Distributed Algorithms, 6.852

People: me, Ziv Bar-Joseph, Joanne Tabbot

Subject matter: Algorithms that work in distributed networks
Algorithms that are intended to work in distributed networks.
Accomplish such tasks as:
- Communication
- Database maintenance & access
- Resource allocation
- Consensus

Must work in difficult setting:
- Concurrent activity at many places.
- Uncertainty of timing, inputs.
- Failure & recovery: of machines, of comm. channels

So, can be complicated.

Hard to design.

Difficult to reason about, analyze.

This course: Approaches the subject matter from a theoretical,
mathematical viewpoint.

Focuses on:
- Defining abstract problems
- Describing (abstractly) algorithms that solve the problems
- Carrying out complexity analysis
- Identifying inherent limitations, proving them
  with lower bounds & other impossibility results

Analogous to study of sequential algorithms, as in CLR.

But in a more difficult setting.
Distributed computing is now a well-developed research area active for past 20+ years. Conferences: PODC, DISC, tracks of other distributed computing conferences (e.g., ICDCS).

Problems & algorithms are derived from practice, though often abstracted quite a bit. Most of the theory that has been developed (excepting cryptography) assumes a fixed network, known set of processes. New distributed settings are less well-behaved than this: More "dynamic" (participants may join & leave, as well as fail & recover. Mobility possible. ) Think peer-to-peer, as extreme example.

The theory of algorithms & inherent limitations is not as well developed in these areas. Opportunities for new research in extending basic theory to these newer, more complicated settings.

This course will introduces these topics by including supplementary readings & presentations.
Administrative Info

1. What this course is about

2. People & places

3. Prerequisites
   - Because the algorithms can be complicated, need to define formal models for everything.
   - Interacting automata.
   - 6.045 gives general ideas about automata, but we won't use any of the theorems.

4. Source material

5. Class conduct
   - Work generally from book.
   - You can read the book. Actually, best if you've read the relevant sections ahead of time.
   - In class, I won't present all the details, but can concentrate more on:
     - Highlight, perspective
     - Clarifying technical issues
     - Discussion
     - (I like discussion better than just lecturing.)

5. Course Requirements
   5.1 Problem sets
   5.2 Grading
   5.3 Reading, presentations
Other remarks

Bring book, when there's significant code to point to.
I'll try to warn you.

JOA language/toolset
JOA automata are a basic model for asynchronous systems and their components.
In particular, distributed algorithms.
JOA language is a formal language for describing JOA automata programming/modeling
Based on the pseudocode in my book.
Has parser, connection to theorem prover
simulator
Other stuff in progress (research projects).
You might find this useful. (Will provide pointers, access &
there's a demand)
to anyone who wants them.)
Specific topic overview

Many variations in model assumption:
- IPC method: sh. memory, message-passing, RPC (RM I)
- Timing assumptions: synch, asynch (arbitrary speeds), partial synch (uses some timing assumptions)
- Failures: processor, stopping, Byzantine
- Communication:
  - Loss, duplication
  - Channel failure (recovery)
  - Network partitions

Not obvious how best to organize it all.
Factors that seem to make the most difference is the timing model.
So that's how it's organized, at top level.

Start with synch: Lock step
- Classes 1-6
  - Not realistic, but basic, & sometimes can emulate in less well-behaved network.
  - Besides, if something is impossible in synch network, then also impossible in less well-behaved networks.

Then asynch (most of course)
- Realistic, but hard to cope with.

Finally, partially synch
- Assume something about timing (e.g. bounds on msg. delivery, proc. speed)
  - But not complete lock-step synchrony.

In each case, start with basic math model, then go on to specific problems & algs.
Specific problems/alg

Sych: Model
- Leader election (synchrony checking)
- Network searching, spanning trees
- Fault-tolerant consensus problems (Byzantine agreement)
- Commit

Async: Model (I/O automate)

Async message-passing systems: Model
- Leader election, searching, spanning trees
- Syndromes (how to run synchronous alg in async networks)

Async: shared-memory systems

Model
- Mutual exclusion, resource allocation
- Consensus - fundamental impossibility result, FLP
- Atomic objects: Looser version of shared memory
  - Slightly which can be implemented in dist. networks

Which leads us to: methods of implementing async
- Sh. mem alg in async networks

Back to async: Networks

+ vice versa.

So, e.g., the basic imposs result carried over from sh. mem to network model.

Back to async networks

- Failure detectors
- Consensus (atomic broadcast) not in book
- Logical time, replicated state machines
- Global snapshot, distributed termination, deadlock det.
- Reliable comm. using unreliable channels
Models

Basic problems revisited: mutex, consensus

Then student presentations ...

Questions?
Now start the actual course

Rest of today: Syrki model

Simple leader election (symmetry breaking) problem

Reading: Ch. 2 + 3 (skip 3.7)

Next week: Ch. 3, each step 4.1 - 4.3

Model: Simple, but introduces some of the complexities of real dist. alg.

Activity at many locations.

Incl. inputs

Coping with failures.

Processes (or processes) at nodes of a network digraph, communicating using messages.

Digraph: \( G = (V, E) \)

\( m = |V| \)

out-mbs, in-mbs:

distance \((i,j)\)

diam = max distance \((i,j)\)

For each index \(i\), a process consisting of:

- states: \( \text{set, not nec. finite} \)
- start: \( \text{nonempty subset} \)
- msgs: \( \text{states} \times \text{out-mbs} \rightarrow MV \times \text{null} \)
- trans: \( \text{states} \times \text{routes (indexed by in-mbs)} \rightarrow \text{MV} \times \text{null} \)

Executes in rounds: msg gen fn, collect msgs, apply trans fn
Remarks: The reduction in amount of computation
Deterministic (a simplification - later, nondet will be
very important)
Can define halting states, but not used as accepting states as
in automata theory.
Later: add more complications like variable start times
failures
Inputs & outputs: Can encode in states, e.g. in special input &
output variables.
Executions: Formal notion needed to describe how a system alg.
operates.
Instead of talking about "what happens" when the alg. runs,
define a formal object known as an "execution".
(a sequence of states & other stuff).
Reason carefully about what appears in various places in
the sequence.

Formally, p. 20
State assignment: mapping from proc. indices to states
msg assignment: mapping from ordered pairs of proc indices
to \( V \) & \( \Sigma \)

\[ \text{Exec: } C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots \]

Infinite seq. C's are state assignments
M's are msgs sent, \( \{ \) could differ, lost msgs.
N's \( \{ \) received

means \( x + z \) look identical to proc. i.
\( \{ \) same seq of states of i
\( \{ \) outgoing msgs from i
\( \{ \) incoming msgs to i

used in some important results.
Leader Election

In general, have network of procs. Initially not distinguished. Want to distinguish one, as leader.

Motivation: Leader can take charge of communication of maintaining states of allocating resources, scheduling tasks, etc. serve as coordinator of database queries, e.g., in commit protocols.

Special case: Ring network simpler, but has some of the key difficulties. Understand this case first.

Assume

1. Clockwise numbering, but processes don’t know these.
2. Just know rings by names “clockwise” or “counterclockwise”.
3. Eventually, exactly one proc. should “output leader” (set special status var to “leader”)

Impossibility result: Theorem 1:

Suppose all procs are identical. Then impossible to elect a leader, even in the best case.

Proof: By contradiction. Assume an algorithm.

Assume wlog that each process has exactly one initial state (or could choose same one for all).

Then exactly 1 execution.
Show by induction on \( n \) of rounds that all proc are in identical states after \( n \) rounds.

(Same msg, same state transition)

Then if ever any process reaches status = leader, all will (+ at the same time). Impossible.

But to solve the problem, someone must...

So, to even get off the ground, must have some way of distinguishing the processes.

So assume proc have UIDs, which they "know" (have access to, can use).

UIDs can be elements of various data types, with different allowed ops.

E.g. \( \{\text{to set with } (<, >, =), \text{ companions}
\}
\)

Integers with full arithmetic

A basic algorithm: (Lehman, Chang, Roberts)

- Undirectional (clockwise)
- Proc don't know \( m \)
- UIDs - companions only

Informally: Each proc sends its identifier in a msg, to pass step-by-step around the ring.

When proc receives incoming UID, compares with its own.
- If incoming is smaller, pass it on
- If equal, proc knows it's the leader & outputs.
- So, proc with the largest UID is elected leader.

Informally: Book p. 28FF shows how to express in terms of the formal model.
M: msg alphabet = set of UIDs

status: This is where all the info that must be remembered is kept. Each state consists of values for the following program variables: (states in automaton modeling systems in terms of variables, components generally have structure)
u, holds its own UID
send, a UID or null, initially its own UID
status ∈ {unknown, leader}, init "unknown"

start: defined by the initializations
msg: send whatever is in the send able to clockwise successor
trans: defined by pseudocode, p. 28 bottom
a case statement, branching on the id comparison (of incoming msg (if any) with u
send := null (cleans out old msg sent)

if incoming = v then
  case
  v > u: send := v
  v = u: status := leader
  v < u: no-op
end

Program translates directly into model.
Note block of code atomic

Correctness proof
Can do this, prove because modeled formally.
Here, prove exactly one process becomes leader.
Let $i_{\text{max}} = \text{proc with max UID}$, $\mu_{\text{max}} = \text{the max UID}$

**Proof.**
1. $i_{\text{max}}$ outputs "leader" by end of round $n$.
2. No other process ever outputs "leader".

1. Show after $n$ rounds, status = leader.

**Proof by induction on # of rounds.** By strengthening (generalizing)
this to talk about what’s true after $r$ rounds, $0 \leq r \leq n$.

**Lemma 2:** For $0 \leq r \leq m - 1$, after $r$ rounds,

$$\text{send}(i_{\text{max}} + r) = \mu_{\text{max}}$$

That is, $i_{\text{max}}$ appears to be making its way around the ring...

**Pf:** Induction on $r$.

- **Base:** By initialization.

- **Inductive Step:** Key is that everyone else lets $i_{\text{max}}$ pass through.

Having proved Lemma 2, use $r = m - 1$ statement and one more argument
about a single round to show the main claim.

**Key fact here:** $i_{\text{max}}$ uses receipt of $\mu_{\text{max}}$ as signal to set

status = leader.

2. Uniqueness: no one except $i_{\text{max}}$ ever outputs "leader".

Again, need stronger claim, about arbitrary $r$. 
Lemma 3: For any $n \geq 0$, after $n$ rounds,
if $i \neq \text{max} + 1 \in \{\text{max}, i\}$ then send $\neq n$.

Says $u_i$ can't get past $\text{max}$ in the cycle.

Proof by induction in $n$.
Key step: While $\text{max}$ throws away $u_i$ (if no one has already).

Lemma 3 can be used to argue that no one except $\text{max}$ can receive its own UID, so only $\text{max}$ can elect self.

Invariant proofs: Lemmas 2 + 3 are examples of invariants:
- Properties true in all reachable states (assuming $n$ is in the state)
  
  (Actually, $n$ is "if $n = i$ then status $i = \text{leader}$"

Invariants are usually proved by induction on # of steps in exec.
May need to strengthen invariant in order to prove.

I won't do detailed proofs of invariants in class, but I expect you to know how to do this.

Book does some:
- will give sketches of key steps as above.
- I expect you know how to fill in the details of such a sketch to turn it into a complete proof. (if you're not sure, try filling in)
**Complexity:**

What to measure?  
\[ \# \text{ of rounds until "leader"} \leq n \]  
true measure

\[ \# \text{ of single-hop messages} \leq n^2 \]  
communication measure

**Variations:**

\underline{Halting}: Add halting states (special states, all transitions are self-loops, no msgs generated)

Modify algorithm: Leader circulates report msg

Any process receiving report passes it on + halts.

2 \( n \) rounds total

\[ \leq n^2 + n \text{ msgs.} \quad O(n^2) \]

\underline{Non-leader announcements}:

all \( \neq \text{leader} \) could say this, e.g. upon receiving report.
(Could also announce who the leader is.)

Actually, non-leader earlier, when receive "ID larger than its own."
Reduction in message complexity:
\[ O(n \log n) \] rather than \[ O(n^2) \] (Hirschberg–Simclair)

This uses a successive doubling trick. Appears in other dist. algs., esp. where don’t know size of network.

Assumptions:
- Bidirectional comm.
- Walk even if msg size unknown
- UIDs with comparisons only.

Inbound: Send id in both dirs, to successively greater distances (double each time)

Going outbound, token is swallowed up if it reaches a node whose own UID is bigger.

Going inbound, everyone passes everyone on.

If you get your own msg in outbound dir, elect self leader.

This describes global behavior.

So for the model, must write in terms of local process descriptions. See p. 33.

Carefulness: LTRR. Can do with invariants, as for LCR, but more complicated (more bookkeeping).

Complexity:
- Time: Worse than LCR, but still \( O(n) \).
  - Time for each phase is \( 2^n \) previous, so dominated by time for last complete phase (geometric series).
  - But last phase is \( O(n) \), so total is also.
  - (More precisely, \( \leq 3n \) if \( n \) is power of 2, \( \leq 5n \) in general.)
Msg cost: $O(n \log n)$

Key idea: $O(\log n)$ phases, number 0, 1, 2, ...

Phase 0: All send both ways $\leq 4m$ msgs if all go out & return (though they won't)

Phase k 70: Within any block of $2^{k-1}+1$ consecutive proc, 
at most one is still alive at leg of phase k (others swallowed up by earlier phases)

So, at most

$$\left\lfloor \frac{n}{2^{k-1}+1} \right\rfloor$$

is the start phase k by sending token.

Total # of msgs at phase k, therefore, is

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

Total # of phases $\leq 1 + \left\lceil \log n \right\rceil$ (successive doubling)

So, total comm $\leq 8m \left( 1 + \left\lceil \log n \right\rceil \right) = O(m \log n)$