6852
Fall, 01

Bring: { Handouts 1-4
chalk
Books - mine & Jennifer's

Distributed Algorithms, 6.852
People: me, Ziv Bar-Joseph, Joanne Talbot

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.

Dist. algs is now a <u>well-developed research area</u>
<u>Active</u> for past 20+ years
<u>Conferences</u>: PODC, DISC, tracks of other distributed computing conferences (e.g., ICDCS).

Problems + algorithms are <u>derived from practice</u>, though often abstracted quite a lot.

<u>Most of the theory</u> that is been developed ~~(+ everything in my book)~~ + book mainly covers this
assumes a <u>fixed network</u>, unchanging, usually <u>known set of processes</u>.

<u>New</u> distributed settings are <u>less well behaved</u> than this:
<u>More "dynamic"</u>: { participants may <u>join + leave</u>, as well as fail + recover.
<u>mobility</u> possible

(think peer-to-peer, as extreme example)

The theory of algorithms + inherent limitations is <u>not as well developed</u> for these areas.

<u>Opportunities for new research</u> 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. Prereqs

⋮ ① Because the settings + 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

② 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, presentations

⋮

Other remarks

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

IOA language / toolset

I/O automata are a basic model for asynchronous systems + their components.
In particular, distributed algorithms.
IOA 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.)

from handout 2

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: processes: 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 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
~~Sim~~ 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 / algs

Synch: Model
Leader election (symmetry breaking)
Network searching, spanning trees
Fault-tolerant consensus problems. (Byzantine agreement, commit)

Asynch: Model (I/O automata)

Asynch message-passing systems: Model
Leader election, searching, spanning trees
Synchronizers (how to run synchronous algs in asynch networks)

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

Which leads us to: methods of implementing ~~sh~~ asynch sh. mem algs in asynch networks.

~~Back to asynch networks~~
& vice versa.

So, e.g., the basic impos result carries over from sh. mem to network model.

Back to asynch networks
Failure detectors
Consensus & atomic bcast } not in book
Logical time, replicated state machines
Global snapshots, distributed termination, deadlock det.
Reliable comm. using unreliable channels.

P Synch: Models
Basic problems revisited: mutex, consensus

Then student presentations ...

Questions?

Now start the actual course

Rest of today: Synch model
Simple leader election (symmetry breaking) problem

Reading: Ch. 2 + 3 (skip 3.7)

~~Model~~:

Next time: ~~Ch 4, can skip 4.3~~ 4.1-4.3

Model: Simple, but introduces some of the complexities of real dist. algs:
activity at many locations.
incl. inputs
Coping with failures.

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

Digraph: $G = (V, E)$
$n = |V|$
$out\text{-}nbs_i$, $in\text{-}nbs_i$
$distance(i,j)$
$diam = \max_{i,j} distance(i,j)$

M, msg alphabet + null placeholder ($null \notin M$)
$\perp$
For each index i, a process, consisting of:
$states_i$ (set, not nec. finite)
$start_i$ (nonempty subset)
$msgs_i$: $states_i \times out\text{-}nbs_i \to M \cup \{null\}$
$trans_i$: $states_i \times$ vectors (indexed by $in\text{-}nbs_i$) of $M \cup \{null\}$
$\to states_i$

Executes in rounds: msg gen fn, collect msgs, apply trans. fn

Remarks: No restriction on amt 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 $\begin{cases} \text{variable start times} \\ \text{failures} \end{cases}$

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 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.

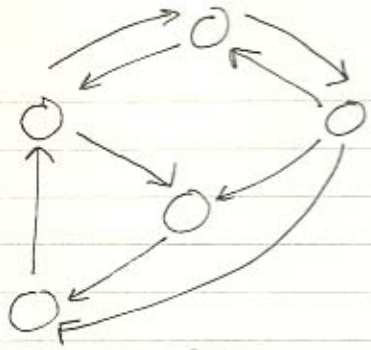
Formally, p. 20
state assignment: mapping from proc. indices to states
msg assignment: mapping from ordered pairs of proc indices to $M \cup \{\perp\}$

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
N's " " received } could differ, lost msgs.

$\boxed{\alpha \underset{i}{\sim} \alpha'}$ means α & α' look identical to proc. i.
(same seq of $\begin{cases} \text{states of } i \\ \text{outgoing msgs from } i \\ \text{incoming msgs to } i \end{cases}$)

Used in some impos. results.

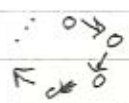


Leader Election

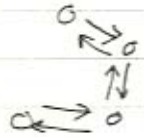
In general, have network of procs.
Initially not distinguished.
Want to distinguish one, as leader.

Motivation: Leader can
- take charge
  - of communication
  - ~~of managing data~~
  - of allocating resources
  - scheduling tasks, etc.
- serve as coordinator of database operations e.g. in commit protocols

Special case: Ring network



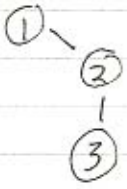
uni or



bi

Simpler, but has some of the key difficulties
Understand this case first.

Assume



clockwise numbering, but processes don't know those.

Just know nbs by names "clockwise" or "counterclockwise".
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,
(bidirectional comm
ring size n known to all)

Proof: By contrad. assume an algorithm.
Assume wlog that each process has exactly 1 start state.
(or could choose same one for all).

Then exactly 1 execution.

Show by <u>induction on # r of rounds</u> that all procs are in identical states after r rounds.
(Same msgs, same state transitions)
Then if ever any process reaches status = leader, all will (+ at the same time). Impossible
~~Contra~~
But to solve the problem, someone must ... □

So, to even get off the ground, must have some way of distinguishing the processes.
So assume procs have UIDs, which they "know" (have access to, can use).
UIDs can be elements of various <u>data types</u>, with different allowed ops.
E.g. { t.o. set with (<, >, =) comparisons
{ integers with full arithmetic

<u>a basic algorithm</u>: (LeLann, Chang, Roberts)

[unidirectional (clockwise)
[procs don't know n
[UIDs - comparisons only

<u>Informally</u>: Each process 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 { bigger, pass it on
{ smaller, discard
{ equal, process knows it's the leader + outputs.
So, process with the largest UID is elected leader.
<u>Formally</u>: Book p. 28FF shows how to express in terms of the formal model. :

M: msg alphabet = set of UIDs

states$_i$: 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 UID

send, a UID or null, initially its own UID

status $\in$ {unknown, leader}, init "unknown".

start$_i$: defined by the initializations

msgs$_i$: send whatever is in the send vble to clockwise successor

trans$_i$: Defined by pseudocode, p. 28 bottom.
a case statement, branching on the id comparison (of incoming msg (if any) with u

```
send := null        (clears 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, ~~with~~ because modeled formally.
Here, prove exactly one process becomes leader.

Let $\underline{i_{max}}$ = proc with max UID, $\underline{u_{max}}$ = the max UID

<u>Prove</u>: 1. i_{max} outputs "leader" by end of round n

2. No other process ever outputs "leader".

---

1. Show after n rounds, $status_{i_{max}}$ = leader.

Prove by induction on # of rounds, by strengthening (generalizing this to talk about what's true after r rounds, $0 \le r \le n$.

<u>Lemma 2</u>: $\forall r$ $0 \le r \le n-1$, after r rounds,

$send_{(i_{max}+r)} = u_{max}$

That is, u_{max} appears to be making its way around the ring ....

<u>Pf</u>: Induction on r.

<u>Base</u>: By initialization

<u>Inductive step</u>: Key is that everyone else lets u_{max} pass through. ☒

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

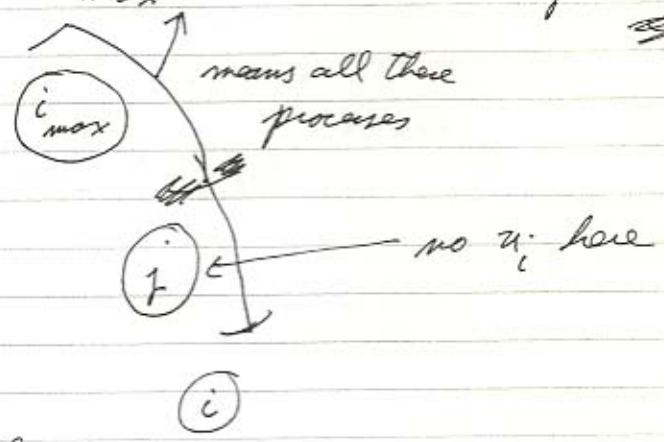
<u>Key fact here</u>: i_{max} uses receipt of u_{max} as signal to set status := leader.

---

2. Uniqueness - no one except i_{max} ever outputs "leader".

Again, need stronger claim, about arbitrary r.

<u>Lemma 3</u>: For any $r \geq 0$, after r rounds,
If $i \neq i_{max}$ & $j \in [i_{max}, i)$ then $send_j \neq u_i$.



Says u_i can't get past i_{max} in the circle

<u>Prove</u> by induction on r.
<u>Key step</u>: Where i_{max} throws away u_i (if no one has already).

$\boxtimes$

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

<u>Invariant proofs</u>: Lemmas 2 & 3 are examples of <u>invariants</u> = <u>properties true in all reachable states</u> (assuming r is in the state)
(actually, so is "if $r = n$ then $status_{i_{max}}$ = leader")

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

I won't do detailed proofs of invariants in class, ~~Book~~ 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 detail for some example proofs.)

Complexity:

What to measure? # of rounds until "leader" $\leq n$

time measure

# of single-hop messages $\leq n^2$

communication measure

Variations:

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

$\leq n^2 + n$ msgs. $O(n^2)$

Non-leader announcements:

all $\neq$ leader could say this, e.g. upon receiving report.

(Could also announce who the leader is.)

~~[illegible]~~ actually, know non-leader earlier, when receive UID larger than its own.

Dist. algs work tries to reduce costs, according to the cost measures that have been defined. Here:

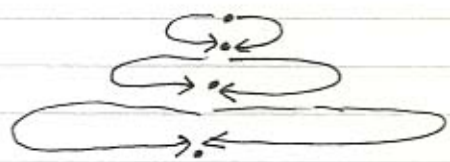
<u>Reducing the message complexity</u>

$O(n \log n)$ rather than $O(n^2)$ (Hirschberg–Sinclair)

~~Con~~ Uses a <u>successive doubling</u> trick. Appears in other dist. algs., esp. where don't know size of network.

<u>assumptions</u>: [Bidirectional comm
Works even if ring size unknown
UIDs with <u>comparisons only</u>.

<u>Informal</u>: Send id in both dirs, to successively greater distances (double each time)



<u>Going outbound</u>, token is swallowed up if it reaches a node whose own UID is bigger

<u>Going inbound</u>, everyone passes everything on.

If you get your <u>own msg in outbound dir</u>, elect self leader.

This describes <u>global behavior</u>.
To fit the model, must write in terms of local process descriptions.
See p. 33.

<u>Correctness</u>: LTTR. Can do with invariants, as for LCR, but more complicated (more bookkeeping).

<u>Complexity</u>:

<u>Time</u>: Worse than LCR, but still $O(n)$

Time for each phase is 2X 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)

<u>Msg cost</u>: $O(n \log n)$

<u>Key idea</u>: $O(\log n)$ phases, number $0, 1, 2, \ldots$

<u>Phase 0</u>: all send both ways $\quad \le 4n$ msgs if all go out & return (though they won't)

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

So, at most $\left\lfloor \dfrac{n}{2^{k-1}+1} \right\rfloor$ start phase k by sending token.

Total # of msgs at phase k, therefore, is

$$\le \underbrace{4}_{\text{out + back, both dirs}} \left(\underbrace{2^k}_{\text{distance}} \left\lfloor \frac{n}{2^{k-1}+1} \right\rfloor \right) \le 8n$$

<u>Total # of phases</u> $\le 1 + \lceil \log n \rceil$ (successive doubling)

<u>So, tot comm</u> $\le 8n(1 + \lceil \log n \rceil) = \boxed{O(n \log n)}$