Class 20
6.852
Fall '01

bring: HW6a
chalk
code-ch 17 — get printed

Reading: 17, 19

Next: 21 (skim), 22

Last time: finished describing impls of atomic objects, esp. R/W atomic objects, in sh memory systems.

Today: related work, on connections between sh memory & network models
& on to global snapshots

Shared Memory + Dist. Networks

Simulating shared memory in distributed network

Popular method for simplifying distributed programming.
Here, will show several ways of doing this, some F-T and some not.
Also, will show how to simulate networks in sh. memory.
This direction is easier, because sh memory is "more powerful".
Useful mainly for carrying over impossibility results from the sh mem model to the network model.

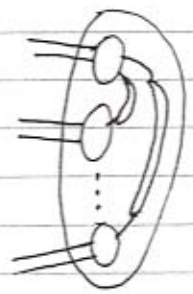
Sh mem → networks:

No-failure case

Start with A, sh. mem system
n ports, 1, ..., n
users U_i

(Need technical restriction described earlier:
For each i, it's always either U_i's turn or proc i's turn to take steps
(Either U_i has no output enabled or proc i has no output or internal enabled)
Allows us to replace sh vars by atomic object impls.
True for the sh mem examples we've seen.)

Want to design dist. network system B.
Same external ports
Procs + channels (S/R FIFO reliable between all pairs)



such that for any exec α of $U \times B$
$\exists$ exec α' of $U \times A$
$\alpha \overset{U}{\sim} \alpha'$

Looks same globally, not just locally.

Also, if α is fair, then α' should be fair
(Talk about fault-tolerance conditions later.)

Simplest scheme:
Locate each sh. var at some process in the network.
When process i of A is supposed to access a var, P_i of B sends a message to the owner and waits for a response.
(Handles locally if it's the owner.)
When response arrives, uses it + resumes the simulation.
On the other side, the owner of a variable handles requests to perform accesses to the object.
Performs on local copy, in one indivisible step.
Sends response.

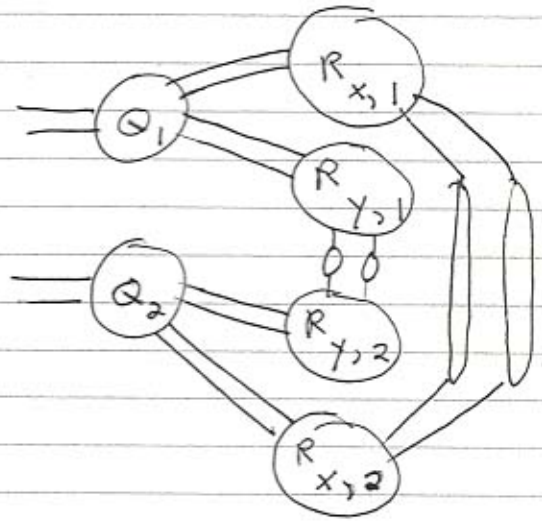
Nice modularity:
Each P_i can be expressed formally as composition of automata,

$$\underbrace{Q_i}_{\text{Simulates proc } i \text{ of } A} \times \prod_x \underbrace{R_{x,i}}_{\text{manages requests involving vble } x.}$$

Q_i: Just use the automata from the general Trans of Ch. 13
(Transformed sh. mem to work with atomic objects of same types – just separates the invs + responses + blocks after invocations.)

$R_{x,i}$: Inputs are invocations from P_i on object x.
Outputs are responses to P_i from " ".

Assume for each x, all the $R_{x,i}$ have dedicated FIFO S/R channels
(Of course, can simulate all by one.)
The composition of $R_{x,i}$ + these channels will be an atomic object of x's type.



$R_{x,i}$ different for the owner of x vs others.

For owner: See code p 568 - 570

Get requests via $\begin{cases} a_{x,i}, & \text{local invocations} \\ \text{messages}, & \text{remote invocations} \end{cases}$

Perform (go over)

For non-owner: Just receives invocation + sends out to owner, awaits response
Code p. 570-571

Correctness:
Easy, based on transf result of Ch. 13 + the fact that for each x, the $R_{x,i}$ + x-channels form an atomic object (composition)

Show this by explicitly defining where op performed (ser. pt).

No fault-tolerance: Process failure kills some vars.

Optimization:

Busy-waiting on a sh. var can be optimized — send one request, let owner of var notify sender when value of var changes (or cond becomes true)

Multi-copy schemes

For improving performance.
Let several procs keep copies of same var.
Then we get coherence problems:

E.g.: read/write vars, multiple copies

Read: any copy
Write: all copies, but not atomically } Read one, write all

Can be faster if read much more than write.

However, coherence issues:

Ex: P_1 + P_2 attempt to write the same x

P_3 + P_4 (who have the copies) can get the writes in opposite orders

yields inconsistent results for later reads.

Ex: even for 1-writer register x

Writer P_1 sends out messages for write of x to all copies.

~~Then~~

a concurrent read at P_2 can happen, getting the new value

Then later, a read at P_3 can get the old value.

(This behavior not allowed by spec of atomic R/W object.)

Need more clever protocols.

E.g., can run all the writes as a single atomic transaction (so that they appear to be consecutive) ~~that all reads/writes~~

- Can implement such a txn with locks, in 2 phases:

  1st phase: lock all copies of x + write them

  2nd phase: release locks

This works because ser pt for write can be placed where all locks are held.

Majority-voting algorithms
(special case of quorum-based)

Write to a majority, read from a majority.

Concurrency anomalies as above imply that a batch of reads or writes has to be run as a transaction, using underlying concurrency control mechanism such as locking.

More precisely,

Each copy includes an integer tag, initially 0.

To read or write perform an atomic txn, e.g. by locking.

P_i read: Read majority, choose copy with largest tag, return the associated value.

P_i write (v): First do embedded-read, determine largest tag t.

Then write (v, t+1) to majority.

Each R or W op has to be done as a txn, "as if" instantaneously.

To see this implements atomic object:

Serialize by using ser. pts from underlying txn impl (e.g. where all locks are held)

Show these ops behave "as if they occurred at their ser. pts".
Depends on handling of the tags:

1. Writes get tags in incr. order, no duplicates, in same order as serialization points.
2. Read gets largest tag write whose ser pt precedes its ser. pt.

(Depends on the fact that all majorities intersect.)

Fault-Tolerance

The previous alg. has no fault-tolerance, because the standard txn impls like locking aren't fault-tolerant:
Someone who fails while holding locks can "kill" the objects it's locking.

An algorithm tolerating failures - ABD (Attiya, Bar-Noy, Dolev)

Can tolerate f stopping failures, $n > 2f$
Assume reliable channels.

Problem def: Introduce $stop_i$ input at each port i
of A (the sh mem system)
+ B (the network system).

Semantics (effect): $\begin{cases} \text{In sh mem, as usual, process stops.} \\ \text{In network, stop process } i\text{, disabling} \\ \quad \text{l.c. actions, but channels all continue.} \end{cases}$

~~Discuss abstract failure actions~~
(Different effect in different models.)

Now $\alpha \to \alpha'$ claim is:

$\forall$ exec α of $U \times B$ — network

$\exists$ α' of $U \times A$ such that — sh mem

$\alpha \overset{U}{\sim} \alpha'$ + stops occur on same procs (as before)

Moreover, if α is fair + contains ~~there are~~ at most f stops (failures) then α' is also fair

(f-simulation def, p. 5)

ABD

Assume: IWmR for all vars of A, the sh. mem system.

(Can be extended to MWMR.)

Again, implement atomic object for each x, separately.

No trans here – very asynchronous !!!

Each process keeps $\begin{cases} \text{copy of } x \\ \text{tag, init } 0 \end{cases}$

write(v): (Assume WLOG that 1 is the unique writer of x.)

Uses next unused tag t (it knows it)

Sets local copy to (v, t) + sends msg (write, v, t) to all others

When receive such a msg, update if > current tag.

In any case, send ack to writer.

When writer knows majority of procs have received tag $= t$, returns ack.

read: Reads own copy + sends read msg to all others, who respond with latest (v, t).

→

, (incl. self)

When hears from majority, <u>prepares to return</u> the (v, t) with the biggest t.

<u>But</u> before doing this, <u>propagates</u> this (v, t) to a majority (including self)
(as in Vitanyi–Awerbuch.)

as before, recipient updates if larger tag; sends ack; sends ack; after majority acks, returns v.

Code $\boxed{\text{p. 576-9}}$

<u>Claim this is an atomic object implementation</u>

<u>Pf</u>: Not obvious.
Easy to see <u>well-formed</u>,
<u>f-failure termin.</u> for $f < \frac{n}{2}$ $\quad (n > 2f)$

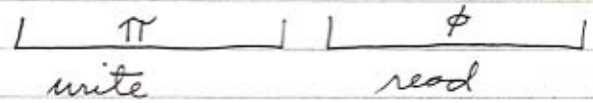
As usual, <u>atomicity</u> is the difficult property to show.
Since this is like V-A, use similar method to prove atomicity.

<u>Lemma 13.16, the p.o. lemma</u>

<u>Here define the p.o. by</u>: $\begin{cases} \text{order writes by tags} \\ \text{order read right after write whose value it gets} \end{cases}$

Show the key Cond. 2: If π finishes before ϕ starts, then don't order ϕ before π.

<u>Cases</u>: <u>E.g.</u>



Then ϕ gets a tag at least as big as what's written by π (by majority intersection), so ordered after, as needed.

E.g. $\underset{\text{read}}{\lfloor \pi \rfloor}$ $\underset{\text{read}}{\lfloor \phi \rfloor}$

then ϕ gets tag at least as big as the one obtained by π.
Because of propagation ~~to [illegible]~~.

Other cases: WW & RW similar (easier).

$\boxtimes$

Can use ABD to get distributed impl of fault-tolerant shared mem algs (1WMR, so far)

By the transf of Ch. 13 - plug in these FT atomic objects for the sh vars

E.g.: $\left\{\begin{array}{l}\text{atomic snapshot}\\ \text{atomic MW register}\end{array}\right\}$ impls in Ch. 13 can be transformed,

via ABD, to corresp dist. network algs.

Fault tolerance: Now original wait-free sh. mem algorithms get transformed into versions that only guarantee f.t. for $n > 2f$.
Since the ABD impl. of atomic R/W objects only tolerates f for $n > 2f$, the whole thing does too.

Impossibility result for $\frac{n}{2}$ failures $(n \leq 2f)$

General fact about the distributed network model is that hardly anything can be guaranteed for $\geq \frac{n}{2}$ failures. (happening at arbitrary times, that is)

Compare with sh. mem model – $\exists$ many wait-free algorithms

Ex: Show can't implement $\begin{pmatrix} m\text{-writer} \\ p\text{-reader} \end{pmatrix}$ atomic register (R/W)

in network where $f \geq \frac{n}{2} = \frac{m+p}{2}$.

Pf: Suppose we could (Canonical type of proof for such things.)

Let $G_1 = \{1, \ldots, n-f\}$

$G_2 = \{n-f+1, \ldots, n\}$

Both $\leq f$, so the system must be able to tolerate the failure of either group (all the procs)

+ The rest should give correct responses for an atomic register.

Suppose WLOG the init value of the reg is 0.

α_1: Fair exec with invocation write(1)$_1$, all G_2 procs fail at beginning.

Write must eventually terminate, with ack$_1$.

α_1' = the finite part of α_1 up to the ack$_1$.

α_2: $read_n$, G_1 procs all fail at beg.

Must eventually terminate, with 0.

α_2' = the part up to response 0.

Now paste to get α:

Do all of α_1' first, then all of α_2'.

Meanwhile, delay all msgs between G_1 + G_2.

Independent behavior, still an execution.

But the ack, precedes the $read_n$, so response of 0 is wrong.
(for atomic R/W reg).

$\boxtimes$

Example implies we can't have any general f-simulation method for $f \geq \frac{n}{2}$.

Because if there were, we could convert a trivial wait-free sh. mem impl. of a R/W atomic object (exists - using one sh var) into an f-FT distributed version.

But no such alg. exists, as we've just seen.

Network → Sh. Mem System

Transformation

Easier, because the sh memory model is more powerful – has reliable sh. mem.

Transf. this way preserves fault-tolerance, even if $f \geq \frac{n}{2}$.

Could use this transformation to run asynch network algorithms in asynch sh mem ~~model~~ system

But not too interesting, because the sh mem model is easier to program.

More interestingly, can use transf to ~~use~~ carry over impos. results from sh. mem to network model.

In particular, impossibility of consensus.

Given: asynch send/rcv system A, digraph G, $stop_i$ event disables P_i + has no effect on channels.

Produce: asynch sh. mem system B simulating A in same sense as before:

$\forall$ exec α of $B \times U$ — sh mem

$\exists$ " α' of $A \times U$ — network

$\alpha \overset{U}{\sim} \alpha'$

+ same failures

Moreover, if α is fair + has $\leq f$ failures then α' is also fair.

(f-simulation)

How: Instead of channel(i,j), use a 1W 1R sh. var $x(i,j)$, writeable by i, readable by j.

Contains queue of msgs, initially empty

Proc. i just adds msgs, none ever removed.

Proc i simulates autom. P_i :

~~Step~~

Step by step.

To simulate send$(m)_{i,j}$, add m to end of $x(i,j)$.

(do with write op, by remembering what you wrote there before)

Also checks all its incoming vars from time to time, looking for new msgs.

(Incremental, keeps track)

If finds any, handles the way P_i handles a received msg.

This yields an n-simulation (tolerates any number of process failures). + gets fair exec of the actual alg.

Corollary: Impossibility of consensus in asynch networks, with even one stopping failure.

Pf: If such an algorithm existed, transform. above would yield 1-FT consensus alg for 1W1R R/W sh. mem, impossible.

In fact !!!

Can extend to impossibility of 1-FT consensus even in asynch network with bcast - a process can put a msg in all its outgoing channels in 1 step, + they are all guaranteed to eventually get delivered.

Still impossible: Could transform such an algorithm into the shared mem model with 1W MR shared vars.

But recall the sh mem impos result holds for that model too!

Very counterintuitive.

Compare with synch network ~~as~~ impos results for consensus, which depend heavily on ~~the~~ ~~[illegible]~~ possibility of failing in middle of beast.

Drop F-T for the ~~[illegible]~~ moment, consider another basic network topic, global snapshots in asynch networks. Ch. 19

No failures. Come back to FT network computing afterwards.