Class 8
6.852
Fall '01

bring: chalk
code handout

Last time:

Various consensus problems: binary vs arbitrary BA
+ ~~f~~ results
connectivity bds for BA
k-agreement
weak agreement

even got into a bit of basic topology to get a l.b. result.

Today: One more consensus problem - distributed commit.
Then asynch systems, starting with models

Skip approx. agreement this time

Summary: Get different kinds of results if we don't require exact agreement, but only approximate.

Each process starts with real number, must decide on real, subject to agreement: values within ϵ

decision

validity: ~~on~~ decisions in range of initial values of nonf. procs

termination: (or could be approx. range)

Can send reals in msgs.

Can solve, both for stopping + Byz. case.

Can do by solving exactly

But this problem admits different strategies also, using rounds where results "converge" (get closer & closer the more rounds that are executed).

- Can be simpler, cheaper than exact agreement.
- Extends to asynch setting whereas, as we'll soon see, exact agreement strategies do not.

Motivation: Clock synchronization, altimeter values } in presence of failures

The Commit Problem

I'll describe some basic, standard algorithms for the problem of distributed database transaction commit.

Situation: db txns execute at multiple sites in a dist. system, + at each node, wind up with a "commit" or "abort" opinion.

Based on whether ~~they are~~ ~~to~~ the work for that txn has been successfully completed at that site. (results made stable)

Assume { process failures only
stopping only }

n-node undirected, complete graph

Agreement: no 2 procs decide differently (uniformity)

Validity: 1. If any process starts with 0 then 0 is the only possible decision value

2. If all start with 1 + no failures then 1 is only possible decision value.

Asymmetry: to commit, everyone needs to say things are OK. Only guarantee must commit if all want to commit initially + msgs all get thru (best case, normal case)

Termination:

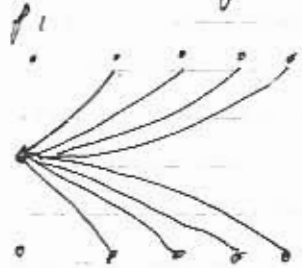
weak version: If there are no failures then all procs eventually decide

strong version: (non-blocking cond) all nonfaulty processes eventually decide.

2P commit

Standard algorithm, blocking (weak termination only).

Assumes distinguished p_1 acts as "coordinator"



p_1 decides consistently with the validity rules

} all send initial values to p_1 (it may send earlier msg to request them)

} p_1 sends out the decision.

(Final dist alg.)

When can they decide:

Anyone whose initial value is 0 decides 0 at beg. ~~(or after rnd)~~

p_1 decides $\begin{cases} 1 \\ 0 \end{cases}$ after getting ~~[illegible]~~ rd 1 msgs, ~~[illegible]~~ consistent with validity cond (if sees a 0 or doesn't receive a msg from someone, decides 0, else 1)

~~Otherwise~~

~~[illegible]~~

~~Otherwise, decides~~

Everyone else decides after rd 2, based on what they hear from p_1.

Agreement (because decision is centralized)

Validity: because of how the coordinator decides

Weak termination: ✓

But not strong term: If p_1 fails before sending all its rd 2 msgs, others won't terminate.

In practice, add termination protocol, where when p_1 fails, others detect + try to finish agreeing on their own.

But this can't always work:

Ex: If

p_1			
0	1	1	1

init values

+ p_1 fails at beg., validity says others should decide 0.

But what if p_1 actually received all the 1s, had a 1 itself, + decided 1 just before failing?

Then 0 would be a disagreement (violates uniform agreement).

but the others can't tell the difference.

Complexity: 2 rds

$\leq 2n$ msgs

3-phase commit

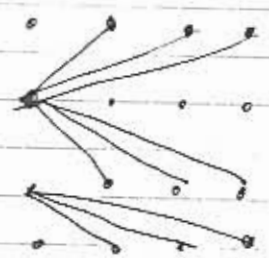
Beef up 2PC to get strong termination

Trick is to introduce an extra stage of agreement, before actually deciding.

3PC: Process states classified into 4 categories:

- dec0: decided 0
- dec1: decided 1
- ready: ready to decide 1 but haven't yet
- unc: uncertain

again, assume p_1 is (initial) coordinator



rd 1: All other procs send their init values to p_1.
All with init value 0 decide 0 ($\rightarrow$ dec 0)
If p_1 gets all 1's + has a 1 itself, then becomes ready
If sees 0 or doesn't hear from someone, p_1 decides 0
($\rightarrow$ dec 0)

rd 2: If p_1 has decided 0, bcasts "decide(0)"
Else bcasts "ready"
Anyone else who receives dec 0 decides 0 ($\rightarrow$ dec 0)
Anyone else who receives ready becomes ready ($\rightarrow$ ready)
After this rd, p_1 decides 1 if it hasn't already decided.

rd 3: If p_1 has decided 1, bcasts "decide 1"
Any other process that receives decide(1) decides 1.

Correctness:
Key is that certain states cannot coexist.
Invariant: (true after 0, 1, 2 or 3 rds)
1. If any process is in ready or dec 1 then all processes' init values are 1.
2. If any process is in dec 0, then $\begin{cases} \text{no process is in dec 1} \\ \text{no non-failed process is ready} \end{cases}$
3. If any process is in dec 1, then $\begin{cases} \text{no process is in dec 0} \\ \text{no non-failed process is uncertain} \end{cases}$

Coexistence possibilities:
dec 0 — unc — ready — dec 1 But fewer possibilities for non-failed

Pf: LTTR ☒

Interesting step: When p_1 decides 1 – check 3rd cond. holds.
It's the end of rd 2. p_1 knows that every other process has either ~~the~~ received "ready" + → ready, or has failed.
✓

(Synchrony critical here – else would have to wait, etc.)
fa ack

Now agreement + validity hold w.r.t. these 3 rds.
But strong termination doesn't yet follow.
E.g. if p_1 fails, can leave others with initial val = 1 undecided after rd 3.

For this need to add more rounds.
usually called "termination protocol"
Amounts to giving others a chance to act as coordinator
E.g., first p_2, then p_3, ... p_n

Rd 4: All send their current status (D0, D1, R, U) to p_2
If p_2 sees any D0 values + hasn't already decided, decides 0.
If " " " D1 " " " " " decides 1.
If all the known values are uncertain then
p_2 decides 0
Otherwise (all known values ∈ {unc, ready} + at least one = ready)
p_2 becomes ready, but doesn't decide yet.

Rds 5 + 6: Are like rds 2 + 3, but with p_2 acting as the coordinator.

Rd 5: If p_2 has (ever) decided, bcasts decide ( ) msg
whatever it is
else bcasts ready.

Any undecided process that receives decide ( ) msg decides accordingly.
Any process that receives ready becomes ready.
Then p_2 decides 1 if hasn't already decided.

Rd 6: As for rd 3.
⋮

Can show inv still preserved.
So get agreement + validity
But now also strong termination.

Complexity: rds: high, 3n worst case (only 3 in "normal case")
msgs: $O(n^2)$ worst case ($O(n)$ normal)

Practical issues with 3PC:

Depends on strong assumptions, which are hard to guarantee in practice.

synch model: But OK since could emulate with time + timeouts

reliable msg delivery: Can retransmit, though must be careful: if it takes too long, synch model not accurately emulated.

** accurate failure diagnosis: Algorithm lets processes conclude that, if they haven't heard from someone, he must have failed.

But <u>hard to guarantee in practice</u>.

Other consensus algs can be used for commit – ones that aren't so dependent on reliability + timing guarantees.

<u>Paxos</u> consensus algorithm is one.

<u>Tolerates many kinds of failures</u>.

another algorithm where <u>all processes try to act as coordinator</u> (sort out conflicts using leader election subalg).

But now a <u>decision of 0 or 1</u> depends on <u>active support</u> by a majority of the processes.

rather than just assuming something about what must be true because you <u>didn't hear</u> from someone.

Asynch Systems

Now switch to asynch model

Processes no longer proceed in rounds, but can go at arbitrary relative speeds.

Will consider two kinds of asynch models in this course:

- asynch networks: consisting of nodes + channels
- asynch ~~sh~~ sh. memory: where procs communicate via shared variables or shared objects

Usually I've started with the ASM.

But this time I'll try something new, going directly to asynch networks

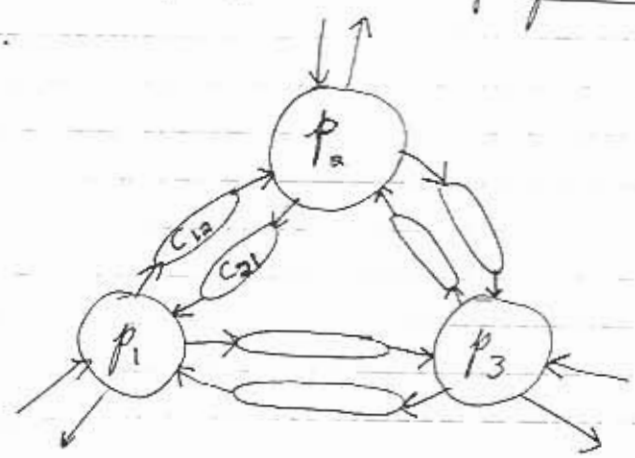
Corresponds closer to the synch networks we've been doing so far

We'll come back to ASM systems.

(They form useful abstractions for programming networks.)

The basic structure will consist of processes communicating via channels.

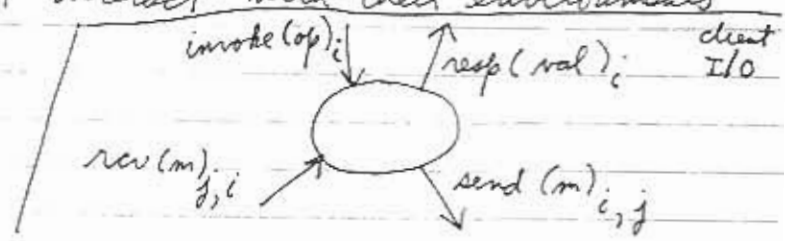
E.g.:



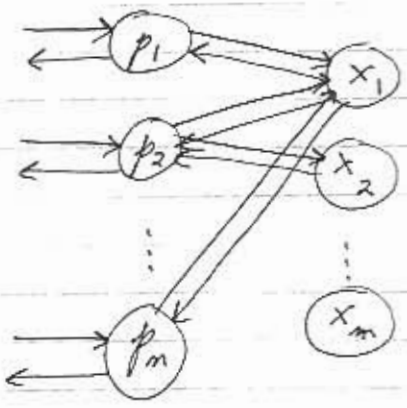
Mathematically, what are the ps and the Cs?

Reactive components, modules that interact with their environments via input and output actions

E.g., invocations + responses
sends + receives



In asynch sh memory systems, we'll also see reactive components, e.g.



Procs have client I + O. plus inv. + response on objects.

Can share some (perhaps with restrictions).

In both cases, we need to talk about reactive components of systems

So I'll start with a general model for reactive components.

I/O Automata

A very general math model for reactive components

Has very little structure

Must specialize it, add structure for particular types of reactive systems – networks, o-o systems, etc.

Designed for describing systems in a "structured" way:

~~Provides notion~~

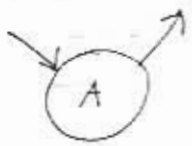
allows description of system components + how they "compose" to yield a larger system.

also allows description of systems at different "levels of abstraction", e.g.,
- detailed impl. vs more abstract algorithm description
- ~~unoptimized, see~~ optimized algorithm vs unoptimized, simpler version (as we've seen before)

Supports good (practical) proof methods: Invariants, compositional methods, simulation relations (like running algs side-by-side)

8.1 I/O Automata

A consists of



sig(A): a signature, which specifies the input, output, + internal actions (acts(A), etc.)

states(A): not necessarily finite

start(A) $\subseteq$ states(A): (nonempty)

trans(A) $\subseteq$ states(A) $\times$ acts(A) $\times$ states(A)

tasks(A), an equiv. reln on local(A) with at most countably many equivalence classes

one reduction:

Requires input-enabling: any input enabled in any state

Remarks:

sig(A): Clarification I/O/Int explain intent

I + O external

out + int locally controlled (under control of component)

input under control of environment, could happen at any time

~~external actions used~~

External actions used as basis for specifying externally-visible behavior of system components.

Input-enabling: Discuss

Not too strong for basic model.

If we want restrictions, can model environment (as another automaton) + put restrictions on it.

Component might sometimes be required to detect bad inputs, or

might exhibit <u>unconstrained behavior</u> in face of inputs – can describe either.

<u>states</u>: Not like FSM, can be infinite (just as in synch model)

<u>trans</u>: labeled by actions

<u>tasks</u>: Think of these as <u>threads of control</u>.
Group actions together that are part of the same thread.
Useful for <u>modeling components that perform > 1 job</u>.
In the formal theory, used to talk about <u>fairness</u> (give fair turns to all tasks, to take steps).
<u>Needed for stating liveness, progress properties</u>; reasoning about whether a system satisfies them.

<u>Examples</u>:

<u>Channel automaton</u>

Reliable FIFO channel, unidirectional, between 2 processes (only one type, could model any type also)

Fix M, msg alphabet.

<u>signature</u>: inputs: $\{send(m) : m \in M\}$
outputs: $\{rcv(m) : m \in M\}$

<u>states</u>: <u>Describe using vbles</u>, as before.
Here, <u>queue</u>, a FIFO queue of M, init λ.

<u>trans</u>: Described by <u>transition defs</u> – little code fragments.
Each t.d. describes a <u>set</u> of transitions, for designated actions (group by type, usually)

send(m)
effect: add m to queue

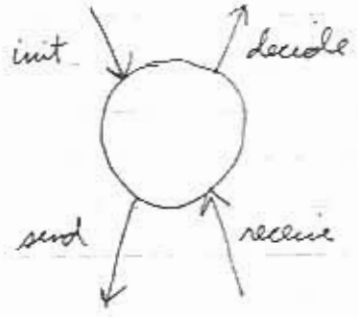
inputs always enabled, so no preconds.

rcv(m)
pre: m = head(queue)
effect: remove head of queue

p. 204

tasks: Put all rcv actions into 1 task

Another ex given in Ch. 8, for a process automaton for some kind of consensus system: (non-FT)



p. 205

Its over: Collects all values – its own by init + others by msgs.
When has a value for all, outputs a value obtained by applying a standard function f.

(Can get new inputs, can change vals, can send + output repeatedly)

Tasks: Separate ones for sending to each nbr
for deciding

How does an IOA execute?
Describe with a formal notion of an execution.
No liveness for the moment, so ignore tasks.

execution = $s_0\ \pi_1\ s_1 \ldots$
↑
start

finite or infinite

Each $(s_i, \pi_{i+1}, s_{i+1})$ is a transition

<u>reachable state</u> : State occurring in some exec.
(equiv., at end of some finite exec)

<u>exec fragment</u> : Same but s_0 needn't be a start state.

<u>Ex</u>: channel

λ, send(a), a, send(b), ab, rcv(a), b, rcv(b), λ

states

finite

Any prefix, e.g. stop here

<u>Still an exec</u>. – no fairness req

<u>traces</u>: Often interested in <u>external beh only</u>

Esp. important when building up system using components.

<u>trace(α)</u> = restriction of α to external acts only

exec or frag

(no state, no internal acts)

<u>Ex</u>: send(a), send(b), rcv(a), rcv(b)

8.2 Operations on Automata

To build system out of components, model needs formal "building" ops.

Composition, hiding (renaming)

Composition: Simply match up same action in different components. All components having that action perform steps involving that action at the same time. (synchronize on actions)

Compatibility: Composing $A_i|_{i \in I}$ I countable (finite or inf.)

Need $\underline{int(A_i) \cap acts(A_j) = \phi}$

Internal acts not seen or shared by anyone else.

Also, $\underline{out(A_i) \cap out(A_j) = \phi}$

Only one aut. controls each output action.

But can have output of one = input of 1 (or more) others.

Allow ∞ many to be composed

Model used for abstract, logical systems, not just physical systems

Including dynamically-created components (e.g. txn systems).

But this necessitates another compatibility restriction:

No action is contained in ∞ many sets $acts(A_i)$

Now define composition $A = \prod A_i$ assuming the A_i are compatible.

sig(A): $out(A) = \bigcup_{i \in I} out(A_i)$

$$int(A) = \bigcup_{i \in I} int(A_i)$$

$$in(A) = \bigcup_{i \in I} in(A_i) - out(A)$$

drivers

$\underline{\text{states}(A)} = \prod_{i \in I} \text{states}(A_i)$ Cartesian prod.

$\underline{\text{start}(A)} = \prod_{i \in I} \text{start}(A_i)$

$\underline{\text{trans}(A)}$: (s, π, s') such that for all i, if $\pi \in \text{acts}(A_i)$
then $(s_i, \pi, s_i') \in \text{trans}(A_i)$

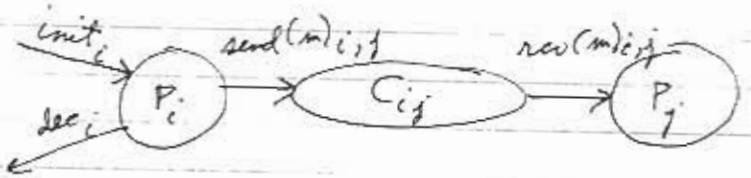
projections

else $s_i = s_i'$

$\underline{\text{tasks}(A)} = \bigcup_{i \in I} \text{tasks}(A_i)$ Maintains all the separate equiv classes (threads)

Later, when we consider fairness / liveness, we'll see this gives the right meaning: concurrent components in a system all keep taking steps.

Ex: Composing channels & consensus processes, as above.
~~begin first~~
Must make sure to disambiguate actions, e.g. $\text{send}(m)_{i,j}$



Subscripts on procs, channels, actions.

(Also, C's M must include the msgs P_i sends.)

Then get a single IOA representing the composed system (picture of 3)

inputs: $\text{init}(v)_i$ for all i

outputs: $\text{decide}(v)_i$, $\text{send}(v)_{i,j}$, $\text{rcv}(v)_{i,j}$ } formally, still outputs, in combined system

internal: none