Atomic and immediate snapshots

MPRI, P1, 2020

© P. Kuznetsov
The space of registers

- Nb of writers and readers: from 1W1R to NWNR
- Size of the value set: from binary to multi-valued
- Safety properties: safe, regular, atomic

All registers are (computationally) equivalent!
Transformations

From 1W1R binary safe to 1WNR multi-valued atomic

I. From safe to regular (1W1R)
II. From one-reader to multiple-reader (regular binary or multi-valued)
III. From binary to multi-valued (1WNR regular)
IV. From regular to atomic (1W1R)
V. From 1W1R to 1WNR (multi-valued atomic)
VI. From 1WNR to NWNR (multi-valued atomic)
VII. From safe bit to atomic bit (optimal, coming later)
This class

- Atomic snapshot: reading multiple locations atomically
  ✓ Write to one, read all
Atomic snapshot: sequential specification

- Each process $p_i$ is provided with operations:
  - $\text{update}_i(v)$, returns ok
  - $\text{snapshot}_i()$, returns $[v_1,\ldots,v_N]$

- In a **sequential** execution:
  For each $[v_1,\ldots,v_N]$ returned by $\text{snapshot}_i()$, $v_j$ ($j=1,\ldots,N$) is the argument of the last $\text{update}_j(.)$ (or the initial value if no such update)
Snapshot for free?

Code for process $p_i$:

**initially:**
shared 1WNR *atomic* register $R_i := 0$

**upon snapshot()**
\[
[x_1, \ldots, x_N] := \text{scan}(R_1, \ldots, R_N) \quad /*\text{read } R_1, \ldots R_N*/
\]
return $[x_1, \ldots, x_N]$

**upon update$_i(v)$**
$R_i.write(v)$
Snapshot for free?

- $\text{update}_1(1)$ ok
- $\text{update}_1(2)$ ok
- $\text{snapshot}()$ [1,1,2]
- $\text{update}_2(1)$ ok
- $\text{read}_1()1$, $\text{read}_2()1$, $\text{read}_3()2$
- $\text{update}_3(1)$ ok
- $\text{update}_3(2)$ ok
Snapshot for free?

$p1$
- $update_1(1)$ ok

$p2$
- $update_2(1)$ ok
- $read_1()1$
- $read_2()1$
- $read_3()2$
- $snapshot()$ [1,1,1]

$p3$
- $update_3(1)$ ok
- $update_3(2)$ ok
- $read_1()1$
- [2,1,1]
- [2,1,2]
▪ What about 2 processes?

▪ What about lock-free snapshots?
  ✓ At least one correct process makes progress (completes infinitely many operations)
Lock-free snapshot

Code for process $p_i$ (all written values, including the initial one, are unique, e.g., equipped with a sequence number)

Initially:

shared 1W1R atomic register $R_i := 0$

upon snapshot()

$[x_1,\ldots,x_N] := \text{scan}(R_1,\ldots,R_N)$

repeat

$[y_1,\ldots,y_N] := [x_1,\ldots,x_N]$

$[x_1,\ldots,x_N] := \text{scan}(R_1,\ldots,R_N)$

until $[y_1,\ldots,y_N] = [x_1,\ldots,x_N]$

return $[x_1,\ldots,x_N]$

upon update$_i(v)$

$R_i.write(v)$
Assign a linearization point to each operation

- \( \text{update}_i(v) \)
  - ✔ \( R_i.\text{write}(v) \) if present
  - ✔ Otherwise remove the op

- \( \text{snapshot}_i() \)
  - ✔ if complete – any point between identical scans
  - ✔ Otherwise remove the op

Build a sequential history \( S \) in the order of linearization points
Correctness: linearizability

S is legal: every snapshot\(_i()\) returns the last written value for every \(p_j\)

Suppose not: snapshot\(_i()\) returns \([x_1,\ldots,x_N]\) and some \(x_j\) is not the argument of the last update\(_j(v)\) in S preceding snapshot\(_i()\)

Let \(C_1\) and \(C_2\) be two scans that returned \([x_1,\ldots,x_N]\)

Read\(_j()\) \(x_j\) Returns the argument of the last update\(_j(.)\)!

C\(_1\) No update\(_j(.)\) linearized here!

C\(_2\)
Correctness: lock-freedom

An update \( i() \) operation is wait-free (returns in a finite number of steps)

Suppose process \( p_i \) executing snapshot \( i() \) eventually runs in isolation (no process takes steps concurrently)

- All scans received by \( p_i \) are distinct
- At least one process performs an update between two collect
- There are only finitely many processes \( \Rightarrow \) at least one process executes infinitely many updates
General case: helping?

What if an update interferes with a snapshot?

- Make the update do the work!

upon snapshot()

\[
\begin{align*}
[x_1, \ldots, x_N] &:= \text{scan}(R_1, \ldots, R_N) \\
[y_1, \ldots, y_N] &:= \text{scan}(R_1, \ldots, R_N) \\
\text{if } [y_1, \ldots, y_N] &= [x_1, \ldots, x_N] \text{ then} &
\quad \text{return } [x_1, \ldots, x_N] \\
\text{else} &
\quad \text{let } j \text{ be such that } x_j \neq y_j \text{ and } x_j = (u, U) \\
&
\quad \text{return } U
\end{align*}
\]

upon update_i(v)

\[
\begin{align*}
S &:= \text{snapshot()} \\
R_i.\text{write}(v, S)
\end{align*}
\]

If two scans differ - some update succeeded!

Would this work?
General case: wait-free atomic snapshot

upon snapshot()

\[ x_1, ..., x_n \] := \text{scan}(R_1, ..., R_N) \\

while true do \\
  \[ y_1, ..., y_n \] := \[ x_1, ..., x_n \] \\
  \[ x_1, ..., x_n \] := \text{scan}(R_1, ..., R_N) \\
  \text{if } \[ y_1, ..., y_n \] = \[ x_1, ..., x_n \] \text{ then} \\
  \quad \text{return } \[ x_1, ..., x_n \] \\
  \text{else if } \text{moved}_j \text{ and } x_j \neq y_j \text{ then} \\
  \quad \text{let } x_j = (u, U) \\
  \quad \text{return } U \\
  \text{for each } j: \text{moved}_j := \text{moved}_j \lor x_j \neq y_j

upon update_i(v)

\[ S := \text{snapshot}() \]
\[ R_i.\text{write}(v, S) \]

If a process moved twice: its last snapshot is valid!
Correctness: wait-freedom

Claim 1 Every operation (update or snapshot) returns in $O(N^2)$ steps (bounded wait-freedom)

**snapshot**: does not return after a scan if a concurrent process moved and no process moved twice

- At most $N-1$ concurrent processes, thus (pigeonhole), after $N$ scans:
  - Either at least two consecutive identical scans
  - Or some process moved moved twice!

**update**: snapshot() + one more step
Correctness: linearization points

**update**$_i$(v): linearize at the R$_i$.write(v,S)

**complete snapshot()**

- If two identical scans: between the scans
- Otherwise, if returned U of p$_j$: at the linearization point of p$_j$’s snapshot
The linearization is:

- **Legal**: every snapshot operation returns the most recent value for each process
- **Consistent with the real-time order**: each linearization point is within the operation’s interval
- **Equivalent to the run** (locally indistinguishable)
One-shot atomic snapshot (AS)

Each process $p_i$:

update$_{i}(v_i)$

$S_i := \text{snapshot()}$

$S_i = S_i[1], \ldots, S_i[N]$ (one position per process)

Vectors $S_i$ satisfy:

§ **Self-inclusion**: for all $i$: $v_i$ is in $S_i$

§ **Containment**: for all $i$ and $j$: $S_i$ is subset of $S_j$ or $S_j$ is subset of $S_i$
“Unbalanced” snapshots

$p_1$ sees $p_2$ but misses its snapshot

update$_1$(1) ok
snapshot() [1,1,0]

$p_1$

update$_2$(1) ok
snapshot() [1,1,1]

$p_2$

snapshot() [1,1,1]

$p_3$

update$_3$(1) ok
Enumerating possible runs: two processes

Each process $p_i$ ($i=1,2$):

update$_i(v_i)$

$S_i := \text{snapshot()}$

Three cases to consider:

(a) $p_1$ reads before $p_2$ writes

(b) $p_2$ reads before $p_1$ writes

(c) $p_1$ and $p_2$ go “lock-step”: first both write, then both read
One-shot atomic snapshot (AS)

Each process $p_i$:
update$_i(v_i)$
$S_i := \text{snapshot}()$

$S_i = S_i[1], \ldots, S_i[N]$ (one position per process)

Vectors $S_i$ satisfy:

§ Self-inclusion: for all $i$: $v_i$ is in $S_i$

§ Containment: for all $i$ and $j$: $S_i$ is subset of $S_j$ or $S_j$ is subset of $S_i$
Topological representation: one-shot AS

- $p_1$ sees $\{p_1, p_2\}$
- $p_2$ sees $\{p_2\}$
- $p_3$ sees $\{p_2, p_3\}$
- $p_1$ sees $\{p_1, p_2\}$
- $p_3$ sees $\{p_1, p_2, p_3\}$
- $p_2$ sees $\{p_2, p_3\}$
- $p_3$ sees $\{p_3\}$

Balanced run:
steps of $p_2$, then $p_1$, then $p_3$
Topological representation: one-shot AS

- $p_1$ sees $\{p_1, p_2\}$
- $p_2$ sees $\{p_1, p_2\}$
- $p_3$ sees $\{p_1, p_2\}$
- $p_2$ sees $\{p_1, p_2, p_3\}$
- $p_3$ sees $\{p_1, p_2, p_3\}$
- $p_3$ sees $\{p_2, p_3\}$
- $p_2$ sees $\{p_2, p_3\}$

"unbalanced" run
One-shot *immediate* snapshot (IS)

One operation: WriteRead(v)

Each process $p_i$:

$$S_i := \text{WriteRead}_i(v_i)$$

Vectors $S_1, \ldots, S_N$ satisfy:

- **Self-inclusion**: for all $i$: $v_i$ is in $S_i$
- **Containment**: for all $i$ and $j$: $S_i$ is subset of $S_j$ or $S_j$ is subset of $S_i$
- **Immediacy**: for all $i$ and $j$: if $v_i$ is in $S_j$, then $S_i$ is a subset of $S_j$
Topological representation: one-shot IS

$p_1$ sees $\{p_1, p_2\}$

$p_2$ sees $\{p_1, p_2\}$

$p_3$ sees $\{p_2, p_3\}$

$p_2$ sees $\{p_2, p_3\}$

A subdivision!
IS is equivalent to AS (one-shot)

- IS is a restriction of one-shot AS => IS is stronger than one-shot AS
  ✓ Every run of IS is a run of one-shot AS

- Show that a few (one-shot) AS objects can be used to implements IS
  ✓ One-shot ReadWrite() can be implemented using a series of update and snapshot operations
IS from AS

shared variables:
  $A_1, \ldots, A_N$ – atomic snapshot objects, initially $[T, \ldots, T]$

Upon $\text{WriteRead}_i(v_i)$

\[
\begin{align*}
r &:= N+1 \\
\text{while true do} \\
  r &:= r-1 \quad // \text{drop to the lower level} \\
  A_r.\text{update}_i(v_i) \\
  S &:= A_r.\text{snapshot}() \\
  \text{if } |S| = r \text{ then} & \quad // |S| \text{ is the number of non-}T \text{ values in } S \\
  \text{return } S
\end{align*}
\]
Drop levels: two processes, $N \geq 3$

- $N$
  - See $< N$

- $N-1$
  - See $< N-1$

- $\vdots$

- $2$
  - See $1$ or $2$

- $1$
  - See $1$
Correctness

The outcome of the algorithm satisfies Self-Inclusion, Snapshot, and Immediacy

- By induction on $N$: for all $N>1$, if the algorithm is correct for $N-1$, then it is correct for $N$

- Base case $N=1$: trivial
Correctness, contd.

- Suppose the algorithm is correct for N-1 processes
- N processes come to level N
  - At most N-1 go to level N-1 or lower
  - (At least one process returns in level N)
  - Why?
- Self-inclusion, Containment and Immediacy hold for all processes that return in levels N-1 or lower
- The processes returning at level N return all N values
  - The properties hold for all N processes! Why?
Iterated Immediate Snapshot (IIS)

Shared variables:
   IS_1, IS_2, IS_3,…  // a series of one-shot IS

Each process p_i with input v_i:
r := 0
while true do
   r := r+1
   v_i := IS_r.WriteRead_i(v_i)
Iterated standard chromatic subdivision (ISDS)
ISDS: one round of IIS
ISDS: two rounds of IIS
IIS is equivalent to (multi-shot) AS

- AS can be used to implement IIS (wait-free)
  - ✓ Multiple instances of the construction above (one per iteration)

- IIS can be used to implement multi-shot AS in the non-blocking manner:
  - ✓ At least one correct process performs infinitely many read or write operations
  - ✓ Good enough for protocols solving distributed tasks!
From IIS to AS

We simulate an execution of full-information protocol (FIP) in the AS model, i.e., each process $p_i$ runs:

\[
\text{state} := \text{input value of } p_i \\
\text{repeat} \\
\quad \text{update}_i(\text{state}) \\
\quad \text{state} := \text{snapshot()} \\
\text{until} \ \text{decided}(\text{state})
\]

(\text{the input value and the decision procedure depend on the problem being solved})

If a problem is solvable in AS, it is solvable with FIP

For simplicity, assume that the $k$-th written value $= k$ (\text{“without loss of generality”} – every written value is unique)
From IIS to AS: non-blocking simulation

Shared: \( IS_1, IS_2, ... \)  // an infinite sequence of one-shot IS memories

Local: at each process, \( c[1,...,N]=[(0,T),..., (0,T)] \)

Code for process \( p_i \):

\[
\begin{align*}
r &:= 0; \quad c[i].\text{clock} := 1; \quad \text{// } p_i \text{‘s initial value} \\
\text{repeat forever} & \\
\quad r &:= r + 1 \\
\quad \text{view} & := IS_r.UnRead(c) \quad \text{// get the view in IS}_r \\
\quad c & := \text{top(view)} \quad \text{// get the top clock values} \\
\quad \text{if } |c| = r & \text{ then } \quad \text{// the current snapshot completed} \\
\quad \quad \text{if } \text{undecided}(ctop) & \text{ then} \\
\quad \quad \quad \quad c[i].\text{val} := ctop; \\
\quad \quad \quad \quad c[i].\text{clock} := c[i].\text{clock} + 1 \quad \text{// update the clock} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{return decision}(ctop) \quad \text{// return the decision}
\end{align*}
\]
From IIS to AS

Each process $p_i$ maintains a vector clock $c[1,\ldots,N]$.

- Each $c[j]$ has two components:
  - $c[j].\text{clock}$: the number of updates of $p_j$ “witnessed” by $p_i$ (i.e., $c\.\text{clock}$ - the corresponding vector).
  - $c[j].\text{val}$: the most recent value of $p_j$’s vector clock “witnessed” by $p_i$ (i.e., $c\.\text{val}$ – the corresponding vector).

- To perform an update: increment $c[i].\text{clock}$ and set $c[i].\text{val}$ to be the “most recent” vector clock.

- To take a snapshot: go through iterated memories until $lcl = \sum_j c[j].\text{clock}$ is “large enough”,
  - i.e., $lcl = r$ (the current round number).
  - As we’ll see, $lcl \geq r$: every process $p_i$ begins with $c[i]=1$. 

• We say that $c \geq c'$ iff for all $j$, $c[j].clock \geq c'[j].clock$ (c observes a more recent state than c)
  ✓ Not always the case with c and c' of different processes

• $lcl = \sum_j c[j].clock$ (sum of clock values of the last seen values)

• For $c = c[1],…c[N]$ (vector of vectors $c[j]$), $\text{top}(c)$ is the vector of most recent seen values:

  $c[1] = [1 \hspace{1cm} 3 \hspace{1cm} 2]$
  $c[2] = [4 \hspace{1cm} 2 \hspace{1cm} 1]$
  $c[3] = [2 \hspace{1cm} 1 \hspace{1cm} 5]$

  $\text{top}(c) = [4 \hspace{1cm} 3 \hspace{1cm} 5]$
Let $c_r$ denote the vector evaluated by an undecided process $p_i$ in round $r$ (after computing the top function)

**Lemma 1** $|c_r| \geq r$

**Proof sketch**
$c_{r+1} \geq c_r$ (by the definition of top)

Initially $|c_1| \geq 1$ (each process writes $c[1].\text{clock}=1$ in IS$_1$)

Inductively, suppose $|c_r| \geq r$, for some round $r$:
- If $|c_r|=r$, then $c'$, such that $|c'|=r+1$, is written in IS$_{r+1}$
- If $|c_r|>r$, then $c'$, such that $c' \geq c_r$ (and thus $|c'| \geq |c_r|$) is written in in IS$_{r+1}$

In both cases, $c_{r+1} \geq r+1$
From IIS to AS: correctness

Lemma 2 Let $c_r$ and $c_r'$ be the clock vectors evaluated by processes $p_i$ and $p_j$, resp., in round $r$. Then $|c_r| \leq |c_r'|$ implies $c_r \leq c_r'$

Proof sketch
Let $S_i$ and $S_j$ be the outcomes of $IS_r$ received by $p_i$ and $p_j$

$c_r = \text{top}(S_i)$ and $c_r' = \text{top}(S_j)$

Either $S_i$ is a subset of $S_j$ or $S_j$ is a subset of $S_i$ (the Containment property of IS)

Suppose $S_i$ is a subset of $S_j$, then each clock value seen by $p_i$ is also seen by $p_j$

$\Rightarrow |c_r| \leq |c_r'|$ and $c_r \leq c_r'$
From IIS to AS: correctness

**Corollary 1** (to Lemma 2) All processes that complete a snapshot operation in round $r$, get the same clock vector $c$, $|c| = r$

**Corollary 2** (to Lemmas 1 and 2) If a process completes a snapshot operation in round $r$ with clock vector $c$, then for each clock vector $c'$ evaluated in round $r' \geq r$, we have $c \leq c'$
Lemma 3 Every execution’s history is linearizable (with respect to the AS spec.)

Proof sketch
Linearization
- Order snapshots based on the rounds in which they complete
- Put each update(c) just before the first snapshot that contains c (if no such snapshot – remove)

By Corollaries 1 and 2, snapshots and updates put in this order respect the specification of AS
From IIS to AS: liveness

Lemma 4 Some correct undecided process completes infinitely many snapshot operations (or every process decides).

Proof sketch

By Lemma 1, a correct process \( p_i \) does not complete its snapshot in round \( r \) only if \( |c_r| > r \)

Suppose \( p_i \) never completes its snapshot

\( \Rightarrow c_r \) keeps grows without bound and

\( \Rightarrow \) some process \( p_j \) keeps updating its \( c[j] \)

\( \Rightarrow \) some process \( p_j \) completes infinitely many snapshots
IIS=AS for wait-free task solutions

- Suppose we simulate a wait-free protocol for solving a task:
  ✓ Every process starts with an input
  ✓ Every process taking sufficiently many steps (of the full-information protocol) eventually decides (and thus stops writing new values, but keeps writing the last one)
  ✓ Outputs match inputs (we’ll see later how it is defined)

- If a task can be solved in AS, then it can be solved in IIS
  ✓ We consider IIS from this point on