FAULT-TOLERANT DISTRIBUTED COMPUTING IN THREE TIMING MODELS

and

ATOMIC OBJECTS

N Lynch

Rapporteur: Dr Paul Ezhilchelvan

Note: Prof. Lynch's two talks are derived from her book "Distributed Algorithms," recently published by Morgan Kaufmann Publishers. The abstracts appear below. The talk slides follow; they include references to specific book chapters.

FAULT-TOLERANT DISTRIBUTED COMPUTING IN THREE TIMING MODELS

Professor Nancy Lynch
MIT
Laboratory for Computer Science
545 Technology Square (NE43-365)
Cambridge, MA 02139
USA

The choice of timing model is a key factor in determining solvability and complexity results for problems in fault-tolerant distributed systems. In this talk, three basic timing models - the synchronous, asynchronous and partially synchronous models - are described. Solvability and time complexity results for consensus problems in these three models are discussed and compared. Related open problems are posed.

and

ATOMIC OBJECTS

Atomic (linearizable) objects are a basic building block for distributed systems. In this talk, atomic objects are defined and their basic properties presented. Some sample implementations are given, along with various useful theorems for proving correctness of implementations. Some uses of atomic objects as building blocks for shared-memory and network-based distributed systems are described.

Fault-Tolerant Distributed Computing in Three Timing Models

Nancy Lynch
MIT

September, 1996

From [Lynch, *Distributed Algorithms*]
Chapters 6, 7, 12, 17, 21, 25

Overview

Choice of timing model is critical for solvability, complexity results in distributed systems.

Describe three timing models:
- Synchronous
- Asynchronous
- Partially synchronous

Discuss, compare solvability, time complexity results for consensus problems, in three models.

Discuss open problems.

I. Introduction

Field of distributed algorithms contains
algorithms, impossibility results for:
leader election, network searching, consensus,
mutual exclusion and other resource allocation problems,
atomic objects, reliable communication, global snapshots,
etc.

Also time complexity upper bounds, lower bounds.

Mostly individual results.
Want more general theory, for many problems, models.
Various timing assumptions.
Various interprocess communication methods,
shared memory vs. message-passing.

Good first step:
Compare results about basic problems across different models.

Here, *fault-tolerant consensus*.

Consensus problems:

Distributed processes agree on something:
  binary decision, sensor reading, clock value, ...
Used for transaction commit, avionics, fault diagnosis, ...

Simple mathematical structure,
  leads to basic, elegant results.

Many fundamental results, interesting comparisons.

Many interesting techniques.
  flooding algorithms, chain arguments,
  stretching and shrinking, topology,
  model transformations, bivalence arguments, ...

Gaps remain.

And many more problems to study.

Outline of Talk:

II. Consensus Problems

Ordinary agreement [Pease, Shostak, Lamport 80]

Message-passing model
Not considering Byzantine failures – just process
stopping failures.

Agreement:
No two processes decide on different values.

Validity:
Any decision value is some process' initial value.

Termination:
All nonfaulty processes eventually decide.

k-agreement [Chaudhuri]

Agreement:
There are at most k different decision values.

III. Synchronous model

Message-passing, rounds.
Time measure = number of rounds.

Floodset algorithm, f failures [Dolev, Strong]:

```
Each process maintains subset $W$ of values,
   initially containing its initial value.
For $f+1$ rounds, broadcast $W$, accept all received
   values into $W$.
At end, choose $min(W)$.
```

Correctness:

Validity, termination easy.

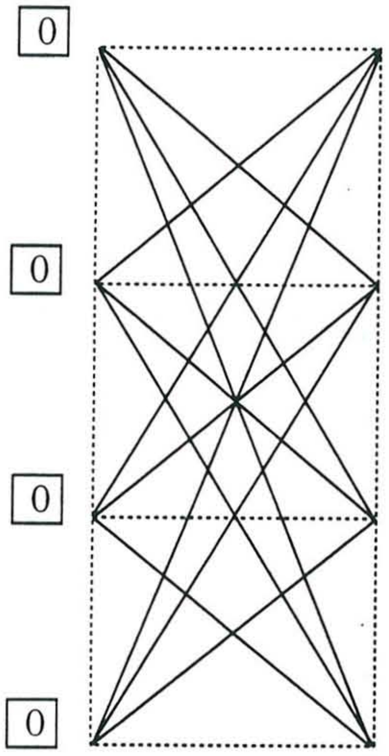
Agreement:

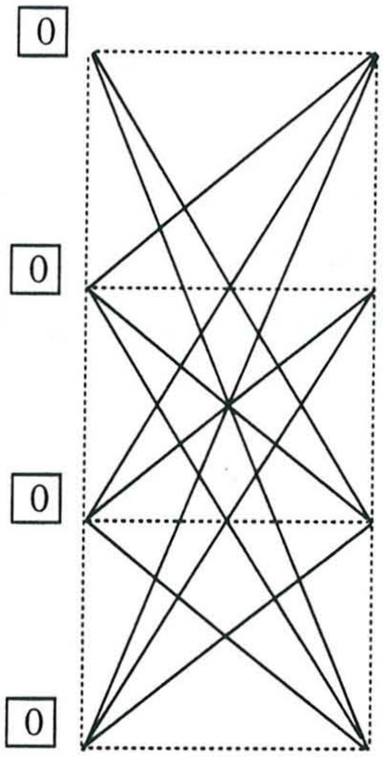
Key: If no process fails during a round r, then all non-failed processes have the same W immediately after r (and thereafter).

Lower bound of $f+1$ rounds [Fischer, Lynch; Merritt]:

Chain argument.

E.g., $f = 1$, can't do in 1 round.





E.g., $f = 2$, can't do in 2 rounds.

Longer, more complicated chain.
Intermediate executions ρ_i:
  Processes $1, \ldots, i$ have input 0,
  $i + 1, \ldots, n$ have input 1,
  no failures.

E.g., spanning from ρ_0 to ρ_1
  Changing process 1's initial value from 0 to 1

First remove all process 1's round 2 messages.
But: now can't remove round 1 messages:
  Destination could inform others in round 2.

Solution:
  Use several steps to remove round 1 message to i.
  1 and i both faulty, in intermediate executions.
  OK since $f = 2$.

k-agreement:

FloodMin algorithm, $\lfloor \frac{f}{k} \rfloor + 1$ rounds [Chaudhuri]:

Each process maintains *min* value,
initially its initial value.
For $\lfloor \frac{f}{k} \rfloor + 1$ rounds, broadcast,
keep min of all values seen.
At end, choose *min*.

Correctness:

Define $M(r)$ = set of distinct *min* values for
non-failed processes after round r.

Set $M(r)$ can only decrease.

Key:
If at most $d - 1$ processes fail during round r, then
$|M(r)| \leq d$.

Lower bound of $\lfloor \frac{f}{k} \rfloor + 1$ rounds
[Chaudhuri, Herlihy, Lynch, Tuttle]:

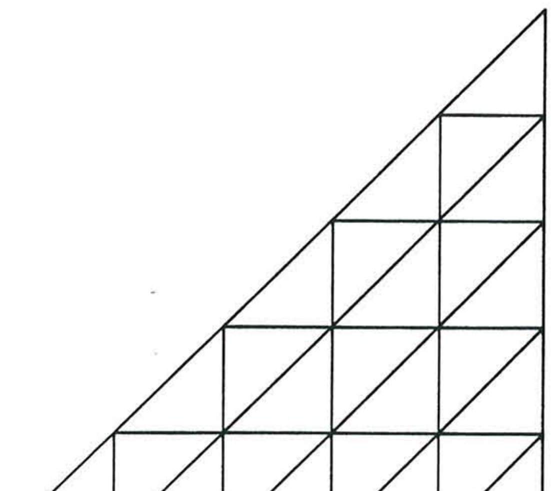
Same as upper bound.

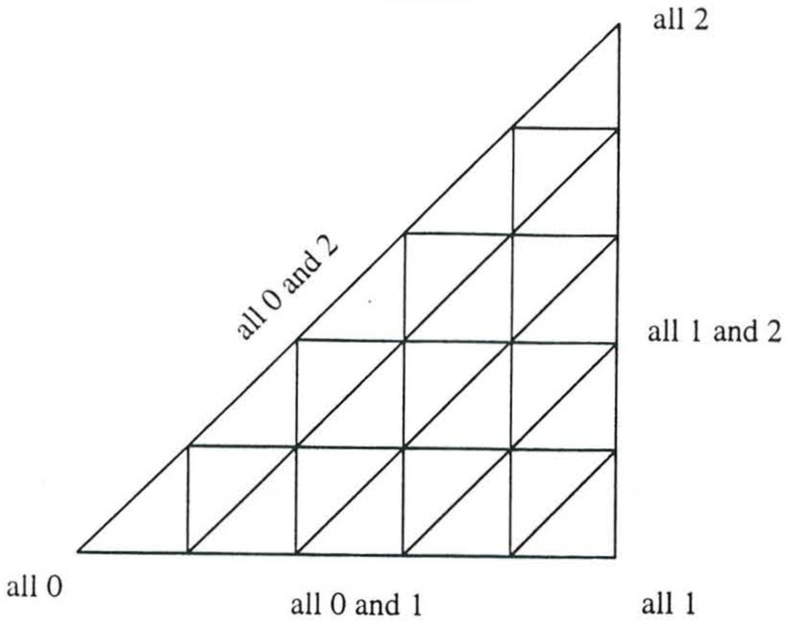
Proof by contradiction:
Fix algorithm solving k-agreement in $r \leq \lfloor \frac{f}{k} \rfloor$ rounds.
Values $= 0, \ldots, k$.
Construct execution with $k + 1$ distinct decisions.

Recall chain argument for $k = 1$.
Not enough here: decision values in one execution do not determine decision values in next execution.

New idea: Construct k-dimensional "chain" of executions.

Bermuda Triangle B:
k-dimensional simplex (algebraic topology)





Assign execution to each vertex of B.

Corner: All processes with same input, no failures.
Vertex on edge: Only inputs from two endpoints.
Vertex on any face: Only inputs from corners of face.

Validity implies same restrictions on decisions.

Also assign process to each vertex.

Ensure that for each tiny simplex, there is some execution that looks like the execution labelling each of its vertices, to the (nonfaulty) process labelling that vertex.

Depends on $r \leq \lfloor \frac{f}{k} \rfloor$.

Color each vertex with "color" in $\{0, \ldots, k\}$:
Assigned process' decision in assigned execution.

Properties:
1. Colors of $k+1$ corners all different.
2. Color of each point on any edge (face) is color of one of the corners of the face.

Sperner Coloring

Sperner's Lemma: Some tiny simplex is colored with $k+1$ distinct colors.

Yields execution with $k+1$ decision values.

Contradiction.

Synchronous case:
Bounds tight, for ordinary agreement and k-agreement.

IV. Asynchronous Model

Message-passing, reliable eventual delivery.

Ordinary agreement:

Impossibility for $f = 1$ [Fischer, Lynch, Paterson]
Fundamental fact about fault-tolerant asynch systems.

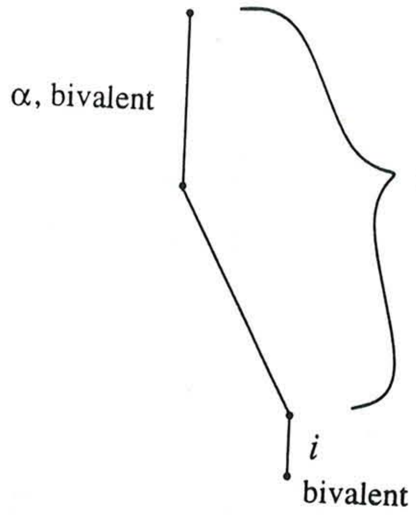
Proof: Do first for asynch read/write shared memory model, then use model transformation from message-passing to shared memory model.

Proof for shared memory:
By contradiction, using bivalence argument.
Fix algorithm solving agreement for 1 failure.

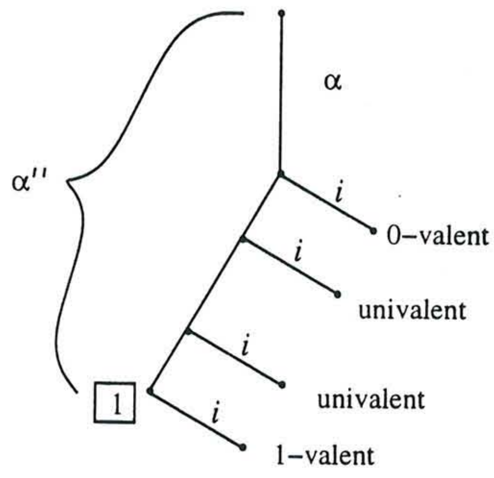
0-valent, 1-valent, univalent, bivalent executions

Lemma 1: There is a bivalent initialization.
Proof: Chain argument

Lemma 2: If α is a bivalent failure-free execution, and i is any process, then there is a failure-free extension α' of α such that $extension(\alpha', i)$ is bivalent.

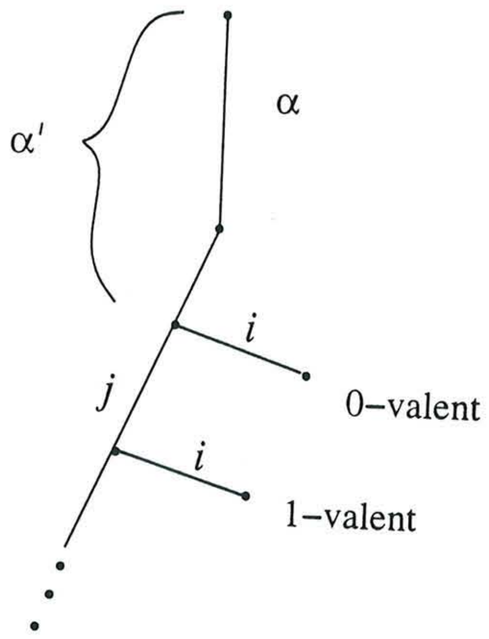


Proof: By contradiction.
Fix α, i for which this is false.



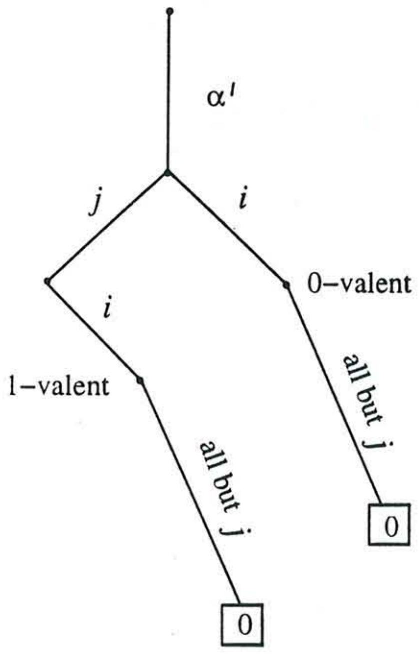
Proof of Lemma 2, cont'd:

Then somewhere:



Case analysis for contradiction.

E.g., $j \neq i$, both write same variable.
Then states after ji and after i indistinguishable
to all processes except j.



So we have:

Lemma 2: If α is a bivalent failure-free execution, and i is any process, then there is a failure-free extension α' of α such that $extension(\alpha', i)$ is bivalent.

Main impossibility proof:

Use Lemma 2 repeatedly, round robin.
Yields failure-free execution in which no process decides.

Impossibility proof for message-passing model:

Read/write shared-memory model can simulate message-passing model.
Fault-tolerance-preserving transformation.
So impossibility for shared memory implies impossibility for message-passing.

k-agreement:

Algorithm for k-agreement, tolerating $k-1$ faults:
  Processes $1, \ldots, k$ (only) broadcast their
    initial values.
  Each process decides on the first value it receives.

Impossibility of solving k-agreement with k faults
  [Herlihy, Shavit; Borowsky, Gafni; Saks, Zaharoglu]:

Generalizes the impossibility result for ordinary agreement.

Algebraic topology

Asynchronous case:
  Results tight, for ordinary agreement and k-agreement.

V. Partially Synchronous Model

Message-passing, reliable delivery in time d
Lower, upper bounds ℓ_1, ℓ_2 on process step time.
  $L = \frac{\ell_2}{\ell_1}$, timing uncertainty
Process stopping failures

Time measure: Real time

Failure detector building block:
  Times out stopped processes.
  Detection time at most $Ld + d + O(L\ell_2)$
  Ignoring ℓ_2, this is $Ld + d$.

Ordinary agreement:

Upper bound by transforming synchronous model:
  $f(Ld + d) + d$

Lower bound from synchronous model:
  $(f + 1)d$

Previous results yield:

Upper bound: $f(Ld + d) + d$
Lower bound: $(f + 1)d$

Gap – Multiplicative L (mostly)
Close this gap?

Have partial results:

Upper bound: $Ld + (2f + 2)d$
Lower bound: $Ld + (f - 1)d$

Note L is not multiplied by fd:
Timing uncertainty has impact for only one round.

Algorithm [Attiya, Dwork, Lynch, Stockmeyer]:

Use failure detector.

Asynchronous rounds $0, 1, \ldots$.

Can only decide 0 at even rounds, 1 at odd rounds.

Round 0:

```
If input is $1$:
    Send $goto(1)$ to all processes.
    Go to round $1$.

If input is $0$:
    Send $goto(2)$ to all processes.
    Decide $0$.
    Send $decided$ to all processes.
```

```
Round $r > 0$:

Wait until either:
  Have received $goto(r + 1)$ from anyone, or
  Have received $goto(r)$ from all that haven't failed
    or decided.

If received $goto(r + 1)$:
  Send $goto(r + 1)$ to all processes.
  Go to round $r + 1$.

Else:
  Send $goto(r + 2)$ to all processes.
  Decide $r \bmod 2$.
  Send *decided* to all processes.
```

Correctness:

Suppose process i decides at round r, none earlier.
First broadcasts $goto(r+2)$ message.
Prevents anyone else from "trying to decide" at round $r+1$.
So no one ever sends $goto(r+3)$.
So no one ever reaches round $r+3$.

So all decisions occur at rounds r and $r+2$, agree.

Validity easy.

Time bound: $Ld + (2f+2)d$

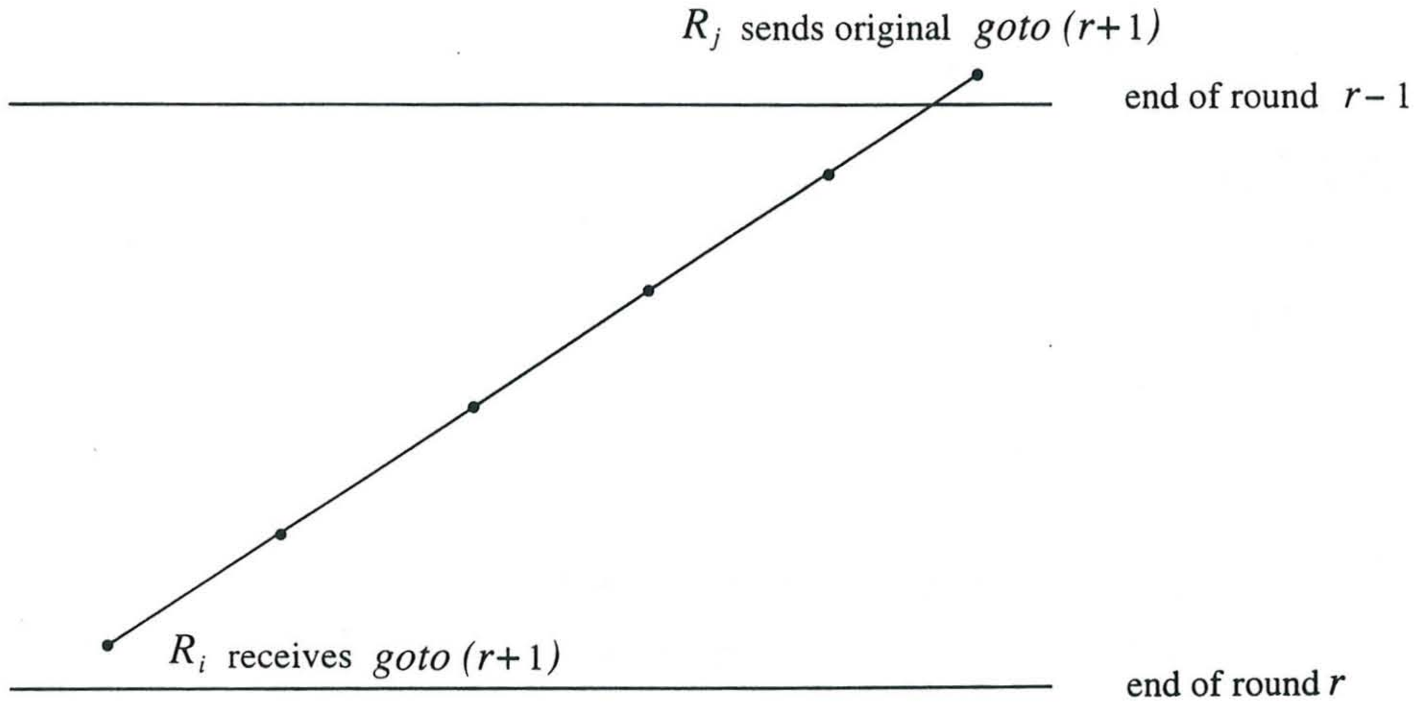
$T(r) =$ first time when every process has failed, decided,
or advanced to next round, $r+1$.

$T(r) \leq T(r-1) + Ld + d$
Depends on timeout.

Quiet round r: Some process never receives $goto(r+1)$.

No process ever advances past a quiet round.

If r is non-quiet, then $T(r) \leq T(r-1) + (f_r + 1)d$, where f_r is the number of processes that fail at round r.


R_j sends original $goto\ (r+1)$
end of round $r-1$
R_i receives $goto\ (r+1)$
end of round r

Lower bound of $Ld + (f-1)d$
[Attiya, Dwork, Lynch, Stockmeyer]:

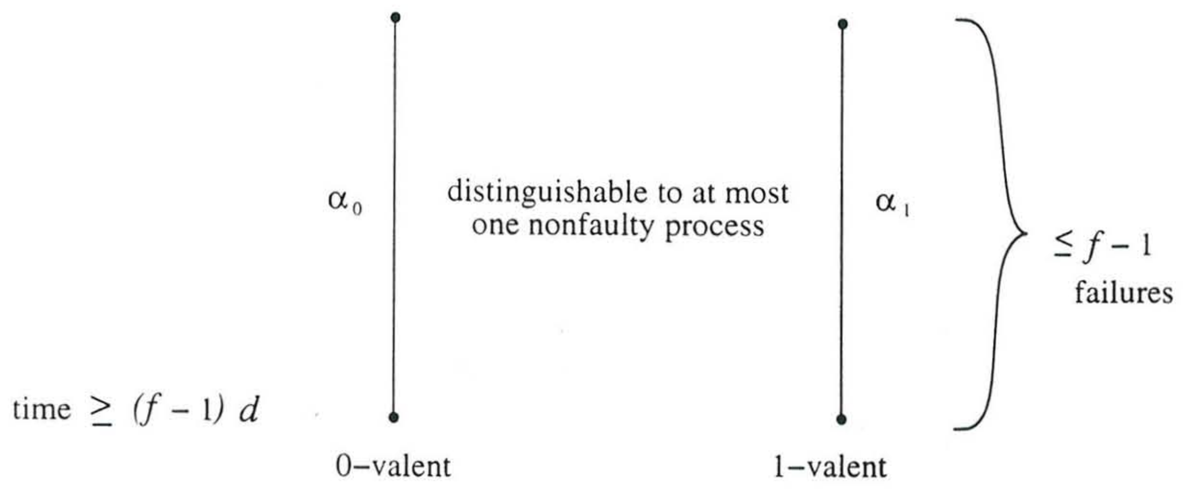
Proof:
By contradiction, using chain arguments,
bivalence, stretching and shrinking.
Fix algorithm taking time $< Ld + (f-1)d$.

Restrict to "synchronous" executions:
Process steps at same times, $t_1, t_2, \ldots$.
Fixed orders of messages, etc.
Blocks $B_1, B_2, \ldots$

Block extensions.
fast
slow
failure-free
fff

0-valent, 1-valent, univalent, bivalent
Defined with respect to fff extensions only

Lemma 1: Bad combination of executions doesn't exist:

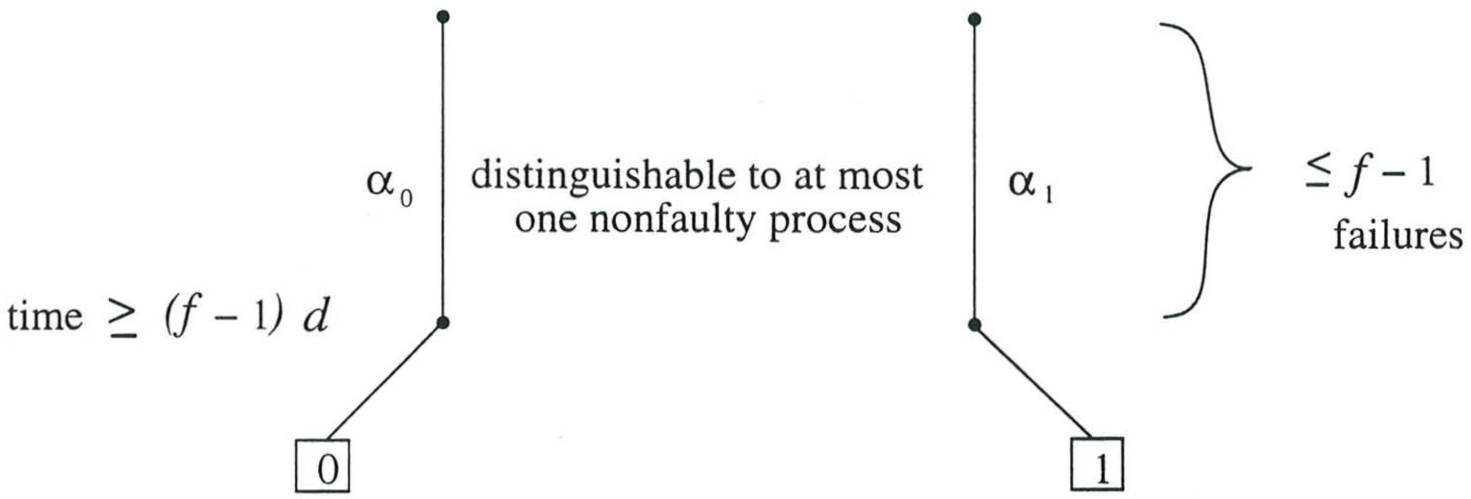


Proof:
Extend both in same way, by slow extensions,
first failing distinguishing process i.
Must decide within additional time $< Ld$.
Decide in same way, say 0.

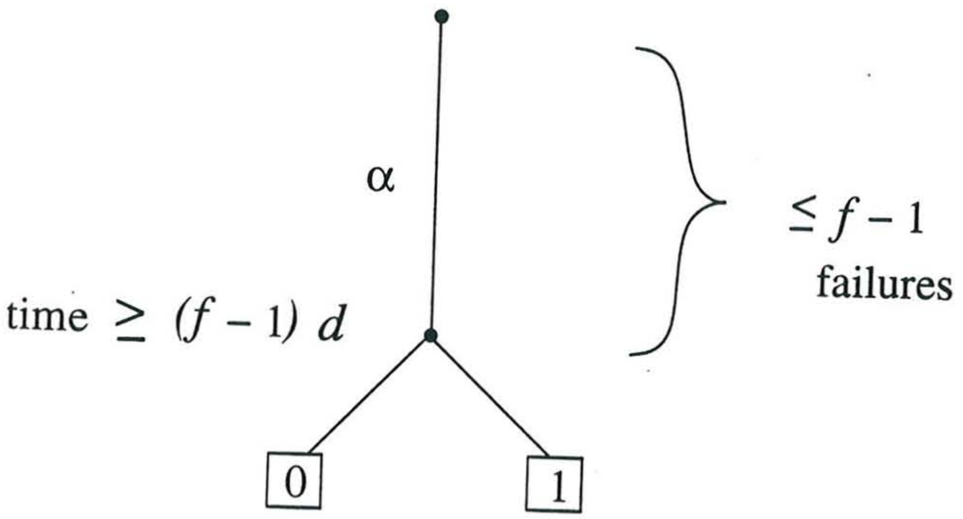
Speed up so time $< d$.
Keep i alive, but delay its messages to time d.
Get fff execution violating 1-valence.

Get contradiction by producing bad combination.

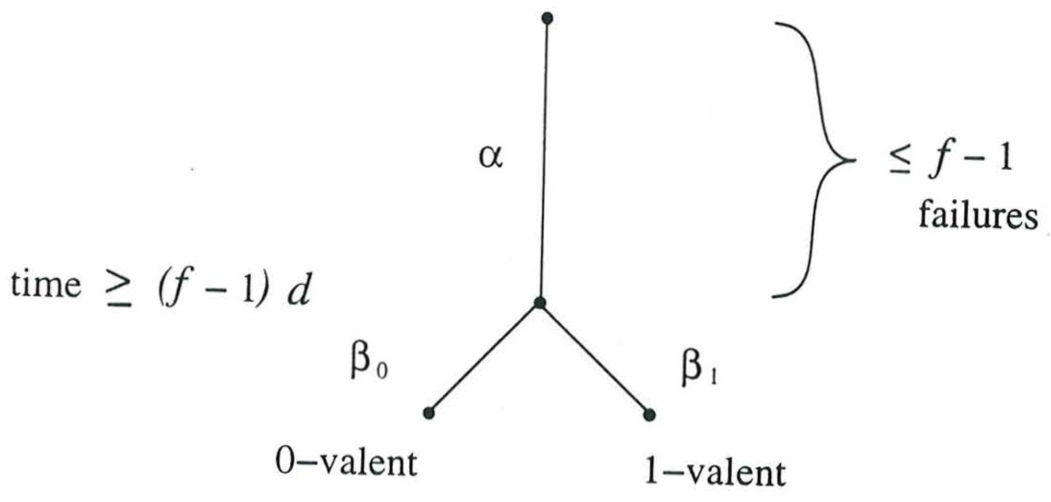
Lemma 2: Combination exists:



Lemma 3: Exists:



Lemma 4: Exists:



Proof: By maximality argument.

But this yields a bad combination, contradiction.

Upper bound: $Ld + (2f+2)d$
Lower bound: $Ld + (f-1)d$

Open question: Close the gap.

k-agreement:

Upper bound from synchronous model:
$\lfloor \frac{f}{k} \rfloor (Ld + d) + d$

Lower bound from synchronous model:
$(\lfloor \frac{f}{k} \rfloor + 1)d$

Gap – Multiplicative L

Close gap? Open question.

E.g., can we show close bounds of approximately
$Ld + \frac{f}{k}d$?
Timing uncertainty impact for only one round?

Partially synchronous case:
Small gap (factor of 2) for ordinary agreement.
Large gap for k-agreement.

VI. Discussion

Summary:

Gave possibility/impossibility, time complexity results
for consensus problems, in three basic timing models.

Illustrated many techniques:

flooding algorithms, chain arguments,
stretching and shrinking, topology,
model transformations, bivalence arguments, ...

A basic step toward a general theory – compare results about
basic problems across different models.

To Do:

Close gaps in results for partially synchronous model,
explaining exactly the impact of timing uncertainty.

Study other problems in the three timing models.
Snapshot, synchronizers, ...

Obtain general characterization results, for many problems.
Topology [Herlihy, Shavit; Borowsky, Gafni]

Obtain general results relating the three timing models.
E.g., relate asynchronous impossibility results to
limit of partially synchronous lower bound results?

Atomic Objects

Nancy Lynch
MIT

September, 1996

From [Lynch, *Distributed Algorithms*]
Chapters 13, 17, 18

Overview

Atomic (linearizable) objects are basic building blocks for fault-tolerant distributed systems.

Define atomic objects.

Present basic properties.

Give sample implementations.

Give theorems showing correctness of implementations.

Show some uses of atomic objects as building blocks for fault-tolerant shared memory and network-based distributed systems.

Discussion, open questions.

I. Introduction

Atomic objects are like shared variables, but accessed concurrently by several processes.
Accesses appear to occur sequentially, in order consistent with invocations and responses.

Fault-tolerance conditions:
- Stopping failures only.
- Wait-free termination.
- *1*-failure termination, *f*-failure termination

Building blocks for multiprocessor systems:
- E.g., start with single-writer/single-reader read/write atomic objects, build.
- Simple, modular, provably correct.
- Performance?

Building blocks for asynchronous network systems
- Many dist. algs. are designed to provide appearance of centralized, coherent shared memory.
- Formally, distributed implementations of atomic objects.

Outline of Talk:

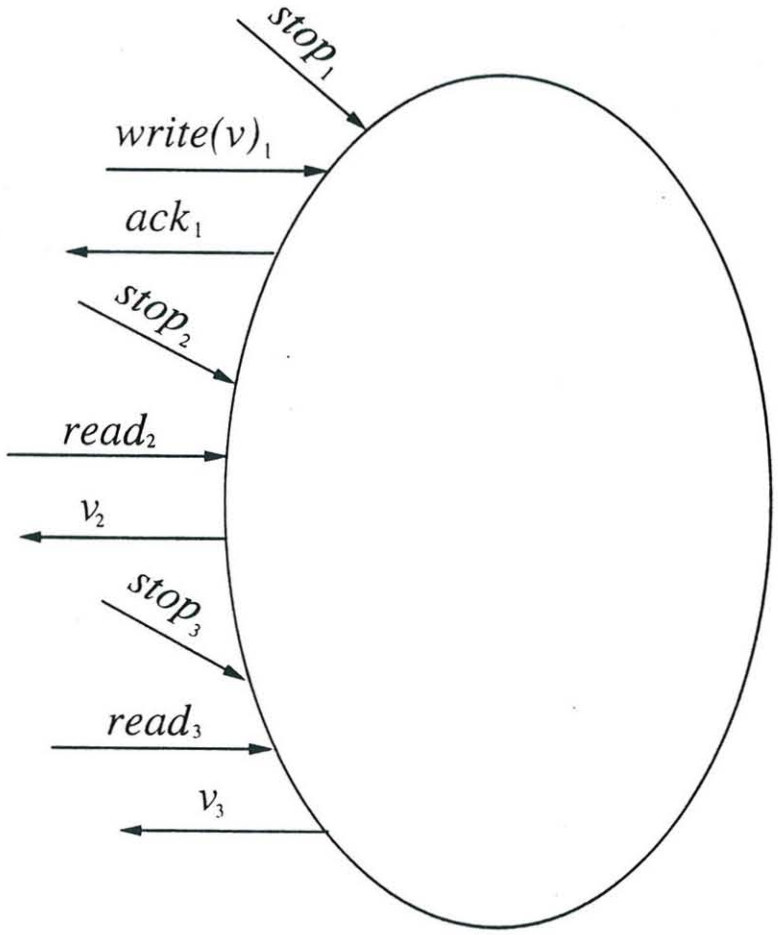
II. Definitions and Basic Results

Atomic object definition:

I/O automaton with n ports.
Invocation, response actions on each port,
   Alternating (well-formed) on each port.
   Particular data type.

Modelling failures: $stop_i$ input actions

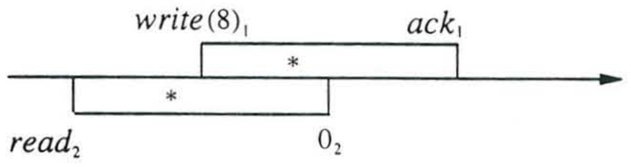
Example: 1-writer/2-reader read/write object interface:



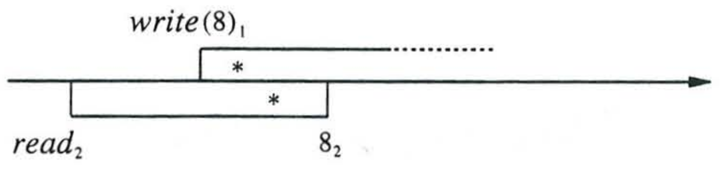
Atomicity property (for a sequence of user actions):

Can choose some of the incomplete operations, Φ,
Invent responses
Insert serialization points within all complete operations
and all operations in Φ
So that shrinking to serialization points gives trace
of underlying data type.

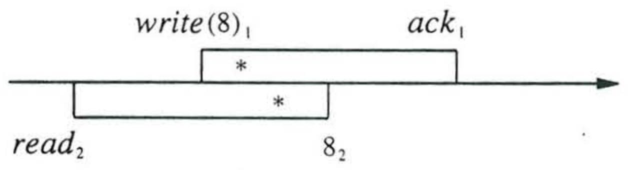
Examples:



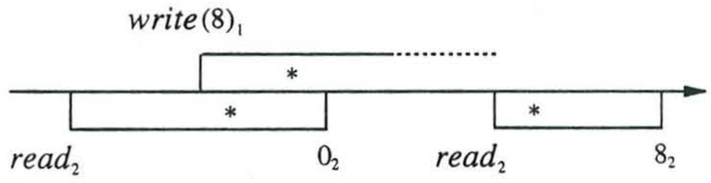
(a)



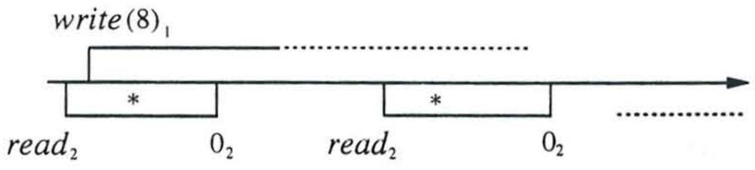
(c)



(b)

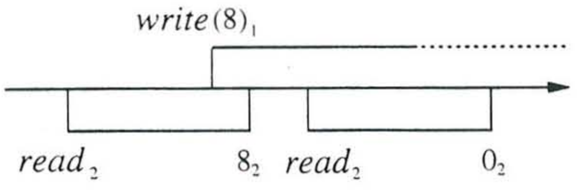


(d)

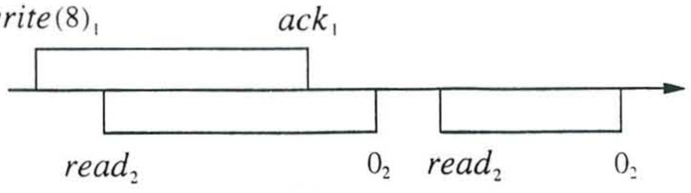


(e)

Examples: Not atomic



(a)



(b)

Atomicity property for executions.

Atomicity condition:
Any (finite or infinite) well-formed execution satisfies
the atomicity property.

Theorem:
The combination of well-formedness and atomicity is a safety property, for traces.

Safety property:
Nonempty, prefix-closed, limit-closed set of traces.

Proof:
Konig's Lemma.

Liveness conditions:

Failure-free termination
In any fair, failure-free execution, every invocation has a response.

Wait-free termination
In any fair execution, every invocation on a non-failing port has a response.

I-failure termination
If $stop$s occur only on ports in I, then every invocation on a non-failing port has a response.

f-failure termination
If $stop$s occur on at most f ports, then every invocation on a non-failing port has a response.

Example: Read/increment atomic object

Domain N, initial value 0,
Operations *read*, *increment*

Asynchronous n-process shared memory system,
read, *increment* on each port.

Shared read/write registers $x(i)$, $1 \leq i \leq n$
$x(i)$ writable by process i, readable by all.

$increment_i$: Increment $x(i)$

$read_i$: Read all the x's, return sum

Guarantees atomicity:
Serialization point for *increment*: at *write*
Serialization point for *read*: at point in its
interval where sum of x's equals return value.

Wait-free termination.

Canonical wait-free atomic object automaton C:

Maintains internal copy of shared variable, initially v_0.
$\mathit{inv\text{-}buffer}$, $\mathit{resp\text{-}buffer}$
Separate steps for invoke, perform, respond.

Also handles liveness.

Theorem: C is an atomic object.

Theorem: Any implementation of C is an atomic object.

Theorem: Any atomic object for the same type implements C.

Composition of atomic objects:

Theorem:
If A and B are atomic objects with ports $\{1, \ldots, n\}$, then $A \times B$ is an atomic object (of the product type), with the same ports.

Moreover, if A and B guarantee I-failure termination, then so does $A \times B$.

Similarly for f-failure termination, wait-free termination, failure-free termination.

Atomic Objects vs. Shared Variables

Can substitute atomic objects for shared variables in a shared memory system.
Under certain circumstances, result behaves the same with respect to the users.
Permits modular construction of shared memory systems.

The circumstances (technical):
For each i, at each point in time, it is either the system's turn or the environment's turn to take a step.

The substitution:
Process P_i behaves like shared memory process i, but instead of directly accessing shared variable, it issues invocation, waits for response.

Theorem:
For any execution α of the transformed system T,
there is an execution α' of the shared memory system S
that is indistinguishable to all users,
and that has the same failures.

Transformation theorem, continued:

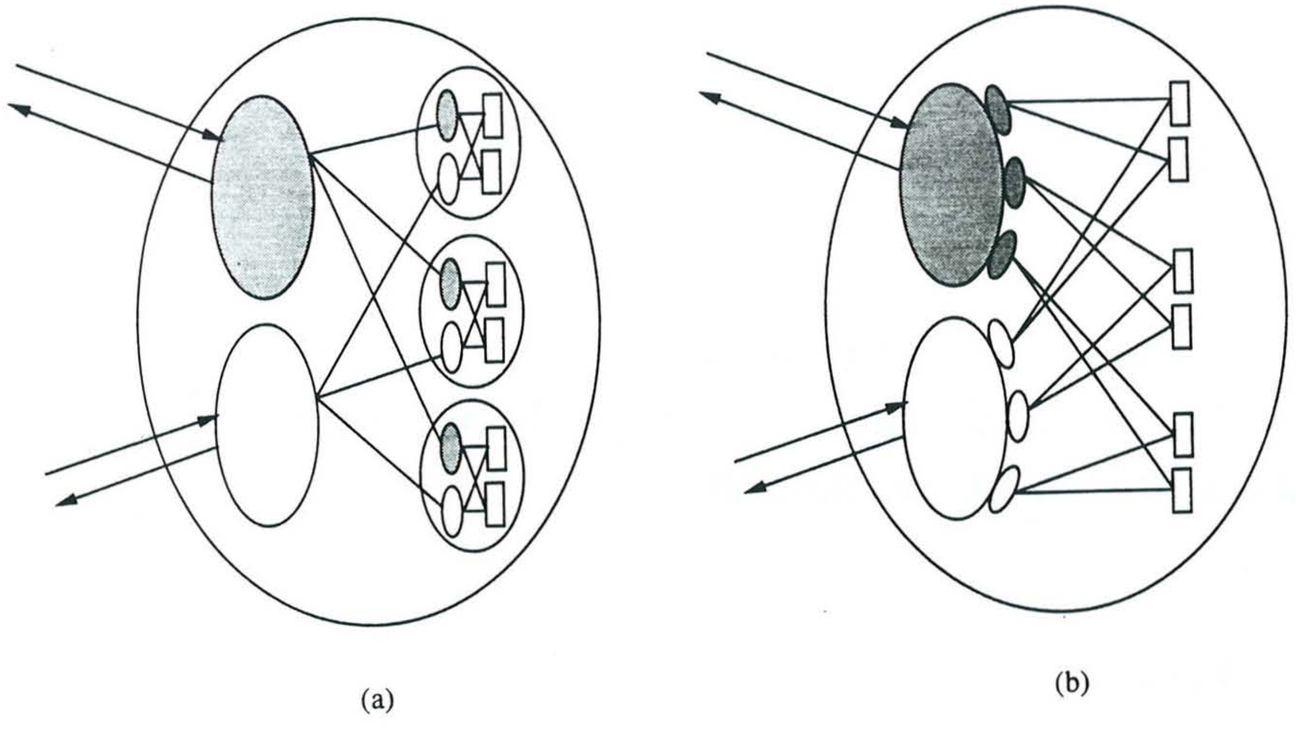
Moreover, if α is a fair execution of T,
 if only ports in I have $stop$ events in α,
 and if the atomic objects guarantee I-failure termination,
then α' is a fair execution of S.

Corollary:
Suppose the substituted atomic objects guarantee wait-free termination.
Then for any fair execution α of T, there is a fair execution α' of S that is indistinguishable to all users, and that has the same failures.

Corollary:
Suppose S is itself an atomic object, and suppose S and all the substituted atomic objects guarantee I-failure termination.
Then T is an atomic objecct guaranteeing I-failure termination.

Hierarchical construction of shared-memory systems

If each substituted atomic object is itself a shared-memory system, then the transformed system T is also a shared-memory system.



Hierarchical construction of atomic objects.

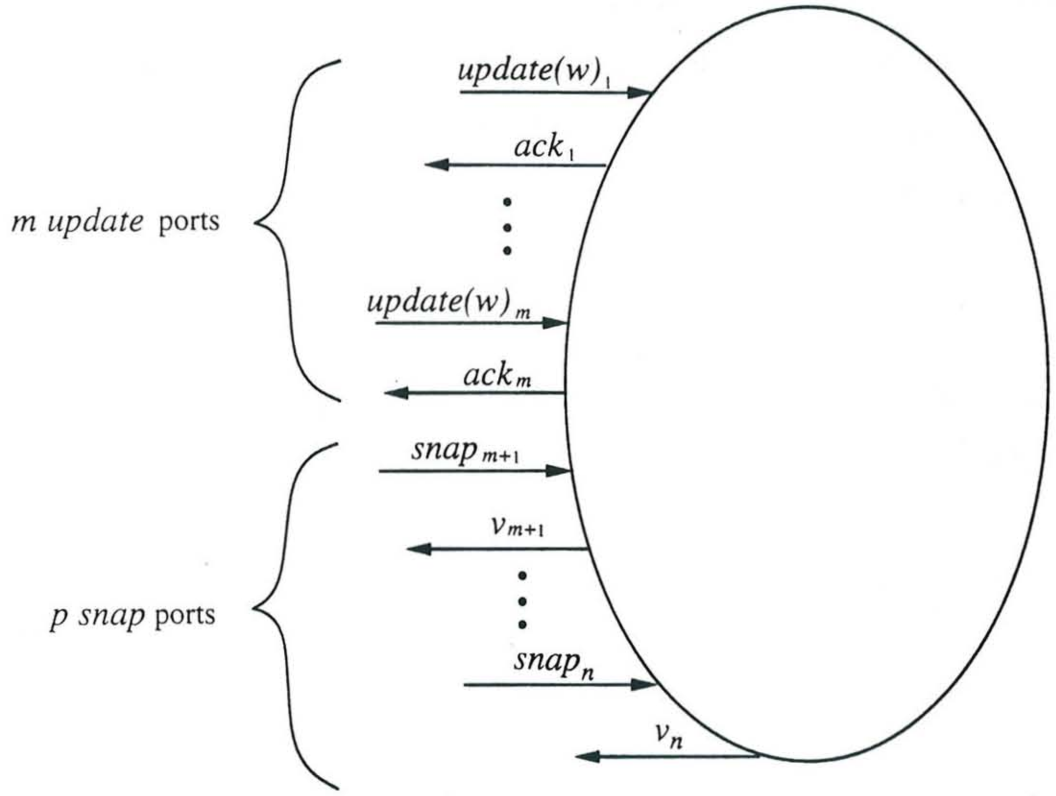
Sufficient condition for showing atomicity:

Lemma:
Suppose that A guarantees well-formedness, failure-free termination.
If every (finite or infinite) execution *containing no incomplete operations* satisfies the atomicity property, then the same is true for every execution, even those with incomplete operations.

Proof:
Uses the fact that atomicity plus well-formedness is a safety property.

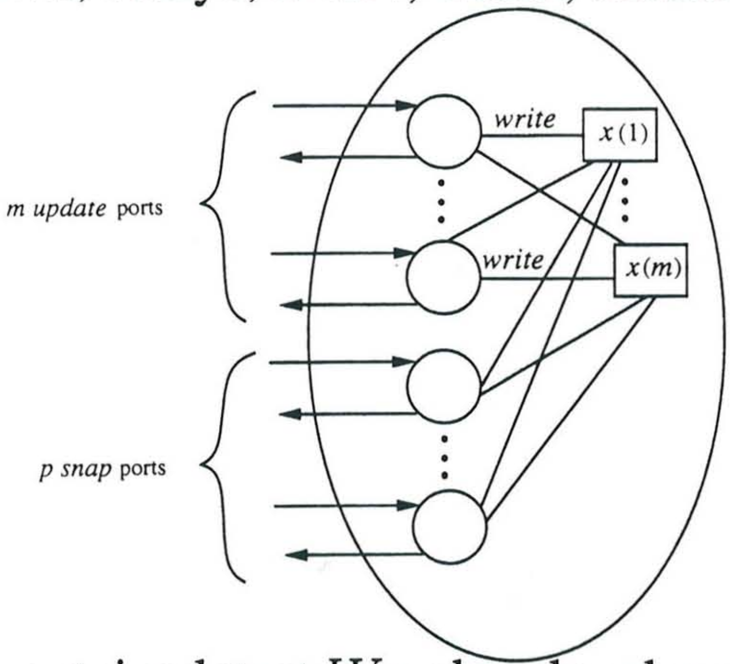
III. Example 1: Snapshot Atomic Objects

Variable type: Length m vectors over W.
  $update(i, w)$, component i, at port i
  $snap$, returns entire vector



Problem: Implement with shared memory system,
  using 1-writer/n-reader read/write registers.

Algorithm with Unbounded Variables:
[Afek, Attiya, Dolev, Gafni, Merritt, Shavit]



$x(i)$ contains latest W value, local sequence number, plus $view$, a vector.

$snap$:
Perform sets of $reads$ until either:
  Two consecutive sets are identical:
    Then return the common vector.
  For some i, four distinct sequence numbers are seen:
    Then return the $view$ from the third.

$update(w)_i$:
Do an $embedded\text{-}snap$.
Write w, sequence number, and snapped vector, to $x(i)$.

Wait-free.

Atomicity:

Consider complete operations only.

Serialization point for *update*: At *write*

Ser. pt. for *snap* (also *embedded-snap*):
  For those that find two identical sets of *reads*:
    Any point between the two sets.
  For those that borrow vectors from updates:
    Same as ser. pt. for update's *embedded-snap*.
    By induction on number of response events.

Every *snap*, *embedded-snap* returns the actual vector at its serialization point.
Prove by induction.

Algorithm with Bounded Variables: [Afek, Attiya, ...]

Unbounded sequence numbers used by *snaps* to
detect when new *update* operations have occurred.

Replace with mechanism based on handshake bits, toggle bits.

snap:
Similar to before, but:
- In each double-read attempt, first read others' handshake bits and set your own equal.
- In double read, check for identical handshake and toggle bits.

update:
Similar to before, but:
- First read all handshake bits.
- When write, set your own handshake bits unequal, and negate toggle bit.

Key: Mechanism correctly detects intervening updates.

IV. Example 2: Read/Write Atomic Objects

Problem: Implement multi-writer/multi-reader read/write atomic object using 1-writer/multi-reader.

Lemma: β, a (finite or infinite) sequence of external actions containing no incomplete ops, has atomicity property provided there is a partial ordering $<$ of ops satisfying:

1. No operation has infinitely many predecessors.
2. If one operation totally precedes another, then $<$ doesn't order in reverse.
3. $<$ orders all $WRITE$s with respect to all $READ$s and all $WRITE$s.
4. Values returned by $READ$s consistent with $<$.

Proof: By inserting serialization points.

Could generalize to other data types.

Algorithm with Unbounded Variables [Vitanyi, Awerbuch]:

m-writer/p-reader from 1-writer/1-reader
$x(i, j)$ readable by i, writable by j

$$
\begin{array}{ll|ccc|ccc}
 & & \multicolumn{3}{c|}{\overbrace{\qquad\qquad\qquad}^{\textit{WRITE}\text{ processes }\textit{write}}} & \multicolumn{3}{c}{\overbrace{\qquad\qquad\qquad}^{\textit{READ}\text{ processes }\textit{write}}} \\
 & & 1 & \cdots & m & m+1 & \cdots & n \\
\hline
\textit{WRITE}\text{ processes }\textit{read} & 1 & x(1,1) & \cdots & x(1,m) & x(1,m+1) & \cdots & x(1,n) \\
 & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
 & m & x(m,1) & \cdots & x(m,m) & x(m,m+1) & \cdots & x(m,n) \\
\hline
\textit{READ}\text{ processes }\textit{read} & m+1 & x(m+1,1) & \cdots & x(m+1,m) & x(m+1,m+1) & \cdots & x(m+1,n) \\
 & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
 & n & x(n,1) & \cdots & x(n,m) & x(n,m+1) & \cdots & x(n,n)
\end{array}
$$

Integer tags, ties broken by process index.

$WRITE(v)_i$:
Process i reads row i, finds greatest tag k.
Writes $(v, k+1, i)$ in column i.

$READ_i$:
Process i reads row i, finds latest triple.
Propagates it throughout column i.
Then returns latest value.

Wait-free.

Atomicity:
  Explicit construction of serialization points hard.
  Use partial order lemma.

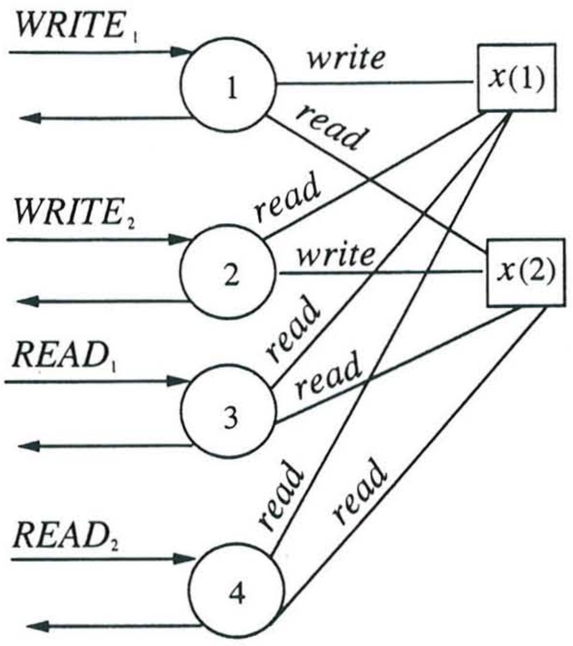
Every operation has unique pair (tag, index) that it writes.

Define $\pi < \phi$ if either:
1. $pair(\pi) < pair(\phi)$ (lexicographic), or
2. $pair(\pi) = pair(\phi)$, π is $WRITE$, ϕ is $READ$.

Satisfies 4 conditions.

Key: Consistency with external ordering of ops.
  Later op sees earlier op, chooses sufficiently large tag.

Bounded Algorithm for 2 Writers [Bloom]:



$x(i)$ holds value, Boolean tag

$WRITE(v)_i$:
  Read $x(\bar{i})$, finding tag b.
  Write value v, tag $b + i \bmod 2$.

$READ$:
  Read both registers, let b be sum of tags $\bmod 2$.
  Reread $x(1)$ if $b = 1$, $x(2)$ if $b = 0$, return value.

Wait-free.

Atomicity:

By abstraction mapping to unbounded version:
  Same, but uses integer tags.
  $WRITE$ writes value, tag $t + 1$, where t is the tag read.
  $READ$: Reads both. If tags within one, reread register
    with greater tag. Else reread either one.

Unbounded algorithm gives atomicity:
  Partial order argument.

Bounded algorithm gives atomicity:
  Forward simulation to unbounded algorithm.

The simulation:
Boolean tags in bounded algorithm are the *second lowest-order* bits of the binary representations of the integer tags in the unbounded algorithm.

V. Atomic Objects as Building Blocks

Shared memory systems:
  Hierarchical approach, already discussed.

Network systems:
  Powerful method.

Example: Simple (no-failure) simulation of shared-memory system in reliable asynchronous network:

Decompose into implementations of individual atomic objects, one per memory location.

Each variable x:
  Maintain at one process.
  Others send messages to read or write.
  Wait for reply.

For each x, the composition of local processes working on behalf of x, plus communication channels between them, is an atomic object for x.

More interesting example:

Fault-tolerant simulation of shared-memory systems.
[Attiya, Bar-Noy, Dolev; Lynch, Shvartsman]

Again, decompose into implementations of individual atomic objects, one per memory location.

Each variable x has copies everywhere.

```
$write(v)_i$:
  Send $query$ messages to everyone, wait for a majority.
  Obtain latest tag $t$.
  Send $propagate(v, t+1, i)$ messages to everyone,
    wait for majority.
  Everyone keeps latest (value, tag, index).
```

```
$read$:
  Send $query$ messages, wait for majority,
    obtain latest (value, tag, index).
  Propagate, wait for majority.
  Return value.
```

Correctness:

For each x, the composition of local processes working
  on behalf of x, plus communication channels between them,
  is an atomic object for x.

Guarantees f-failure termination, if $n > 2f$.

Use transformation theorem to get simulation.
  f-fault-tolerant simulation, $n > 2f$.

Other distributed algorithms implement atomic objects:
  Replicated state machine approach using logical time,
  Reconfiguration algorithms,
  ...

VI. Discussion

Summary:

- Gave definition of atomic objects,
- Basic properties:
  - Atomicity is a safety property.
  - Canonical automaton characterization
  - Composition
  - Substitution for shared variables
  - Hierarchical construction of shared memory systems
  - Can avoid considering incomplete operations.
- Algorithms for atomic snapshot, read/write objects
- Proof methods:
  - Explicitly defining serialization points
  - Showing object implements canonical automaton
  - Partial order method
  - Abstraction mappings (forward simulations)
- Applications:
  - Hierarchical construction of shared memory systems
  - Distributed simulation of shared memory
- Modular handling of fault-tolerance

To Do:

Extend partial order method to other data types.
Develop other proof methods.

Use methods to prove, decompose other distributed algorithms.

- Multi-writer from single-writer register algorithms, etc.
- Algorithms based on Isis-like primitives.
- ...

Extend theory/techniques to weaker memory coherence conditions:

- Sequential consistency
  - E.g., partial order technique used for Orca algorithms.
- Weaker conditions

DISCUSSION

Rapporteur: Dr Paul Ezhilchelvan

Lecture One

Professor Randell wanted the speaker to highlight on the extent of link between the implementors and the distributed algorithm developers, citing Professor Mehlhorn's (previous) work on LEDA as an example of such a link between implementation and the development of non-distributed algorithms. The speaker admitted that the link is weak at present but is growing; she recalled that the development and evolution of ISIS and Horus systems was influenced and guided very much by the theoretical knowledge of that time.

Professor Mehlhorn asked which one of the three models is closest to reality. The speaker replied that the answer has to be the asynchronous model; the other two models assume that the communication delays and relative processing speeds can be bounded and known; these bounds, however carefully estimated, may be violated at some point during system operation.

Lecture Two

When the speaker was questioned about the performance degradation that might result when bigger atomic object are constructed as an assembly of smaller atomic objects, she replied that performance degradation in such cases is the price paid for being able to prove the correctness of the system development. Professor Randell commented, and the speaker agreed, that the end-to-end argument can put a stop to building objects in that manner.

When the speaker was explaining the I/O interface of the atomic object, Professor Henderson obtained clarification that the alternating of input to and output from the atomic object is from user's point of view and not from the object's point of view.

When the speaker was describing the 'snap-shot' algorithm employed within the atomic object, Professor Hesselink observed that taking an embedded snapshot requires taking another snapshot and queried whether this recursion would make the algorithm non-terminating. The speaker replied that the algorithm is not recursive as it appears and that part of the algorithm is tricky to elaborate; she urged the audience to assume that the algorithm always terminates and to refer to her book for a detailed proof.

After the talk, Dr Andersson queried about the existing implementations of atomic objects; the speaker referred to the Transis work and commented that most implementations are proved to be correct. Professor Randell observed that quite a number of 'well-known' implementations are not completely correct to the specifications and some are correct with respect to 'untrue' fault assumptions. The speaker agreed that sufficient care should be taken to verify whether fault assumptions made are realistic.