Immediate Snapshot

CA 2017

Jingjing Wang, LPD, EPFL
A **snapshot** has two operations: `update()` and `scan()` and maintains an array `x` of size `n`.

**Sequential specification**

**scan()**:

- Return `(x)`

**update(i, v)**:

- `x[i] := v;`
- Return (OK)
Motivation for immediate snapshot

Snapshot

- Update some state
- Take a “picture” of all states
- Separately

Immediate snapshot

- Immediately take a “picture” of all states after updating a state
Semantics

The memory is accessed via a single \texttt{update\_snapshot} operation.

Semantics: each write operation, in addition to writing, also returns an atomic snapshot.

“Weakly atomic” = runs of standard atomic snapshot include runs of immediate snapshot.
The power of registers

Can immediate snapshot be implemented by atomic registers?

- Yes. At least for one-shot version

One-shot: Each process invokes at most once that operation
Immediate snapshot

An immediate snapshot has a single operation: `update_snapshot()` and maintains an array `x` of size `n`.

Sequential specification

`update_snapshot(vi)`:  
- `x[i] := vi;`
- Return `{(1, x[1]), (2, x[2]), …, (n, x[n])}`
Properties

Liveness. An invocation of `update_snapshot()` terminates

Self-inclusion. \((i, vi) \in viewi\)

Containment. \(viewi \subseteq viewj\) or \(viewj \subseteq viewi\)

Immediacy. If \((j, vj) \in viewi\), then \(viewj \subseteq viewi\)
Naive implementation

n processes share an atomic snapshot object x

\texttt{update\_snapshot(vi)}:

- \texttt{x.update(i, vi)};
- \texttt{a := x.scan();}
- Return \{(1, a[1]), (2, a[2]), \ldots, (n, a[n])\}
Immediacy?

$\text{update\_scan()} - \{(1, v1), (2, v2)\}$

$p_1$

$\text{update}_1(v_1) \quad \text{snapshot}_1()$

$p_2$

$\text{update}_2(v_2)$

$p_3$

$\text{update}_3(v_3) \quad \text{snapshot}_3()$
Immediacy?

update_scan()
- { (1, v1), (2, v2), (3, v3) }
Snapshot vs. immediate snapshot

An atomic snapshot

An immediate snapshot that satisfies

- Liveness, self-inclusion, containment, immediacy
Possible execution?

$$\{(1, v1), (2, v2)\}$$

$$\text{update\_snapshot}_1(v_1)$$

$$\{(1, v1), (2, v2)\}$$

$$\text{update\_snapshot}_2(v_2)$$

$$\{(1, v1), (2, v2), (3, v3)\}$$

$$\text{update\_snapshot}_3(v_3)$$
Possible execution?

{\[(1, v1), (2, v2)\]}

Liveness. ✓
Self-inclusion. ✓
Containment. ✓
Immediacy. ✓
A property that follows

(Self-inclusion. \((i, vi) \in \text{view}_i\)

+ Immediacy. If \((j, v_j) \in \text{view}_i\), then \(\text{view}_j \subseteq \text{view}_i\))

Property: If \((i, -) \in \text{view}_j\) and \((j, -) \in \text{view}_i\), then \(\text{view}_j = \text{view}_i\)

=> Compared with sequential execution?
Atomicity

Every operation appears to execute at

- Some indivisible point in time (called linearization point) between
- The invocation and reply time events
Atomic execution?

\{ (1, v1), (2, v2) \}

\text{update\_snapshot}_1(v_1)

\{ (1, v1), (2, v2) \}

\text{update\_snapshot}_2(v_2)

\{ (1, v1), (2, v2), (3, v3) \}

\text{update\_snapshot}_3(v_3)
Set linearizability

Linearization replaced by set-linearization:

- These invocations are set-linearized at the same point of the time line

For one-shot immediate snapshot,

- The invocations which are set-linearized at the same point do return the very same view
Key idea for set linearizability

To `update_scan()`, a process keeps reading other processes’ updates

For any two processes pi and pj,

- If pi and pj see each other's update, then pi and pj retry reading until they are going to return the same result
Enforcing set linearizability

The processes share an array of registers REG[1], REG[2], REG[3], …

- REG[x] is again an array of registers
- REG[x] contains a view
- REG[x][i] can only be written by pi

Pi reads REG[x]

- If pi cannot return REG[x], then pi retries, writes and reads the next REG
Enforcing set linearizability

The processes share an array of registers REG[1], REG[2], …, init’ed to ⊥

A recursive implementation:

- \textit{update\_snapshot}(vi):
  - my\_viewi := rec\_update\_snapshot(next, vi)
  - Return my\_viewi
Enforcing set linearizability

Every process keep a local array of registers Regi

- **rec_update_snapshot**(x, v):
  - REG[x][i].write(v);
  - For each j ∈ {1, ..., n} do Regi[j] := REG[x][j].read();
  - Viewi := { (j, Regi[j]) | Regi[j] ≠ ⊥ };
  - if( some condition ) then resi := viewi;
  - Else resi := rec_update_snapshot(next, v);
  - Return resi
### Possible execution?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>p2</td>
<td>v2</td>
<td>v2</td>
<td>v2</td>
<td></td>
</tr>
<tr>
<td>p3</td>
<td>v3</td>
<td>v3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key idea for liveness

If pi and pj see each other’s update, then pi and pj retry

- Pi is waiting for pj’s last-minute view
- So is pj
- Which view is the last one?
Key idea for liveness (cont’d)

Suppose: At most $x$ processes access $REG[x]$ (invariant)

If $pi$ sees $REG[x]$ contains exactly $x$ updates, then

- $pi$ is one of the last processes which accesses $REG[x]$
- Or linearized as such

```plaintext
REG[x][1].write
REG[x][2].write
REG[x][3].write
```
Key idea for liveness (cont’d)

Suppose: At most x processes access REG[x] (invariant)

If pi sees REG[x] contains exactly x updates, then

- pi is one of the **last** processes which accesses REG[x]
- Or linearized as such

If the invariant is true, then after pi, REG[x] remains the **same**.
Key idea for liveness (cont’d)

Suppose: At most x processes access REG[x] (invariant)

If pi sees REG[x] contains exactly x updates, then

- pi is one of the last processes which accesses REG[x]
- Or linearized as such

If the invariant is true, then after pi, REG[x] remains the same

- Pi can return REG[x]
- As well as other processes who see pi’s update
Key idea for set-linearizability & liveness

Recall that we consider one-shot version:

- Each process invokes at most once `update_scan()`
- This means at most n processes access the first REG
Key idea for set-linearizability & liveness

Recall that we consider one-shot version:

- Each process invokes at most once `update_scan()`
- This means at most n processes access the first $\text{REG} = \text{REG}[n]$

If some condition = a process’s view of $\text{REG}[n]$ contains n values, then

- Return $\text{REG}[n]$
- Otherwise, go to the next $\text{REG} = \text{REG}[n-1]$
Key idea for set-linearizability & liveness (cont’d)

The processes share an array of registers REG[n], REG[n-1], …, REG[1]

- Each contains a view

Claim:

(a) At most x processes can access REG[x]
(b) At least one process return REG[x]
Immediate snapshot implementation

- `update_snapshot(vi):`
  - `my_viewi := rec_update_snapshot(n, vi)`
  - Return `my_viewi`
Immediate snapshot implementation

The processes share an array of *registers* \(\text{REG}[1, \ldots, n]\), init’ed to \(\perp\).

Every process keeps a local array of registers \(\text{Reg}_i\)

- \(\text{rec_update_snapshot}(x, v)\):
  - \(\text{REG}[x][i].\text{write}(v)\);
  - For each \(j \in \{1, \ldots, n\}\) do \(\text{Reg}_i[j] := \text{REG}[x][j].\text{read}()\);
  - \(\text{View}_i := \{ (j, \text{Reg}_i[j]) \mid \text{Reg}_i[j] \neq \perp \}\);
  - If( \(|\text{view}_i| = x\) ) then \(\text{res}_i := \text{view}_i\);
  - Else \(\text{res}_i := \text{rec_update_snapshot}(x-1, v)\);
  - Return \(\text{res}_i\)
Possible return value?

\{ (1, v1), (2, v2) \}

update\_snapshot_1(v_1)

\{ (1, v1), (2, v2) \}

update\_snapshot_2(v_2)

\{ (1, v1), (2, v2), (3, v3) \}

update\_snapshot_3(v_3)
Possible execution?

<table>
<thead>
<tr>
<th></th>
<th>REG[n]</th>
<th>REG[n-1]</th>
<th>...</th>
<th>REG[3]</th>
<th>REG[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>v1</td>
<td>v1</td>
<td>...</td>
<td>v1</td>
<td>v1</td>
</tr>
<tr>
<td>p2</td>
<td>v2</td>
<td>v2</td>
<td>...</td>
<td>v2</td>
<td>v2</td>
</tr>
<tr>
<td>p3</td>
<td>v3</td>
<td>v3</td>
<td>...</td>
<td>v3</td>
<td></td>
</tr>
</tbody>
</table>
References
