Simulating Read / Write Objects

Can we provide a **shared read / write variable** in an asynchronous message-passing system, when processes can fail?

- Yes, if we have enough nonfaulty processes

Can we provide **stronger types** of read / write variables, when processes can fail?

- Yes, as long as we don’t read-**and**-write

Simulating Shared Memory

- Provide a **single-writer single-reader register** (this is the high-level) in a message-passing system
  - Accessed by **read** and **write** operations
- Underlying system is **asynchronous message passing** (this is the low-level), where less than half the processes can crash
Linearizability

Simulating Shared Memory w/ Failures

- Requires a majority of nonfaulty processes
- Otherwise, the system can be partitioned
  - A read “misses” the latest write
Must have \( n > 2f \)

**Theorem:** A simulation of a 1-reader, 1-writer read/write linearizable register in an asynchronous message passing tolerates at most \( f < n/2 \) crash failures.

**Proof:** Suppose in contradiction there is an algorithm tolerating \( f = n/2 \) crash failures.

Partition processes into two sets, \( Q_0 \) and \( Q_1 \), each of size \( f \).

Consider an execution in which
- initial value of simulated register is 0
- all processes in \( Q_1 \) crash initially
- process \( p_0 \) in \( Q_0 \) invokes `write(1)` at time 0 and no other operations are invoked
- the write completes at some time \( t_0 \) without any process in \( Q_0 \) receiving a message from any process in \( Q_1 \)
Must have $n > 2f$: Reading

Consider another execution in which

- initial value of simulated register is 0
- all processes in $Q_0$ crash initially
- process $p_1$ in $Q_1$ invokes a read at time $t_0+1$ and no other operations are invoked
- the read completes at some time $t_1$ without any process in $Q_1$ receiving a message from any process in $Q_0$
- the read returns 0, due to linearizability

Must have $n > 2f$: Arithmetic

Now paste the views of processes in $Q_0$ from the first execution with the views of processes in $Q_1$ from the second execution

- messages between $Q_0$ and $Q_1$ are delayed to arrive after time $t_1$

This execution is not linearizable, since read(0) follows write(1)

⇒ Must assume a majority of nonfaulty processes
The Algorithm in a Nutshell: Write

- The simulated register is replicated at each process

- Each data item has a unique sequence number
  - sequence of values

- `write(d, val, seq#)`
  - generate next sequence number
  - send a message with the value and the sequence number to all processes
  - each recipient updates its replica and sends ack
  - writer waits for n-f > n/2 acks
The Algorithm in Action: Write

The Algorithm in a Nutshell: Read

- Each data item has a unique sequence number
- `read(d)` returns `(val, seq#)`
  - send a request to all processes
  - each recipient sends back current value of its replica
  - wait for > n/2 replies
  - return value associated with largest sequence number
  - do a write-back to ensure atomicity of reads
The Algorithm in Action: Read

Key Idea for Correctness

- Each read should return the value of "the most recent" write
- Each read or write communicates with > n/2 processes
  - The set of processes communicating with a read intersects the set of processes communicating with a write

- Since system is asynchronous, a message on behalf of an operation might be overtaken by a message on behalf of a later operation
  - reader and writer keep track of "status" of each link
  - don't send a message on a link before receiving ack on previous message (ping-pong)
Proving Linearizability

Let \( \text{ts}(W) = \) sequence number of \( W \)
Let \( \text{ts}(R) = \) sequence number of write that \( R \) reads from

\( O_1 \rightarrow O_2 \) denotes \( O_1 \) completes before \( O_2 \) starts

Key lemmas:

- If \( W_1 \rightarrow W_2 \), then \( \text{ts}(W_1) < \text{ts}(W_2) \)
- If \( W \rightarrow R \), then \( \text{ts}(W) \leq \text{ts}(R) \)
- If \( R \rightarrow W \), then \( \text{ts}(R) < \text{ts}(W) \)
- If \( R_1 \rightarrow R_2 \), then \( \text{ts}(R_1) \leq \text{ts}(R_2) \)

Simulating R/W Registers from R/W Registers

- single-reader
  - single-writer
    - binary-valued
  - single-writer
    - multi-valued

- multi-reader
  - single-writer
    - multi-valued

- multi-reader
  - multi-writer
    - multi-valued

atomic snapshots
Atomic Snapshots

- m components
- **Update** a single component
- **Scan** all the components “at once” (atomically)

Provides an instantaneous view of the whole memory, very useful for designing shared-memory algorithms

Has a wait-free implementation from read/write variables

---

Atomic Snapshots: More Formally

Operations are
- invocation \texttt{scan}_i, returns V, where V is an array of n values
- invocation \texttt{update}(i,d) where d is a data value, returns ok

**Legal** sequences: if V is returned by a scan, V[k] is the parameter of latest preceding \texttt{update}(k,*)

For example:

\texttt{update}(1,x) \texttt{update}(2,y) \texttt{scan}([a,x,y]) \texttt{update}(0,z) \texttt{scan}([z,x,y])
Atomic Snapshots: Key Ideas

• Store each component in a separate variable
• To update: write to the respective variable
• To scan: Collect (read) values of the segments twice
  – If no segment is updated during the "double collect",
    this is a valid snapshot -- return it
• How to tell if a segment is updated?
  – Tag each value with a sequence number (1,2,3,...)

Atomic Snapshots: Partial Algorithm

Update(k,v)
A[k] = ⟨v, seq_i, i⟩

Scan()
repeat
read A[1],...,A[m]
read A[1],...,A[m]
if equal
return A[1,...,m]

Linearize:
• Updates with their writes
• Scans inside the double collects
Atomic Snapshot: Linearizability

Double collect (read a set of values twice)
If equal, there is no write between the collects
--- Assuming each write has a new value (seq#)

\[
\text{read } A[1],\ldots,A[m] \quad \text{read } A[1],\ldots,A[m] \\
\text{write } A[j]
\]

Creates a “safe zone”, where the scan can be linearized

---

Wait-free Atomic Snapshot

Embed a scan within the Update, and write its view to the segment
Scanner returns view obtained in last collect

\[
\text{Update (v, k)} \\
V = \text{scan} \\
A[k] = \langle v, \text{seq}_i, i, V \rangle
\]

---

Linearize:
- Updates with their writes
- Direct scans as before
- Borrowed scans with source

---

©Hagit Attiya 2018 (object simulations)
Atomic Snapshot: Borrowed Scans

Interference by process $p_j$
And another one...
$\Rightarrow p_j$ does a scan inbetween


write $A[j]$  \textcolor{red}{\textbf{embedded scan}}  write $A[j]$

Linearizing with the borrowed scan is OK.

Complexity of Atomic Snapshots

• Uses $O(m)$ read/write variables (some are large)
• Scan needs $O(n^2)$ reads and writes, why?
• Update needs $O(n^2)$ reads and writes
Simulating R/W Registers from R/W Registers

Multi-Valued From Binary

The simulated register takes values \{0,...,K-1\}

**Binary** approach: a different binary register stores each bit of the multi-valued register being simulated

- **Read algorithm** reads all registers and returns the resulting value
- **Write algorithm** writes the new bits in some order

**Errors when the reader overlaps a slow write and sees some new bits and some old bits**
A Unary Approach

Use an array of $K$ binary registers, $B[0..K-1]$

- value $v$ is represented with $B[v] = 1$ and other entries 0

- **Read** algorithm: read $B[0]$, $B[1]$, ..., until finding the first 1; return the index
- **Write** algorithm: set new entry of $B$ and zero the old entry of $B$

OK if reads and writes don't overlap.

When Reads and Writes Overlap...

**Problem:** reader may never find a 1 in $B$

**Solution:** write algorithm only clears (sets to 0) entries that are smaller than the entry that is set (to 1)
New-old inversion

Corrected Algorithm

**Read**: scans up to first 1, *then read down* to check those entries are still 0; return smallest index set during downward read

**Write(r)**: set r to 1 and then set to 0 entries smaller than r

Clearly, **wait-free**:
- writer does at most $K$ (primitive) writes
- reader does at most $2K-1$ (primitive) reads
Linearization Proof for Multi-Valued Construction

Fix an admissible execution of the algorithm
– Primitive operations (binary read / write) are atomic

We give a permutation of the (high-level) operations that is legal (by construction)
Show it respects real-time ordering of non-overlapping operations

Reads-From Relations

Primitive read r of a binary register B[v] reads from primitive write w to B[v] if w is the latest write to B[v] that precedes r in the execution
High-level read R reads from high-level write W if R returns v and W contains the primitive write that R's last primitive read of B[v] reads from
The Permutation

Primitive read $r$ of a binary register $B[v]$ reads from primitive write $w$ to $B[v]$ if $w$ is the latest write to $B[v]$ that precedes $r$ in the execution. High-level read $R$ reads from high-level write $W$ if $R$ returns $v$ and $W$ contains the primitive write that $R$'s last primitive read of $B[v]$ reads from:

- Place (high-level) writes in the order they occur
  - no concurrent writes
- Consider each (high-level) read $R$ in the order they occur
  - no concurrent reads
- If $R$ reads from write $W$, place $R$ immediately before the write that follows $W$ in the permutation

Legal by construction

Permutation Preserves Real-Time Order

- **write-write**: OK, by construction
- **read-write**: OK, since cannot read from a later write
- Two cases remain:
  - write-read
  - read-read

**Lemma:** Assume a high-level read $R$ returns $v$, and $R$ read of any $B[u], u < v$, during its upward scan, reads from a primitive write contained in high-level write $W$. Then $R$ does not read from a write that precedes $W$. 
Lemma: Assume a high-level read $R$ returns $v$, and $R$ read of any $B[u]$, $u < v$, during its upward scan, reads from a primitive write contained in high-level write $W$. Then $R$ does not read from a write that precedes $W$.
Simulating R/W Registers from R/W Registers

- Single-reader
- Single-writer
- Binary-valued
- Single-reader
- Single-writer
- Multi-valued
- Multi-reader
- Single-writer
- Multi-valued
- Atomic
- Snapshots
- Multi-reader
- Multi-writer
- Multi-valued

Multi-Reader from Single-Reader: 1st Attempt

**Use n single-reader registers**

**Write**: write new value in each Val[i] register

**Read**: return value from own Val[i] register

- **Pw**: write 1 to Val[1]
- **P1**: read 1 from Val[1]
- **P2**: read 0 from Val[2]

Note: new-old inversion
Readers Must Write

**Theorem:** In a wait-free simulation of a multi-reader single-writer register from single-reader single-writer registers, at least one reader writes

**Proof:** Suppose, in contradiction, there is an algorithm in which readers never write

- \( p_w \) is the writer, \( p_1 \) and \( p_2 \) are the readers
- initial value of simulated register is 0
- \( S_1 \) are the single-reader registers read by \( p_1 \)
- \( S_2 \) are the single-reader registers read by \( p_2 \)

Readers Must Write

- Consider execution in which \( p_w \) writes 1 to the simulated register, by a series of writes, \( w_1, \ldots, w_k \), to the single-reader registers.
  - Each of them is either in \( S_1 \) or in \( S_2 \) (but not both)
Readers Must Write

$v_j^i$ denotes the value returned if $p_i$ reads after $w_j$.

For each reader $p_i$ the value of the simulated register "switches" from 0 (old) to 1 (new), at some point:

- $v_1^1 = ... = v_{a-1}^1 = 0$, $v_a^1 = ... = v_k^1 = 1$
- $w_a$ is a write to a register in $S_1$
- $v_1^2 = v_2^2 = ... = v_{b-1}^2 = 0$, $v_b^2 = ... = v_k^2 = 1$
- $w_b$ is a write to a register in $S_2$

Assume $a < b$

Since readers do not write, they return the same values

⇒ old-new inversion, not linearizable
Corrected Multi-Reader Algorithm

In the simulated read, announce the value to be returned

Check values returned by previous reads

Sequence numbers allow to compare returned values

Writer's Algorithm

- get the next sequence# – an integer, incremented by 1 each time
- write (value, sequence#) to Val[1],...,Val[n] (one copy for each reader)
Reader $p_i$'s Algorithm

- read (value, sequence#)
  from Val[i]
- read (value, sequence#)
  from Report[j,i]
- pick pair with largest
  sequence#
- write that pair to row i of Report
- return value component of that pair

Correctness of Multi-Reader Algorithm

- Obviously wait-free
  - Write: $n$ primitive writes
  - Reade: $n+1$ primitive reads and $n$ primitive writes
- To prove linearizability, show a permutation of
  the high-level operations that is clearly legal
  and then prove it preserves real-time order of
  non-overlapping operations.
Constructing the Permutation

• Put in all writes in the order they occur in the execution
  – Single writer \( \Rightarrow \) writes do not overlap
• Consider the reads in the order of their responses in the execution
  – read \( R \) reads from write \( W \) if \( W \) generates the sequence\# associated with the value \( R \) returns
  – place \( R \) immediately before the write that follows \( W \)
• By construction, the permutation is legal.

Preserving Real-Time Order

• **write-write**: by construction
• **read-write**: \( R \) precedes \( W \) in the execution. Then \( R \) cannot read from \( W \) or any later write.
  \( \Rightarrow R \) is placed before \( W \) in the permutation
• **write-read**: \( W \) precedes \( R \) in the execution. Then \( R \) reads \( W \)'s sequence\# or a larger one from \( \text{Val}[\ ] \) and reads from \( W \) or a later write.
  \( \Rightarrow R \) is placed after \( W \) in the permutation.
• **read-read**: \( R_i \) by \( p_i \) precedes \( R_j \) by \( p_j \) in the execution. Then \( p_j \) reads \( R_i \)'s sequence\# or a larger one from \( \text{Report}[i,j] \). So \( R_j \) reads from the same write that \( R_i \) reads from or a later write
  \( \Rightarrow R_j \) is placed after \( R_i \) in the permutation.
Simulating R/W Registers from R/W Registers

Multi-Writer from Single-Writer: Key Ideas

- Each writer announces each value it wants to write to all the readers, by writing the value to its own (single-writer multi-reader) register
- Each reader reads all the values written by the writers and returns the latest one
- How to determine latest value?
  - use timestamps (as in Bakery algorithm)
  - since multiple processes generate timestamps, need to coordinate timestamp generation
Multi-Writer from Single-Writer

- Wait-free by construction

Create linearization:
- Place writes in timestamp order
- Insert each read before the write following the write it returns

Add logical time to values

\[
\text{Write}(v, X) \quad \text{read } TS_1, \ldots, \text{read } TS_n \\
TS_i = \max TS_j + 1 \\
\text{write } (v, TS_i, i) \text{ to } R_i
\]

Read(X)

\[
\text{read } R_1, \ldots, \text{read } R_n \\
\text{return } v_j \text{ with maximal } <TS_j, j>
\]

Multi-Writer from Single-Writer

- Wait-free by construction

Create linearization:
- Place writes in timestamp order
- Insert each read before the write following the write it returns

- Legality is immediate
- Real-time order is preserved since a read returns a value (with timestamp) larger than all preceding operations