Distributed Algorithms, 6.852

People: me

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 CLRS.

But in a more difficult setting.
Out. algos 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 (including much of this book) 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 for these areas.
Opportunities for new research in extending basic theory to these newer, more complicated settings.

This course will introduce these topics by including supplementary readings & presentations.
Administrative Info
From handout 1.

1. What this course is about

2. People + places

3. Notes
   ① Because the algorithms can be complicated, need to define formal models for everything.

   Interacting automata.

   0.045 gives general ideas about automata, but we won't use any of the theorems.

4. Course material

3. 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:
   - Highlights, 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 presentation
other remarks

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

I/O language/toolset

I/O automata are a basic model for asynchronous systems and their components.
In particular, distributed algorithms.
I/O language is a formal language for describing I/O 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 if 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: sync, async, (arbitrary speeds), partial
- sync: (uses some timing assumptions)
- failures: processor, stopping, Byzantine
- communication: loss, duplication, channel failure, recovery
- network partitions

Not obvious how best to organize it all.

Factor that seems to make the most difference is the timing model.
So that's how it's organized, at top level.

Start with sync:

- classes 1-6: not realistic, but basic, + sometimes can emulate in less well-behaved network.
- Besides, if something is impossible in sync network, then also impossible in less well-behaved networks.

Then async: (most of course)

- realistic, but hard to cope with.

Finally, partially sync:

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 + algo.
Specific problems / algos

**Synchronous Model**
- Leader election, symmetry breaking
- Network searching, spanning trees
- Fault-tolerant consensus problems
  - (Byzantine agreement)
- Commit

**Asynchronous Model**
- (Turing automata)

**Asynchronous message-passing systems**
- Leader election, searching, spanning trees
- Synchronizers (how to run synchronous algo in asynchronous networks)

**Asynchronous shared-memory systems**
- Mutual exclusion, resource allocation
- Consensus - fundamental impossibility result, T.C.P.
- Atomic objects: looser version of shared memory
- Slightly which can be implemented in dist. networks

Which leads us to: methods of implementing & asynchronous shared mem. algos in asynchronous networks.

Back to asynchronous networks

Back to asynchronous networks

Basic abstractions
- Consensus & atomic broadcast
- Logical time, replicated state machines
- Global snapshots, distributed termination, deadlock det.
- Reliable comm. using unreliable channels
P. Synch Models

Basic problems revisited: mutex, consensus

Ten student presentations...

Questions?
Now start the actual course.

Rest of today: Byzaki model

Simple leader election (symmetry breaking) problem

Reading: Ch. 2 + 3 (skip 3.7)

Next two: Self-study pp. 4.1-4.3

Model: Simple, but introduces some of the complexities of real distributed activity at many locations.

Incl. inputs

Coping with failures.

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

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

\[ m = |V| \]

\[ \text{out-mlbs}_i, \text{in-mlbs}_i \]

\[ \text{distance}(i,j) \]

\[ \text{diam} = \max_{i,j} \text{distance}(i,j) \]

\[ M, \text{msg alphabet + null placeholder (null \in M)} \]

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

\[ \text{trans}_i \]: \[ \text{states}_i \times \text{out-mlbs}_i \rightarrow M \cup \{ \text{null} \} \]

\[ \text{msg}_i \]: \[ \text{states}_i \times \text{states}_j \rightarrow M \cup \{ \text{null} \} \]

\[ \text{executes in rounds}: \text{msg gen fn, collect msgs, apply trans fn} \]
Remarks: No restriction on 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

Inputs + outputs: Can encode in states, e.g. in special input + output variables.

Executions: Formal notion needed to describe how a synch alg operates.

Instead of talking about "what happens" when the algo 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 reduced pairs of proc. indices to N \times \mathbb{N}

Ex: C_0, M_1, N_1, C_1, M_2, N_2, C_2, ...

Infinite seq. C_i's are state assignments
M_i's are msg sent (could differ, lost msgs)
N_i's "is received"

\[ L \equiv L' \]

means L = L' look identical to proc. i

(same seq of states of i

outgoing msgs from i

incoming msgs to i)

used in some proofs, results.
Leader Election

In general, have network of processes, initially not distinguished. Want to distinguish one as leader.

Motivation: Leader can take charge of communication, maintaining state of allocating resources, scheduling tasks, etc. serve as coordinator of database operations 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 ranks by names "clockwise" or "counterclockwise".
3. Eventually, exactly one proc. should "output "leader" (set special status run to "leader")

Impossibility result: Shown!

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

Proof: By contradiction. Assume an algorithm.

Assume log that each process has exactly 1 start state.

Then exactly 1 execution.
Show by induction on \( n \) of rounds that all \( n \) processes are in identical states after \( n \) rounds.
(Same map, same state transitions)
Then if ever any process reaches \( \text{status} = \text{leader} \), 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 every process has \( \text{UIDs} \), which they "know" (have access to, can see).

\( \text{UIDs} \) can be elements of various data types, with different allowed ops.
E.g. \( \{ \text{t.o. set with } <, >, = \} \) comparisons
\( \text{integers with full arithmetic} \)

\underline{A basic algorithm:}
\[
\begin{align*}
\text{unidirectional (clockwise)} \\
\text{procs don't know } n \\
\text{UIDs - comparisons only}
\end{align*}
\]

Informally: Each process sends its identifier in a ring, to pass step-by-step around the ring.
When proc. receives incoming \( \text{UID} \), compares with its own.
If incoming is (bigger, pass it on)
If smaller, discard
If equal, process knows it's the leader \& outputs.
So, process with the largest \( \text{UID} \) is elected leader.

Formally: Book p. 28 FF shows how to express in terms of the formal model.
M: msg alphabet = set of VIDs
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 automata modeling systems + components generally have structure, in terms of variables.)

u, holds its own VID
send, a VID or null, initially its own VID
status ∈ {unknown, leader, init, "unknown"}

start: defined by the initializations
msgs: send whatever is in the send slot to clockwise successor
trans: Defined by pseudocode, p. 28 onward.
A case statement, branching on the id comparison (if 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: ns-op
end

Program translates directly into model.
Note block of code atomic

Correctness proof
Can do this, since modeled formally.
Here, prove exactly one process becomes leader.
Let \( i_{\text{max}} \) peer with \( \text{max VID} \), \( n_{\text{max}} = \text{the max VID} \).

**Proof:**
1. \( i_{\text{max}} \) outputs "leader" by end of round \( n \).
2. No other process can output "leader".

1. Show after \( n \) rounds, \( \text{status}_{i_{\text{max}}} = \text{leader} \).

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

**Lemma 2:** For all \( 0 \leq n \leq n - 1 \), after \( n \) rounds,

\[
\frac{(\text{max} + n)}{\text{max}} = n
\]

That is, \( x_{\text{max}} \) appears to be making its way around the ring.

**Of:** Induction on \( n \).

- **Base:** By initialization
- **Inductive step:** Key is that everyone else lets \( i_{\text{max}} \) pass through.

Having proved Lemma 2, use \( n = n - 1 \) statement and one more argument about a single round to show the main claim.

**Key fact here:** \( i_{\text{max}} \) uses receipt of \( i_{\text{max}} \) as signal to set \( \text{status} := \text{leader} \).

2. All uniquenesses - no one except \( i_{\text{max}} \) ever outputs "leader".

Again, need stronger claim, about arbitrary \( n \).
Lemma 3: For any $n \geq 0$, after $n$ rounds, if $i \neq \text{max} \quad \forall j \in [\text{max} + 1, n] \quad \text{then send} \quad \#j$.

Proof by induction on $n$.

Key step: When $\text{max}$ 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 and 3 are examples of invariants = properties true in all reachable states (assuming $n$ is in the state)

(Actually, $n$ is "if $r = n$ then status$_{\text{max}}$ = leader")

Invariants are usually proved by induction on # of steps in race.

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.

Back two points:

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 details for some example proofs.)
Complexity:
What to measure?

- \# of rounds until "leader" ≥ n
- \# of single-hop messages
  communication measure
  \leq n^2

Variation:

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.

2n rounds total
≤ n^2 + n msgs. \(O(n^2)\)

Non-leader announcements:

all non-leader could say this, e.g. upon receiving report.
(Could also announce who the leader is.)

Actually, non-leader declares when receive UID larger than its own.
Reduce message complexity to $O(n \log n)$ rather than $O(n^2)$ (Hirschberg-Sinclair). Uses a successive doubling trick. Appears in other dist. algos, esp. where don't know size of network.

Assumptions:
- Bidirectional comm.
- Works even if ring size unknown
- UIDs with comparisons only

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

- Going outbound, token is swallowed up if it reaches a node where own UID is bigger
- Going inbound, everyone passes everything on
- If you get your own msg in outbound dir, elect self leader.

This describes global behavior. To fit the model, must write in terms of local process descriptions, see p. 33.

Constraints: LTTR. 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 $2^p$ previous, so dominated by time for least complete phase (geometric series)
- But last phase is $O(n)$, so total is also
- (More precisely, $\leq 3n$ if $m$ is power of 2)
- $\leq 5n$ in general
Msg cost: $O(n \log m)$

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

Phase 0: all send both msgs $\leq 4n$ msgs if all go out and return (though they won't)

Phase $k \geq 0$: Within any block of $2^{k-1}+1$ consecutive phases, at most one is still alive at beg of phase $k$ (others swallowed up by earlier phases)

So, at most

$$\frac{m}{2^{k-1}+1}$$

Start phase $k$ by sending token.

Total # of msgs at phase $k$, therefore, is

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

Output & back, distance both dir.

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

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