Another Look at the Implementation of Read/write Registers in Crash-prone Asynchronous Message-Passing Systems

D. IMBS\textsuperscript{1} A. MOSTEFAOUI\textsuperscript{2} M. PERRIN\textsuperscript{2} M. RAYNAL\textsuperscript{3}

\textsuperscript{1}LIF, Université d’Aix-Marseille, France
\textsuperscript{2}LINA, Université de Nantes, France
\textsuperscript{3}IUF & IRISA, Université de Rennes, France & Dpt of Comp., Polytechnic University, Hong Kong
Table of contents

- Fundamental issues in distributed computing
- Atomic read/write register
- The SCD communication abstraction
- SCD-broadcast captures RW registers (Snapshot, ...)
- Conclusion
A glance at

Read/Write Registers
FUNDAMENTAL pbs of DC

- Communication
  - Reliable broadcast
  - Read/Write register

- Agreement

In the presence of adversaries such as Asynchrony, failures, mobility, etc.
What is a register?

- Something that can be
  - written (posted/marked) and
  - read (understood)

- Historical perspective:
  - One of the most ancient (3500 BC) ways to record history/information: Sumerian clay tablets
  - More recently (1936): Turing machine tape: the fundamental object of computing
On the many faces of registers

- **Capacity**: binary, bounded, unbounded
- **Access**: SWSR, SWMR, MRMR
- **Facing concurrency**
  - Safe register
  - Regular register
  - Atomic register

- From safe binary SWSR registers to atomic multivalued MWMR registers despite asynchrony and process crashes (Lamport 1986)

- In **sequential computing**: registers are universal objects (Turing, 1936)
Atomic read/write register

- Read and write operations appear as
  - ★ if they have been executed sequentially,
  - ★ and this sequence
    - ★ complies with real-time order
      ⇒ respects process order
    - ★ satisfies the seq spec of a register

- Non-deterministic behavior when concurrency

- Why atomicity is fundamental:

  Atomic objects compose for free!
Sequentially consistent read/write register

- Read and write operations appear as
  - if they have been executed sequentially,
  - and this sequence
    - not required to comply with real-time order
      - but respects process order
    - satisfies the seq spec of a register

- Non-deterministic behavior when concurrency

- Sequentially consistent objects do not compose for free!
Atomic read/write register: Example

Here $R = 1$
Here $R = 2$
Here $R = 3$

THE communication abstraction for read/write registers
Sequentially consistent read/write register: Example

Here $R = 1$

Here $R = 2$

Logical time line

THE communication abstraction for read/write registers
Process and basic communication model

- **Process** model:
  - $n$ sequential processes $p_1, \ldots, p_n$
  - asynchrony: unknown arbitrary speed

- **Communication** model:
  - complete point-to-point network
  - no bound on transfer delays (but finite)
  - reliable (no loss, creation, duplication, alteration)
  - point-to-point $\Rightarrow$ sender can be identified
  - channels: not required to be FIFO
• operations “send tag \((m)\) to \(p_j\)” and “receive ()”

• the macro-operation “broadcast tag \((m)\)” : shortcut for

\[
\text{for each } j \in \{1,\ldots, n\} \text{ send } m \text{ to } p_j \text{ end for}
\]
Process failure model

• Crash failure = unexpected halt
  ✤ A process executes correctly until it (possibly) crashes
  ✤ No recovery

• Model parameters $n$ and $t$
  ✤ $t$ = upper bound on the nb of faulty processes
  ✤ Upper bound on $t$: $t < n/2$ (Attiya, Bar Noy, Dolev 1995)
  ✤ Notation: $CAMP_{n,t}[\emptyset]$ and $CAMP_{n,t}[t < n/2]$

• Broadcast is not reliable
Implementing a register $REG$ in a MP system

Peer-to-peer system model
Each $p_i$ is both a client and a server

application processes

distributed shared memory abstraction
Classical implementations of an atomic register in crash-prone asynchronous message-passing systems
ABD95 algorithm: Dijkstra Prize 2011


  • Impossibility to know if a process has crashed or is only very slow
  
  • The problem can be solved iff $t < n/2$: proof based on indistinguishability argument

  • Typical algorithm:
    
    ★ Sequence numbers
    ★ Notion of intersecting quorums
    ★ Notion of requests and acknowledgments
      – Write copies of a majority of processes
      – Read copies from a majority of processes
A few other algorithms

Aim of the paper
Objects to be built

- basic model $\text{CARW}_{n,t}[\emptyset]$: Atomic read/write registers
- **Snapshot objects** (can be built in $\text{CARW}_{n,t}[\emptyset]$)
  - array $\text{REG}[1..m]$ of atomic read/write registers with two operations, write() and snapshot()
  - **MWMR snapshot**
    - write$(r, v)$ assigns $v$ to $\text{REG}[r]$
    - snapshot() returns the value of the full array as if the operation had been executed instantaneously
  - **SWMR snapshot:**
    - $m = n$ and
    - $r = i$ for write$(r, v)$ by $p_i$
Answer the question

Which is the communication abstraction that matches RW registers and snapshot objects? and also help solve other problems...
More precisely

<table>
<thead>
<tr>
<th>Concurrent object</th>
<th>Communication abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causal read/write registers</td>
<td>Causal msg delivery</td>
</tr>
<tr>
<td>Consensus</td>
<td>Total order broadcast</td>
</tr>
<tr>
<td>Snapshot object (R/W reg.)</td>
<td>SCD-broadcast</td>
</tr>
</tbody>
</table>
The SCD-Broadcast abstraction: definition (1)

SCD = Set-Constrained Delivery

- Two operations:
  - \texttt{scd\_broadcast}(m): broadcasts a message \(m\)
  - \texttt{scd\_deliver}(): returns a non-\(\emptyset\) set of messages

- Five properties:
  - \textbf{Validity}:
    If a process scd-delivers a set containing a message \(m\), then \(m\) was scd-broadcast by some process
  - \textbf{Integrity}:
    A msg is scd-delivered at most once by each process
The SCD-Broadcast abstraction: definition (2)

- **MS-Ordering:**
  A process \( p_i \) scd-delivers a message set \( ms_i \) containing a message \( m \) and later a message set \( ms'_i \) containing a message \( m' \)

\[ \Rightarrow \]

no process scd-delivers first a message set \( ms'_j \) containing \( m' \) and later a message \( ms_j \) containing \( m \)

- **Termination-1:**
  If a non-faulty process scd-broadcasts a message \( m \), it terminates its scd-broadcast invocation and scd-delivers a message set containing \( m \)

- **Termination-2:**
  If a process scd-delivers a message \( m \), every non-faulty process scd-delivers a message set containing \( m \)
The SCD-Broadcast abstraction: PROPERTIES (1)

If each message set contains a single message

- Validity + Integrity + Termination-1 + Termination-2

= Uniform Reliable Broadcast
The SCD-Broadcast abstraction: PROPERTIES (2)

A containment property

- let $ms_i^l = \ell$-th message set scd-delivered by $p_i$
- at some time: $p_i$ scd-delivered the sequence of message sets $ms_i^1, \ldots, ms_i^x$
- let $MS_i^x = ms_i^1 \cup \cdots \cup ms_i^x$
- let $MS_j^y = ms_j^1 \cup \cdots \cup ms_j^y$
- $\forall i, j, x, y: (MS_i^x \subseteq MS_j^y) \lor (MS_j^y \subseteq MS_i^x)$
The SCD-Broadcast abstraction: PROPERTIES (3)

Graph interpretation

- Local scd-delivery order: \( m \rightsquigarrow_i m' \)
  - \( p_i \) scd-delivers a set containing \( m \)
  - before a set containing \( m' \)

- Global scd-delivery order: \( \rightsquigarrow = \bigcup_{1 \leq i \leq n} \rightsquigarrow_i \)
  \( \rightsquigarrow \) is partial order (no cycle)
  (useful to understand and proofs)
From SCD-Broadcast to MWMR Snapshot

Building a snapshot object in $\text{CAMP}_{n,t}[\text{SCD-broadcast}]$
Local representation of the snapshot object $REG$

- $reg_i[1..m]$: current value of $REG[1..m]$, as known by $p_i$
- $done_i$: Boolean variable
- $tsa_i[1..m]$: array of timestamps associated with the values stored in $reg_i[1..m]$

  $\star tsa_i[j].date$ and $tsa_i[j].proc$ (timestamp of $reg_i[j]$)

- Lexicographical total order $<_{ts}$:

  $\star ts1 = \langle h1, i1 \rangle$ and $ts2 = \langle h2, i2 \rangle$

  $\star ts1 <_{ts} ts2 \text{ def } (h1 < h2) \lor ((h1 = h2) \land (i1 < i2))$
Algorithm: snapshot operation

operation snapshot() by \( p_i \) is
\[
\begin{align*}
\text{done}_i & \leftarrow \text{false}; \\
\text{scd}_i \text{broadcast} \ \text{SYNC} \ (i); \\
\text{wait} (\text{done}_i); & \quad \% \text{end of synchronization} \\
\text{return}(\text{reg}_i[1..m]). \\
\end{align*}
\]

- \text{SYNC} \ (i) synchronization message
- allows \( p_i \) to obtain an atomic value of \( REG[1..m] \)
Algorithm: write operation

```
operation write(r, v) by p_i is
    done_i ← false;
    scd_broadcast SYNC (i);
    wait(done_i);  % end of synchronization 1
    done_i ← false;
    scd_broadcast WRITE (r, v, ⟨tsa_i[r].date + 1, i⟩);
    wait(done_i).  % end of synchronization 2
```
Algorithm: snapshot operation

when the message set
\{WRITE(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1 \rangle), \ldots, WRITE(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x \rangle),
SYNC(j_{x+1}), \ldots, SYNC(j_{y}) \} is scd-delivered do

for each \( r \) such that \( WRITE(r, -, -) \in \) the message set do

let \( \langle date, writer \rangle = \) greatest timestamp in \( WRITE(r, -, -) \);

if \( (tsa_i[r] < ts \langle date, writer \rangle) \)

then let \( v \) the value in \( WRITE(r, -, \langle date, writer \rangle) \);

\hspace{1cm} reg_i[r] \leftarrow v; tsa_i[r] \leftarrow \langle date, writer \rangle

end if

end for;

if \( \exists \ell : j_\ell = i \) then done_i \leftarrow true end if.

Observation: no quorum at this abstraction level!
The case of a sequentially consistent snapshot object

Suppress the messages **SYNC**!

These messages ensure compliance wrt real-time operation.

**operation snapshot() by** $p_i$ **is**

```
return(reg_i[1..m]).
```

**operation write($r, v$) by** $p_i$ **is**

```
done_i ← false;
scd广播 WRITE ($r, v, \langle tsa_i[r].date + 1, i \rangle$);
wait(done_i).
```

**when the message set**

\{WRITE($r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1 \rangle$), ..., WRITE($r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x \rangle$)\}

**is scd-delivered do** ..........

---

THE communication abstraction for read/write registers
From MWMR Snapshot to SCD-Broadcast

Building SCD-Broadcast in $\text{CARW}_{n,t}[\text{Snapshot}]$ ($\text{CARW}_{n,t}[\emptyset]$)
Shared objects

\( \varepsilon \): empty sequence
\( \oplus \): concatenation

- **SENT[1..n]**: snapshot object, initialized to \([\emptyset, \cdots, \emptyset]\)
  \(\text{SENT}[i] = \) messages scd-broadcast by \(p_i\)

- **SETS_SEQ[1..n]**: snapshot object, initialized to \([\varepsilon, \cdots, \varepsilon]\)
  \(\text{SETS_SEQ}[i] = \) seq. of msg sets scd-delivered by \(p_i\)
Local objects

- $sent_i$: local copy of the snapshot object $SENT$
- $sets\_seq_i$: local copy of the snapshot object $SETS\_SEQ$.
- $to\_deliver_i$: set whose aim is to contain the next message set that $p_i$ has to scd-deliver.
- $\text{members}(\text{set}\_seq)$ returns the set of messages in $set\_seq$
operation scd_broadcast($m$) by $p_i$ is

\[ sent_i[i] \leftarrow sent_i[i] \cup \{m\}; \text{SENT}.write(sent_i[i]); \text{progress}(). \]

background task $T$ is

\[ \text{repeat forever} \text{ progress()} \text{ end repeat}. \]
Algorithm (2)

procedure progress() by $p_i$ is
  enter_mutex();
  catch_up();
  $sent_i \leftarrow SENT$.snapshot();
  $to\_deliver_i \leftarrow (\cup_{1 \leq j \leq n} sent_i[j]) \setminus$ members($sets\_seq_i[i]$);
  if ($to\_deliver_i \neq \emptyset$)
    then $sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$;
      $SETS\_SEQ$.write($i, sets\_seq_i[i]$);
      scd_deliver($to\_deliver_i$)
  end if;
  exit_mutex().
Algorithm (3)

procedure catch_up() by $p_i$ is
  $sets_{seq_i} \leftarrow SETS\_SEQ$.snapshot();
  while
    $(\exists j, set : set$ first set in $sets_{seq_i}[j] \land set \not\subseteq$ members($sets_{seq_i}[i]$))
    do to_deliver$_i \leftarrow set \setminus$ members($sets_{seq_i}[i]$);
    $sets_{seq_i}[i] \leftarrow sets_{seq_i}[i] \oplus$ to_deliver$_i$;
    $SETS\_SEQ$.write($i, sets_{seq_i}[i]$);
    scd_deliver(to_deliver$_i$)
  end while.

THE communication abstraction for read/write registers
From sequentially consistency to atomicity

From non-composable to composable snapshot objects

The power of the messages $\text{SYNC}()$ (real-time compliance)

- Start from a sequentially consistent snapshot object ($\text{CARW}_{n,t}[\text{Snapshot}]$)
- Build SCD-Broadcast on top of it
  we obtain $\text{CAMP}_{n,t}[\text{SCD-broadcast}]$
- Build atomic snapshot on top of $\text{CAMP}_{n,t}[\text{SCD-broadcast}]$

First (?) systematic construction from SC to Atomicity
On the implementation of SCD-Broadcast
Implementing SCD in \textit{CAMP}\{t < n/2\}

\begin{verbatim}
operation scd_broadcast(m) is
    forward(m, i, sn_i, i, sn_i);
    wait(\(\not\exists\) msg \(\in\) buffer\(_i\) : msg.sd = i).

when the message forward(m, sd, sn_{sd}, f, sn_{f}) is fifo-delivered do \(\%\) from \(p_f\)
    forward(m, sd, sn_{sd}, f, sn_{f});
    try_deliver().

procedure forward(m, sd, sn_{sd}, f, sn_{f}) is
    if (sn_{sd} > clock\(_i\)[sd])
        then if (\(\exists\) msg \(\in\) buffer\(_i\) : msg.sd = sd \& msg.sn = sn_{sd})
            then msg.cl[f] \(\leftarrow\) sn_{f}
            else threshold[1..n] \(\leftarrow\) [\(\infty\), \ldots, \(\infty\)]; threshold[f] \(\leftarrow\) sn_{f};
                let msg \(\leftarrow\) \langle m, sd, sn_{sd}, threshold[1..n]\rangle;
                buffer\(_i\) \(\leftarrow\) buffer\(_i\) \cup\{msg\};
                fifo_broadcast forward(m, sd, sn_{sd}, i, sn_i);
                sn_i \(\leftarrow\) sn_i + 1
        end if
    end if
end if.
\end{verbatim}
Implementing SCD in $\mathcal{CAMP}\{t < n/2\}$ (Cont’d)

procedure try_deliver() is
    let $to\_deliver_i \leftarrow \{msg \in buffer_i : |\{f : msg.cl[f] < \infty\}| > \frac{n}{2}\}$;
    while ($\exists msg \in to\_deliver_i, msg' \in buffer_i \setminus to\_deliver_i : |\{f : msg.cl[f] < msg'.cl[f]\}| \leq \frac{n}{2}$) do
        $to\_deliver_i \leftarrow to\_deliver_i \setminus \{msg\}$
    end while;
    if ($to\_deliver_i \neq \emptyset$) then
        for each ($msg \in to\_deliver_i$ such that $clock_i[msg.sd] < msg.sn$) do
            $clock_i[msg.sd] \leftarrow msg.sn$
        end for;
        $buffer_i \leftarrow buffer_i \setminus to\_deliver_i$;
        $ms \leftarrow \{m : \exists msg \in to\_deliver_i : msg.m = m\}$; scd_deliver($ms$)
    end if.

• As $t < n/2$ is necessary and sufficient to build read/write registers in $\mathcal{CAMP}_{n,t}[\emptyset]$, it is also necessary ans sufficient to build SCD-broadcast in $\mathcal{CAMP}_{n,t}[\emptyset]$

• All the “technical details” are hidden in this algorithm which is designed and proved once for all!
Cost of SCD-broadcast implementation

- **Assumption:**
  - Let $\Delta =$ message delay
  - Local computation: zero cost

- **Cost:**
  - Time: $2\Delta$
  - Messages: $n^2$
Conclusion
To summarize

read/write or snapshot
shared memory
Seq consistent or atomic

SCD-Broadcast

Other applications:
lattice agreement, commutative operations, ...
Conceptual issues

- Better **understanding of basic mechanisms** needed to implement a read/write shared memory
- SCD-broadcast captures the "right" abstraction level
- **Simplicity** of the proposed (register/snapshot) algo.
- **Genericity** of the proposed algorithms wrt
  - read/write vs snapshot objects (same algorithms)
  - atomicity vs sequential consistency (SYNC msgs)
More important: He Told me

“Algorithms are at the core of Informatics”