Last Tea:
Stated describing atomic objects
Defined the basic
like an atom, but admit separate innovation and responsibility actions on designated endpoints ("ports")
Also admit stopping failure, modeled as inputs on ports.

Defined the safety requirements of atomic objects in two ways:
1. By saying that all the completed ops + some subset of the incomplete ops can get serialization points, e.g., if the two ops are not moved to the ser file, the result is a correct sequential exec.
2. By defining an abstract object automaton → (next page)

Today go on to describe additional fault-tolerance properties, then describe some common algorithms for implementing atomic objects in shared memory systems.

Next, we'll talk about doing this in networks, in general, relate shared memory & network models.
Note: Atomicity ( mutual-finish ) is a safety property.
( If fails, can be detected at some point in the execution. )
Not completely obvious - it's a complicated property.
( Proof in book uses Hornig's Lemma. )

Alternative proof based on automaton traces:

Alternative spec ( for atomicity )
Equivalent in terms of the set of allowed traces
See 13.1.2

Canonical object automaton:
Keep internal copy of state, plus delay buffers for invs & reqs:

![Diagram of automaton]

Behavior modelled by:
- inv arrives \( \rightarrow \) in-buffer
- perf in separate internal step
  ( same as ser. perf )
- later returns

Has internal perform step ( convenient ), even though we're
only interested in specifying the external behavior.

Every trace exhibited by this automaton ( with W-F us ) is atomic,
according to the definition above.

And every trace allowed by atomic object ( with W-F us ) is exhibited
by this.

So, this can be regarded as an alternative spec.
( Probably better than the trace-oriented one. )
Liencens

Failure-free termination: (basic requirement)
In any fair exec of $AXV$, every inn gets a response.
(Now fairness refers to the ToA def of fairness.)

Atomic object: w-f, atomicity, ff-term, (for all V)
and, as for consensus, can add extra fault-tolerance cards:
wait-free termination:
In any fair exec of $AXV$, every inn on a non-failing port has a response.

Notice I didn't say anything about steps causing ports to disable actions.
I haven't even said there's a sh. men system inside.
Could be something else, e.g., a network - then steps might have different effects.
The point is not to specify this here - just use abstract failure actions at interface, leave consideration of what the effect of the failure is until we consider the impulse.

$-$ Failure termination: $0 \leq f \leq n$:
In any fair exec of $AXV$ in which failures occur on $\leq f$ ports, every inn on non-failing port gets a response.

($+$ book generalizes to $-$ failure termination, I a specific set of ports. Isolates failures on this particular set of ports, (or any subset)
Cute example of m-1 step atomic object

Read/increment on all parts
m-proc shared memory system, values $x(i)$, $1 \leq i \leq m$  R/W roles
N, init. 0
writeable by i, readable by all

increment i: Increment local role
Can do this using write only, by remembering the previous value written.

read i: Read all the shared vars, 1 at a time, in any order.
Return the sum.

wait-free ✓

atomic: Set pt for increment; at actual write.
Set pt for complete read?
Must be somewhere between inv. + resp.

Returns value \( S \geq \text{sum at beg.} \)
\( \leq \text{sum at end} \)

So since increases by 1 each time, \( S \) someplace where the sum is the value returned.

Put set pt. there.

(of course, depends heavily on restricted)

operations
Atomic objects vs sh. nerts
A0s aren't shared variables.
But an important basic result says it's possible to substitute them for
shared nerts in a sh. nert system, & the resulting system
still behaves the same.
This enables modular construction...

The substitution: Given A, a sh. nert system
and given atomic nert B_x for each x,
(same type, interface corresponding to
the connections - who connects what)

Exam (A) is composition of I0As: 31 per process i
1 per sh. nert x

For the nerts, use the B_x's.

For the processes, use P_i's, where:

- inputs of P_i:
  inputs of A on port i
  responses of all B_x's on port i

- outputs of P_i:
  outputs of A on port i
  invocations to B_x on port i

Steps of P_i: Simulate steps of i in A directly, except:
1. When i in A performs access to x, P_i instead issues invocation
to B_x, then blocks.
2. When response arrives, P_i resumes simulating i
3. When stop occurs, all tasks of P_i disabled
   (stop_i also input to the B_x's)
What is preserved by this transformation?

**Theorem**: Given exec \( \lambda \) of Trans \( (A) \times U \) \((U = \text{context of all users})\)

\[ \exists \ \lambda' \] \text{exec of } \ A \times U \text{ (the same user system)} \]

s.t.:
1. \( \lambda = \lambda' \) w.r.t. to \( U \) \((\lambda|U = \lambda'|U)\)
2. For each \( i \), a step occurs in \( \lambda' \) iff a step occurs in \( \lambda \).

That is, looks same to users, \( \lambda \) also same processes fail.

(technically needed: at any point, \( \lambda \) either the system or the user is enabled to do something, but never both.)

**Pf:** Begin \( \lambda \), try to construct corresponding \( \lambda' \).

- Insert sen. pts for all completed ops on the \( B_x \)'s and some of the incomplete ops, as \[ \text{ended} \] by the atomic def.

- Obtain the responses for incomplete ops.

Next, assume the inc & resp actions until they appear just surrounding their sen pts.

Or as far as the \( B_x \)'s are concerned.

For the \( P_i \)'s, might have to more actions past other actions? (would be a problem.)

Claim not: Because \( P_i \) is black.

+ Because no requests will arrive at \( P_i \) from \( U \) (since \( U \) is the system's turn to take steps).

Result is still an exec of Trans \( (A) \times U \). Now remove the \((\text{inc, script, resp})\) triples + replace with single access.
Construction also preserves some fairness:
\[ \text{want: } \alpha \text{ fair } \Rightarrow \alpha' \text{ fair} \]
\[ (\text{so fairness guarantees} \ f \text{-failure termination}) \]
\[ \text{of } \alpha' \text{ carry over to } \alpha \]

So this need extra condition:
\[ \text{Any } i \text{ for which step } \alpha_i \text{ occurs in } \alpha \]
\[ \text{there is some set } I \text{ such that} \]
\[ \text{Every } i \text{ for which step } \alpha_i \text{ appears in } \alpha \text{ is in } I \]
\[ \text{Every } B_x \text{ guarantees } f \text{-failure termination} \]
\[ (\text{e.g., at most } f \text{ failures, } \forall B_x \text{ guarantee } f \text{-failure termination}) \]
\[ \text{"The failures that happen are tolerated by the objects."} \]

Means this construction introduces no blocking of processes.

Only possibility would be if objects don't respond
\[ \text{So insist the objects always respond, except to failed processes} \]

In special case where the objects are robust-free, they never
\[ \text{introduce blocking.} \]

Applications of these results:

1. Special case where \( A \) itself is also an atomic object
   - Say with \( f \)-failure termination.
   - Also suppose the \( B_k \)'s give \( f \)-failure termination.
   - Then the Tarski \( A \) is also an atomic object with \( f \)-failure term.

2. Can build shmem systems hierarchically

   If \( B_{x'} \)'s are themselves shmem systems:
Collapse the P.s. and their agent processes inside
the objects, get a sh mem system

In particular, can build fault-tolerant atomic objects hierarchically
(transitive)

Sufficient cond. for atomicity: Enough to prove for complete ops
(implies for others)

LTTT 53.1.5

RMW Atomic Objects in terms of RW sh vars

Non-fault-tolerant impl:
Use RW var sh. memory mutual exclusion alg.
Access simulated RMW variable within critical region,
which is implemented using fair RW solution to
m.e., e.g. Peterson

1-failure termination:

Theorem: A sh. mem system using RW sh vars that implements
a general (function as inputs) RM/MW atomic object+
guarantees 1-failure termination.
Pf. Suppose there is, B.
Let A be RMW agreement algorithm that uses 1 shared vole, wait-free hence 1-failure.
Let Trans(A) using B to implement the sh var.
Solves agreement in R/W model, with 1-failure term, contend.
(Collapse space so the result is formally in the sh mem model)

Now some example algo to implement atomic objects in the

particular kind of

sh mem model.

\[\text{Snapshot objects} \setminus \text{R/W objects} \]

\[\text{Atomic Snapshot objects.} \]

Think of R/W model where each process has 1 vole it writes, then read it.
Would be nice to add this model the capability for one
process to read all the vars atomically.
Atomic snapshot object provides this kind of capability.
We'll define & show how it can be implemented in terms of simple
R/W shared vars.

\[\text{Variable type: (for snapshot object)} \]
\[\text{values: Vector } v \text{ of fixed length } m, \text{ with values in some}
\text{lower level domain } W.\]
\[\text{init value: } (w_0, \ldots, w_0) \text{ where } w_0 \text{ is init value for } W.\]
Involutions & responses

update \((i, w)\) means write \(w\) into component \(i\)

response ack

snap

with a response that is a vector (instantaneous value)

External interface:

Assume each update port only accepts updates of corresponding vector components.

\[
\text{update } (i, w)
\]

\(m = m + p\) total ports

Implement this atomic snapshot object using 1 process per port \(1\) WMR

\(R/W\) sh word

Unfold variable alg

\(x(i)\) sh. var for update process \(i\), \(1 \leq i \leq m\)

\(i\) can write, all can read (other updaters + snappers)

Each \(x(i)\) holds:

\[
\begin{align*}
&\text{element of } W \\
&\text{tag (unfold integer)} \\
&\text{some other stuff}
\end{align*}
\]
The pros are supposed to make these separate R/W views look like a single atomic snapshot object.

**First idea:** To write \( W \) to component \( i \), process \( i \) updates its own \( X(i) \). Also adds a tag that uniquely identifies the update (sequence #5, starting with \( i \)).

**Q:** What can a snapper do?

- Starts reading all, one at a time.
- Then reads all again, sees no changes (can tell by tags).

In this case, can return these values. The vector actually appeared in memory at some point. This will be the serialization point for the snap.

So, try using an alg. that has the snapper reading everything repeatedly, until it sees 2 identical vectors.

This is correct if it completes.

But: the snap might never return, because of concurrent updates.

**Second idea:** Suppose it ever sees the same \( X(i) \) having 4 different tags \( t_1, t_2, t_3, t_4 \).

Then it knows that the update that wrote \( t_3 \) is entirely contained in the interval of the snap.

(Because: Remnants of \( t_1 \) imply that \( t_2 \) couldn't have completed by beginning of snap, so \( t_3 \) couldn't have started.

Likewise, \( t_3 \) must finish before end of snap because \( t_4 \) begins.)
So, modify the alg so that each update process i does more. Before it writes to \( x(i) \), first executes its own embedded-snap subroutine, just like snap.

When writes \( (\text{value}, \text{tag}) \in x(i) \), also writes result of its embedded-snap ("view").

Now, a snap that sees four different tags for the same \( i \), as above, can return the value of the embedded-snap associated with \( \text{tag}_3 \).

**Summary:** \( x(i) \) has

\[
\begin{align*}
\text{val} \in W, & \quad \text{init } W_0 \\
\text{tag} \in N, & \quad \text{init } 0 \\
\text{view, vector indexed by } f_1, \ldots, m^3, & \quad W, \text{ init } = W_0
\end{align*}
\]

**snap:** Repeatedly reads all, any order until one of:

1. 2 sets return same \( x(i), \text{tag} \) for all \( i \)
   Then snap returns the common vector of values.
2. For some \( i \), 4 distinct values of \( x(i), \text{tag} \) are seen
   Then return \( x(i), \text{view from the third tag} \).

**update \( (w) \):** Perform embedded-snap, (same as snap but common not returned to user)

**Write to \( x(i) \):**

\[
\begin{align*}
1. \text{val} := w \\
2. \text{tag} := \text{next unused (local)} \\
3. \text{view} := \text{vector returned by embedded-snap} \\
   \quad \text{+ then send ack to user}
\end{align*}
\]

Then: snapshot atomic object, wait-free.
Pf: Well-formedness clear.
Wait-free termination easy - always returns by one case or the other.

Atomicity: 3ix2, exec of alg + users.
Assume WLOG all ops complete. (suffices, by lemma)
Insert scripts: { update: at point of writes
snap: more complicated:

Do for all embedded-snaps as well as regular snaps:

1. Show that terminate by finding two - sets of reads
Choose any point between end of first set + beginning of second set.

2. Show that terminate by finding 4 + tags
Insert ser pts for these 1 by 1, in order of their response events.

Recursively.
For each, say T_i, note that the read returned cases from embedded-snap &
whose interval is totally contained within interval of T_i:

\[ T_i \]

So by induction, \$ has already been assigned a ser. pt.
Insert ser. pt. for \$ right after that for \$.
all scripts in required intervals:

- Observe for upates.
  
  first type of snap.

- Second type, argue by induction on \( \# \) of response events.

Result of shrinking is a serial trace, because each snap returns the "correct" vector at its serialization point.

Result of all writes up to that point

(Observe for type 1 snaps; for type 2, follows by induction on response events.)

Complexity:

- Size
  
  kubdd vars

  \( m \) vars for length \( m \) vector

  Size for snap: \( \leq (3m+1)mn \) all non-accesses

  \( O(m^2) \) time

   Since inodes are large, published analyses charge based on length of vector, so gives added factor of \( m \).

   in view component

Time for update: Also \( o(m^2) \) because of embedded snap.

Algorithm using add vars:

- Can do best tag racing but tricky handshake protocol needed.

- Attiya et al., LITR (in book)

Other snapshot algos exist, with better complexity, but complicated.