Class 20
6.852
Fall '01

Homework
HW 6a
check codes ch 17 — get printed

Reading: 17, 19

Next: 21 (skin), 22

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

Today: Related work on connections between sh memory & network models

Rolled to global snapshot
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 $\rightarrow$ networks:

No-failure case
Start with A, sh mem system
- m ports, 1,..., m
- uses $U$.

Had technical restriction described earlier: (just $\rightarrow$)
For each i, it's always either $U_i$'s turn or $\emptyset$'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
- pros + channels (5/R FIFO)
  reliable (between all pairs)
- such that for any exec $x$ of $U \times B$
  $\exists$ exec $x'$ of $U \times A$
  $x \equiv x'$
Looks same globally, not just locally.
Also, if \( x \) is fair, then \( x' \) should be fair

(Talk about fault-tolerance conditions later.)

Simpler scheme:
Locate each \( x \) at some process in the network.
When process \( i \) of \( A \) is supposed to access \( x \), \( P_i \) 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,

\[
Q_i \times \Pi \xrightarrow{R_i} \times_{i} \xrightarrow{i}
\]

Simulates proc \( i \) manages requests involving variable \( x \) of \( A \)

\( Q_i \): Just use the automaton from the general flows of Ch. 13
(Transposed so main to work with atomic objects of vars types -
just separate the axis + responses + blocks after invocations.)

\( R_{x,i} \): Inputs are invocations from \( P_i \) on object \( x \).
Outputs are responses to \( P_i \) from \( x \).
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 \( R_{x_i} \) and these channels will be an atomic object of \( x_i \)'s type.

\[ R_{x_i} \text{ different from the owner of } x \text{ vs others.} \]

For owner: See code p 568-570

Get requests via \( \{ \frac{x}{x_i}, \text{ local invocations} \}
\]

\( \{ \text{messages, remote invocations} \} \)

Perform (go over)

For non-owner: just receives invocation & sends out to owner

Await response

Code [p. 570-571]

Correctives:

Easy, based on `trans` 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 of performed (see p7).
No fault-tolerance: Process failure kills some vars.

Optimization:

Busy-waiting on a 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 nodes 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

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

\[ P_3 \text{ then } P_4 \text{ then } P_2 \text{ then } P_1 \]

Yields inconsistent results for later reads.

Ex: 1-writer register \( x \)

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

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.

atomic

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

- Can implement such a turn with locks, in 2 phases:
  
  1st phase: lock all copies of x and then 4 words
  
  2nd phase: release locks

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

Majority-voting algorithms

(special case of quantum-bread)

Write to a majority, read from a majority.

Concurrent 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 a write, perform an atomic turn, 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+i) to majority.

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

So this implements atomic object.

Serialize by using seq. pts from underlying turn 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. Write get tags in incr. order, no duplicates, in same order as serialization points.

2. Ready gets largest tag write whose ser. pt proceeds it ser. pts.

    (depends on the fact that all majors intersect.)


Fault - Tolerance

The previous alg. has me because the standard twin simple like locking aren't fault - tolerant;

Someone who fails while holding locks can "kill" the object it's locking.

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

Can tolerate f stopping failures, m > 2f

Assume reliable channels.

Problem def: Introduce stop: inputs at each part i of A (the sh mem syst.)

+ B (the network syst.)

Semantics (effect): [For sh mem] as usual, process stops.

[For network] stop process i, disable i.e. actions, but channels all continue.

Distinguish failure f actions

(Different effect in different models.)
Now a \( \Rightarrow a' \) claim is:

### A exec \( a \) of \( U \times B \)

\[ \exists a' \text{ of } U \times A \text{ such that} \]

\( a \cap a' \) steps occur on same ports (as before).

Moreover, if \( a \) is fair \( \Rightarrow \) at most \( f \) steps (failures) then \( a' \) is also fair.

\( f \)-simulation def. (p. 5)

\[ \text{ABD} \]

Assume \( \text{IWM} \text{R} \) for all rows of \( A \), the \( \text{sh. mem.} \) system.

(Can be extended to \( \text{MWM} \text{R}. \))

Again, implement atomic object for each \( X \), separately.

Its tags here - very asynchronous

Each process keeps a copy of \( X \)

\[ \{ \text{tag}, \text{unit} 0 \} \]

### write \((v)\) (Assume WLOG that 1 is the unique writer)

Uses next unused tag \( t \) (it knows it)

Sets local copy to \((v, t)\) and sends msg \((\text{write}, v, t)\) to all others.

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

In any case, send ack to writer.

When writer knows majority of facets have received \( \text{tag} = t \), returns ack.

### read

Reads own copy and sends read msg to all others who respond with latest \((v, t)\).
When hears from majority, proposes to return the \((v, t)\) with the biggest \(t\).

But before doing this, propagates this \((v, t)\) to a majority (including self) (as in Vitanyi-Weber/)

As before, recipient updates if larger tag; sends ack; reads ack; after majority acks, returns \(v\).

Claim this is an atomic object implementation.

**Pf:** Not obvious.

Easy to see well-formed for \(f < \frac{m}{2}\) (since \(m > 2f\)).

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

Lemma 13.16, the p.o. lemma

Here define the p.o. by:

\[
\text{order writes by tags, order read right after write whose value it gets.}
\]

Show the key Cond. 2: If \(\phi\) finishes before \(\psi\) starts, then don't order \(\phi\) before \(\psi\).

**Cases:** E.g. \(\phi\) write \(\psi\) read

Then \(\phi\) gets a tag at least as big as what's written by \(\psi\) (by majority intersection), so ordered after as needed.
E.g. \( 1 \to 1 \to 1 \frac{1}{2} \)

```
When \( \frac{1}{2} \) gets to at least as big as the one obtained by \( 1 \),
because of propagation to all nodes.
```

Other cases: \( \text{WW} + \text{RW} \) similar (easier).

---

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

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

E.g.:

\[
\begin{align*}
\{ \text{atomic snapshot} \} & \quad \text{implies in Ch. 13 can be transformed,} \\
\{ \text{atomic MW register} \} & \quad \text{via ABD to consensus dist. network alg.}
\end{align*}
\]

Fault tolerance: Now original wait-free sh. mem. alg.
got transformed into versions that only guarantee
f.t. for \( m > 2f \).

Since the ABD impl. of atomic R/W objects only tolerates
1 for \( m > 2f \), the whole thing does too.
impossibility result for $m \leq 2f$  \( m \leq 2f \)

A general fact about the distributed network model is that

happily anything can be guaranteed for $m \leq 2f$ failures.

(happening at arbitrary times, that is)

Compare with shared memory model - many wait-free algorithms

**Ex:** Show can't implement \((m\text{-writer}) \text{ atomic register}\) \(R/W\)

in network where \( f \geq \frac{m}{2} = \frac{m+f}{2} \).

**Pf:** Suppose we could  \((\text{Canonical type of proof for such things})\)

Let \( G_1 = \{1, \ldots, m-f\} \)

\( G_2 = \{m-f+1, \ldots, m\} \)

Both \( G_1 \) s the system must be able to tolerate the failure of

either group (all the processes)

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

Suppose \( W \leq 6 \) the initial value of the reg is 0.

\( x_1 \): Fair exe with invocation write \((1)\) to all \( G \) processes fail at beginning.

Write must eventually terminate, with ack.

\( x_1' \) = the finite part of \( x \), up to the ack.

$2^1$ read $\rightarrow$ $G_1$ i-syncs all fail at $f_0$

Must eventually terminate, with 0.

$2^1 = \text{the part up to response 0.}\$

Now paste to get $x_i$:

Do all of $x_1$ first, then all of $x_2$.

Meanwhile, delay all msgs between $G_1 + G_2$.

Independent behavior, still an execution.

But the ack precedes the read $\rightarrow$ response of 0 is wrong.

(for atomic R/W req).

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

Because of this we can't convert a trivial want-free
sh. men impl. of a R/W atom. object
into an $f$-FT distrubuted (exists - using one sh var)
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 relative
sh. mem.

Transf. this may preserve fault-tolerance, even if $f \geq \frac{m}{3}$.

Could use this transformation to run async. network algorithms
in async. sh. mem. system
But not too interesting, because the sh. mem. model is easier
to program.

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

In particular, impossibility of consensus.

Given: Async. send/rex. system $A$, digraph $G$, stop. event

disables $P_i$ & has no effect on channels.

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

$A$ exec $\pi$ of $B$ x $U$ sh. mem.

$F = \pi'$ of $A$ x $U$ network

$\pi \sim \pi'$

+ same failures

Moreover, if $\pi$ is fair & has $\leq f$ failures then $\pi'$ is also fair.

(f - simulation)

How: instead of channel $(i, j)$, use a 1W1R sh. var
$x(i, j)$, writable by $i$, readable by $j$.

Contains queue of msgs; initially empty

If $x$ rec. $i$ first adds msg, more are never removed.
Proc. i simulates action. P_i:

Step by step.
So simulate send (m, i, j), add m to end of X(i, j).
(dx 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).

Cadwall: Impossibility of consensus in async networks, even
with only one stopping failure.

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

In fact, !!!

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

Still impossible: Could transform such an algorithm into
the shared mem modeled with 1W MR shared vars.
But recall the sh mem implos result holds for that model too!
Very counterintuitive.
Compare with sync network as empiric results for consensus, which depend heavily on possibility of failing in middle of beast.

Drop F-T for the moment, consider another classic method topic, global snapshot in asych networks. Ch. 19.
No failings. Come back to FT network computing afterwards.