Commit, k-agreement
- Asynchronous model (I/O automata)
- Asynchronous networks, no failures:
 - Model
 - Leader election, network searching, spanning trees, revisited.
 - Synchronizers (used to run synchronous algorithms in asynchronous networks)
 - Logical time, replicated state machines.
 - Stable property detection (termination, deadlock, snapshots).

More topics

- Asynchronous shared-memory systems, no failures:
 - Model
 - Mutual exclusion algorithms and lower bounds
 - Practical mutual exclusion algorithms
 - Resource allocation, Dining Philosophers
- Asynchronous shared-memory, with failures
 - Impossibility of consensus
 - Atomic (linearizable) objects, atomic read/write objects, atomic snapshots
 - Wait-free computability; wait-free consensus; wait-free vs. f -fault-tolerant objects
- Shared-memory multiprocessor programming
 - Contention, caching, locality
 - Reader/writer locks
 - List algorithms: locking algorithms, optimistic algorithms, lock-free algorithms
 - Transactional memory

Still more topics

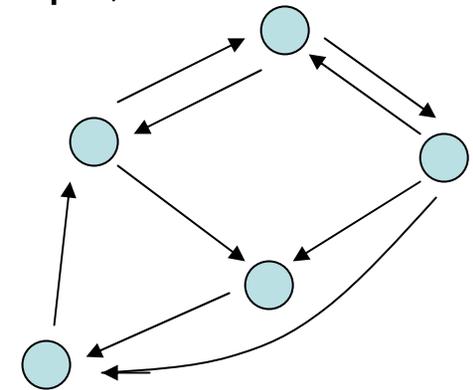
- Asynchronous networks, with failures
 - Asynchronous networks vs. asynchronous shared-memory
 - Impossibility of consensus, revisited
 - Failure detectors and consensus
 - Paxos consensus algorithm
- Self-stabilizing algorithms
- Partially-synchronous systems
 - Models
 - Mutual exclusion, consensus
 - Clock synchronization

Now start the course...

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

Synchronous network model

- Processes (or processors) at nodes of a network digraph, communicate using messages.
- Digraph: $G = (V, E)$, $n = |V|$
 - out-nbrs_i, in-nbrs_i
 - distance(i,j), number of hops on shortest path
 - diam = max_{i,j} 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 × out-nbrs_i → M ∪ { \perp }
 - trans_i : states_i × vectors (indexed by in-nbrs_i) of M ∪ { \perp } → states_i
- Executes in rounds:
 - Apply msgs_i to determine messages to send,
 - Send and collect messages,
 - Apply trans_i to determine new state.



Remarks

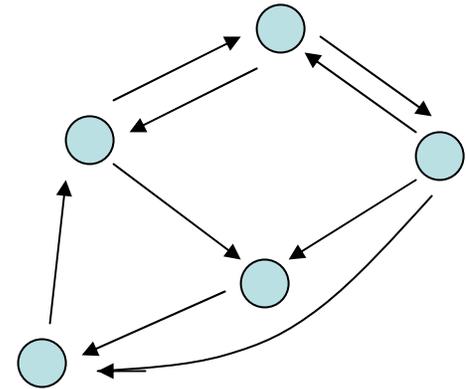
- No restriction on amount of local computation.
- Deterministic (a simplification).
- Can define “halting states”, but not used as accepting states as in traditional automata theory.
- Later, we will consider some complications:
 - Variable start times
 - Failures
 - Random choices
- Inputs and outputs: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

Executions

- An execution is a mathematical object 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: $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \dots$
 - C's are state assignments.
 - M's are messages sent.
 - N's are messages received. } Message assignments
 - Infinite sequence (but could consider finite prefixes).

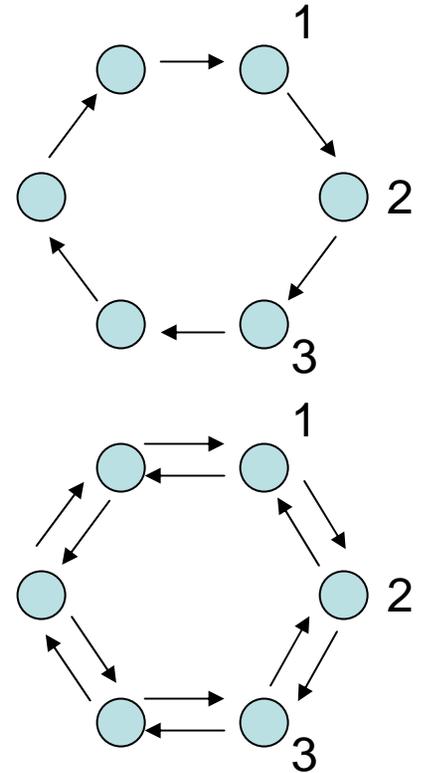
Leader election

- Network of processes.
- Want to distinguish exactly one, as the “leader”.
- Eventually, exactly one process should output “leader” (set special status variable to “leader”).
- Motivation: Leader can take charge of:
 - Communication
 - Coordinating data processing (e.g., in commit protocols)
 - Allocating resources
 - Scheduling tasks
 - Coordinating consensus protocols
 - ...



Simple case: Ring network

- Variations:
 - Unidirectional or bidirectional
 - Ring size n known or unknown
- Numbered clockwise
- Processes don't know the numbers; know neighbors as "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 best 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 (could choose same one for all).
- Then there is exactly one execution
 $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \dots$
- Prove by induction on the number r of completed rounds that all processes are in identical states after r rounds.
 - Generate same messages, to corresponding neighbors.
 - Receive same messages.
 - Make the same state changes.
- Since the algorithm solves the leader election problem, someone eventually gets elected.
- Then everyone gets elected, contradiction.

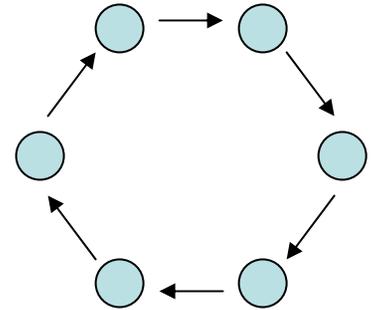
So we need something more...

- 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.:
 - Arbitrary 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 msg, to be relayed step-by-step around the ring.
 - When process receives a UID, compares with its own.
 - If incoming is:
 - Bigger, pass it on.
 - Smaller, discard.
 - Equal, process declares itself the leader.
 - Elects process with the largest UID.



In terms of our formal model:

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

```
if incoming = v, a UID, then
  case
    v > u: send := v
    v = u: status := leader
    v < u: no-op
  endcase
```
 - Entire block of code is treated as atomic.

Correctness proof

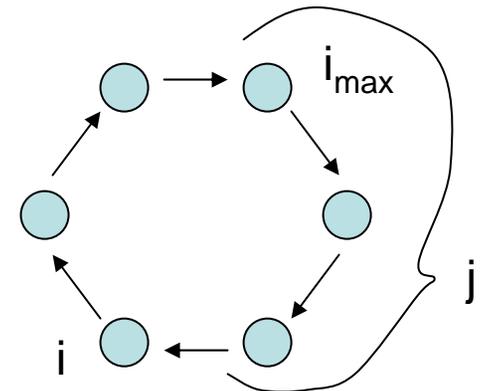
- Prove exactly one process ever gets elected leader.
- More strongly:
 - Let i_{\max} be the process with the max UID, u_{\max} .
 - Prove:
 - i_{\max} outputs “leader” by end of round n .
 - No other process ever outputs “leader”.

i_{\max} outputs “leader” after n rounds

- Prove by induction on number of rounds.
- But need to strengthen, to say what happens after r rounds, $0 \leq r \leq n$.
- **Lemma 2:** For $0 \leq r \leq n-1$, after r rounds, **send** at process $(i_{\max} + r) \bmod n$ contains u_{\max} .
- That is, u_{\max} is making its way around the ring.
- **Proof of Lemma 2:**
 - Induction on r .
 - Base: Check the initialization.
 - Inductive step: Key fact: 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 the correct output happens.
- Key fact: i_{\max} uses arrival of u_{\max} as signal to set 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 no one has already).
- Use Lemma 3 to show that no one except i_{\max} ever receives its own UID, so never 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 $i_{\max} = \text{leader}$.
- Invariants are usually proved by induction on the number of steps in an execution.
 - May need to strengthen, to prove 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 you had to.
 - Try some examples in detail.
- Invariant proofs may seem like overkill here, 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 provide the main method for proving properties of distributed algorithms.**

Complexity bounds

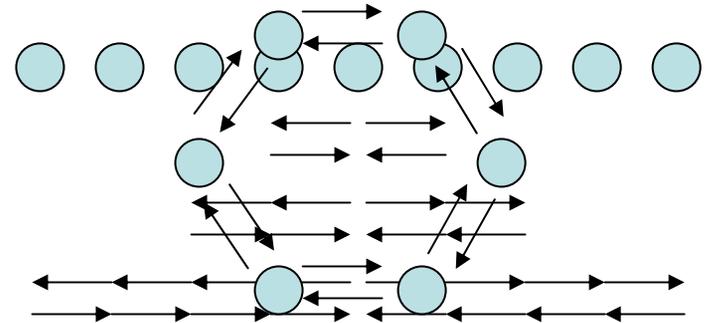
- What to measure?
 - Time = number of rounds until “leader”: n
 - Communication = number of single-hop messages: $\leq n^2$
- Variations:
 - Non-leaders announce “non-leader”:
 - Any process announces “non-leader” as soon as it sees a UID higher than its own.
 - No extra costs.
 - Everyone announces 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.
 - Time: $\leq 2n$
 - Communication: $\leq n^2 + n$
 - Doesn't rely on synchrony or knowledge of n .

Halting

- 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 (in basic problem) after it sees and relays some UID larger than its own?
- A: No---it still has to stay alive to relay 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: $0, 1, 2, \dots$
- Phase 0: All send messages both ways, $\leq 4n$ 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 are 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 is at most

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

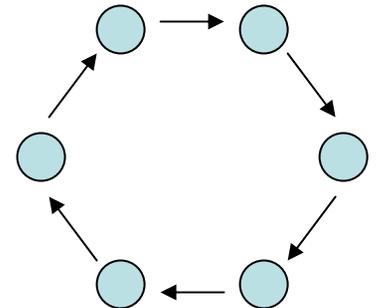
Out and back,
both directions

New distance

- So total communication is at most $8n(1 + \lceil \log n \rceil) = O(n \log n)$

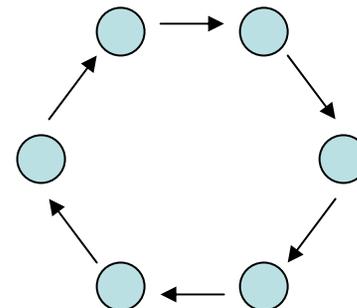
Non-comparison-based algorithms

- 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?
- A: Yes, easily!
- Consider case where:
 - n is known
 - Ring is unidirectional
 - UIDs are positive integers, allowing arithmetic.
- Algorithm:
 - Phases 1,2,3,...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:
 - $u_{\min} n$
 - Not practical, unless the UIDs are small integers.



- Q: What if n is unknown?
- A: Can still get $O(n)$ messages, though now the time is even worse: $O(2^{u_{\min}} n)$.
 - **VariableSpeeds** algorithm, Section 3.5.2
 - Different UIDs travel around the ring at different speeds, smaller UIDs traveling faster
 - UID u moves 1 hop every 2^u rounds.
 - Smallest UID gets all the way around before next smallest has gone half-way, etc.

Next time...

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

MIT OpenCourseWare
<http://ocw.mit.edu>

6.852J / 18.437J Distributed Algorithms
Fall 2009

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.