Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits

Damien Imbs  Achour Mostefaoui  Michel Raynal
Matthieu Perrin

LIF  LS2N  IUF, IRISA
Université Aix-Marseille  Université de Nantes  Université de Rennes

Descartes Seminar
October 1st, 2018

From a paper published at ICDCN 2018
Hypothesis on the system

leader (+Ω)
- State machine replication \(\simeq\) Atomic broadcast
- Consensus

majority (+Σ)
- Atomic registers, sets, counters, key-value stores... \(\simeq\) ?
- Lattice agreement

asynchronous with crashes
- Weakly consistent CRDTs \(\simeq\) Causal/FIFO/(Uniform) Reliable broadcast
- Causal memory

Objects and tasks

Broadcast abstractions

A \(\simeq\) B: A can be implemented from B and reciprocally
Introduction – Equivalence Broadcast/Objects

Hypothesis on the system

leader (+Ω)

- State machine replication ≃ Atomic broadcast
- Consensus

majority (+Σ)

- Atomic registers, sets, counters, key-value stores...
- Lattice agreement

asynchronous with crashes

- Weakly consistent CRDTs
- Causal memory

Objects and tasks

Broadcast abstractions

\[ A \simeq B: A \text{ can be implemented from } B \text{ and reciprocally} \]
Introduction – Equivalence Broadcast/Objects

Hypothesis on the system

leader \((+\Omega)\)

- State machine replication \(\simeq\) Atomic broadcast
- Consensus

majority \((+\Sigma)\)

- Atomic registers, sets, counters, key-value stores... \(\simeq\) ?
- Lattice agreement

asynchronous with crashes

- Weakly consistent CRDTs \(\simeq\) Causal/FIFO/ (Uniform) Reliable broadcast
- Causal memory

Objects and tasks

Broadcast abstractions

\(A \simeq B\): A can be implemented from B and reciprocally
Introduction – Outline

1. Introduction
2. Consistency
3. Definition of SCD-Broadcast
   Intuition
   Definition
   Properties
4. Abstraction Power
   Sequentially consistent grow-only set
   Atomic grow-only set
   Sequentially consistent snapshot object
   Atomic snapshot object
5. Implementation and Computability Limits
   Computability limits
   Message-passing implementation
   Complexity
6. Perspectives
Consistency – Which histories are *correct*?

**Linearizable:**

\[ p \xrightarrow{\text{insert}(1)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1\}} \xrightarrow{\text{insert}(2)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1, 2\}} \xrightarrow{\text{read}\{1, 2\}} \xrightarrow{\perp} \]

\[ p_1 \xrightarrow{\text{insert}(1)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1\}} \xrightarrow{\text{read}\{1, 2\}} \]

\[ p_2 \xrightarrow{\text{insert}(2)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1, 2\}} \xrightarrow{\text{read}\{1, 2\}} \]

**Sequentially consistent:**

\[ p_1 \xrightarrow{\text{insert}(1)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1\}} \xrightarrow{\text{read}\{1, 2\}} \]

\[ p_2 \xrightarrow{\text{insert}(2)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1, 2\}} \xrightarrow{\text{read}\{1, 2\}} \]

**Causally consistent:**

\[ p_1 \xrightarrow{\text{insert}(1)} \xrightarrow{\perp} \xrightarrow{\text{read}\{1\}} \xrightarrow{\text{read}\{1, 2\}} \]

\[ p_2 \xrightarrow{\text{insert}(2)} \xrightarrow{\perp} \xrightarrow{\text{read}\{2\}} \xrightarrow{\text{read}\{1, 2\}} \]
Consistency – Which histories are correct?

Linearizable:

Sequentially consistent:

Causally consistent:
Consistency – Which histories are correct?

**Linearizable:**

\[ p \rightarrow \text{insert}(1) \downarrow \rightarrow \text{read} \rightarrow \{1\} \rightarrow \text{insert}(2) \downarrow \rightarrow \text{read} \rightarrow \{1, 2\} \rightarrow \text{read} \rightarrow \{1, 2\} \]

**Sequentially consistent:**

\[ p_1 \rightarrow \text{insert}(1) \downarrow \rightarrow \text{read} \rightarrow \{1\} \rightarrow \text{read} \rightarrow \{1, 2\} \rightarrow \text{read} \rightarrow \{1, 2\} \]

\[ p_2 \rightarrow \text{insert}(2) \downarrow \rightarrow \text{read} \rightarrow \{1, 2\} \rightarrow \text{read} \rightarrow \{1, 2\} \rightarrow \text{read} \rightarrow \{1, 2\} \]

**Causally consistent:**

\[ p_1 \rightarrow \text{insert}(1) \downarrow \rightarrow \text{read} \rightarrow \{1\} \rightarrow \text{read} \rightarrow \{1, 2\} \rightarrow \text{read} \rightarrow \{1, 2\} \]

\[ p_2 \rightarrow \text{insert}(2) \downarrow \rightarrow \text{read} \rightarrow \{2\} \rightarrow \text{read} \rightarrow \{1, 2\} \rightarrow \text{read} \rightarrow \{1, 2\} \]
Consistency – Which histories are correct?

Linearizable:

Sequentially consistent:

Causally consistent:
Consistency – Which histories are \textit{correct}?

\[ p \xrightarrow{\text{insert(1)}} \perp \xrightarrow{\text{read}} \{1\} \xrightarrow{\text{insert(2)}} \perp \xrightarrow{\text{read}} \{1, 2\} \xrightarrow{\text{read}} \{1, 2\}\]

**Linearizable:**

\[ p_1 \xrightarrow{\text{insert(1)}} \perp \xrightarrow{\text{read}} \{1\} \xrightarrow{\text{read}} \{1, 2\} \]
\[ p_2 \xrightarrow{\text{insert(2)}} \perp \xrightarrow{\text{read}} \{1, 2\} \xrightarrow{\text{read}} \{1, 2\}\]

**Sequentially consistent:**

\[ p_1 \xrightarrow{\text{insert(1)}} \perp \xrightarrow{\text{read}} \{1\} \xrightarrow{\text{read}} \{1, 2\} \]
\[ p_2 \xrightarrow{\text{insert(2)}} \perp \xrightarrow{\text{read}} \{1, 2\} \xrightarrow{\text{read}} \{1, 2\}\]

**Causally consistent:**

\[ p_1 \xrightarrow{\text{insert(1)}} \perp \xrightarrow{\text{read}} \{1\} \xrightarrow{\text{read}} \{1, 2\} \]
\[ p_2 \xrightarrow{\text{insert(2)}} \perp \xrightarrow{\text{read}} \{1, 2\} \xrightarrow{\text{read}} \{1, 2\}\]
Consistency – Sequential consistency

An execution is sequentially consistent if

There is a total order on all operations such that:

- The order is compliant with the order of each process
- The sequential specification is respected

Counter-example

Example
An execution is linearizable if

Same as sequential consistency, and:

- The order is compliant with real-time

**Counter-example**

\[ p_1 \quad \text{insert}(1) \quad \text{read} \triangleright \{1, 2\} \quad \text{read} \triangleright \{1, 2\} \]

\[ p_2 \quad \text{insert}(2) \quad \text{read} \triangleright \{2\} \quad \text{read} \triangleright \{1, 2\} \]

**Example**

\[ p_1 \quad \text{insert}(1) \quad \text{read} \triangleright \{1\} \quad \text{read} \triangleright \{1, 2\} \]

\[ p_2 \quad \text{insert}(2) \quad \text{read} \triangleright \{1, 2\} \quad \text{read} \triangleright \{1, 2\} \]
Definition – Intuition: a sequentially consistent set

Grow-only set (G-Set) object: operations

\( \text{insert}(v) \): adds \( v \in \mathbb{N} \) to the set

\( \text{read}() \): returns the full set

Algorithm with Atomic Broadcast

1. **operation** \( \text{read}() \): return \( \text{state} \);
2. **operation** \( \text{insert}(v) \): atomic-broadcast \( (I(v)) \); \( \text{wait} \) delivery;
3. **event** \( \text{atomic-deliver}(I(v)) \): \( \text{state} \leftarrow \text{state} \cup \{v\} \);

\[\begin{align*}
1: & \quad \text{insert}(1) \\
2: & \quad \text{read}() \triangleright \{1\} \\
3: & \quad \text{insert}(2) \\
4: & \quad \text{read}() \triangleright \{1, 2\} \\
5: & \quad \text{read}() \triangleright \{1, 2\}
\end{align*}\]
**Definition – Intuition:** a sequentially consistent set

Grow-only set (G-Set) object: operations

- **insert**($v$): adds $v \in \mathbb{N}$ to the set
- **read**: returns the full set

**Algorithm with FIFO Broadcast**

1. **operation** **read**(): return state;
2. **operation** **insert**($v$): fifo-broadcast ($I(v)$); **wait** delivery;
3. **event** fifo-deliver($I(v)$): state $\leftarrow$ state $\cup \{v\}$;
Definition – Intuition: a sequentially consistent set

Grow-only set (G-Set) object: operations

\( \text{insert}(v) \): adds \( v \in \mathbb{N} \) to the set
\( \text{read}() \): returns the full set

Algorithm with FIFO Broadcast

1. operation \( \text{read}() \): return \( \text{state} \);
2. operation \( \text{insert}(v) \): fifo-broadcast \( (I(v)) \); \text{wait} delivery;
3. event \( \text{fifo-deliver}(I(v)) \): \( \text{state} \leftarrow \text{state} \cup \{v\} \);

\[ \begin{align*}
1: & \quad \text{insert}(1) \\
2: & \quad \text{read}() \triangleright \{1\} \\
3: & \quad \text{insert}(2) \\
4: & \quad \text{read}() \triangleright \{1, 2\} \\
5: & \quad \text{read}() \triangleright \{1, 2\}
\end{align*} \]
**Definition** – **Intuition:** a sequentially consistent set

Grow-only set (G-Set) object: operations

- \(\text{insert}(v)\): adds \(v \in \mathbb{N}\) to the set
- \(\text{read}()\): returns the full set

**Algorithm with SCD Broadcast**

1. **operation** \(\text{read}()\): return \(\text{state}\);
2. **operation** \(\text{insert}(v)\): scd-broadcast \((I(v))\); **wait** delivery;
3. **event** scd-deliver\((\{I(v_1), \ldots, I(v_k)\})\): \(\text{state} \leftarrow \text{state} \cup \{v_1, \ldots, v_k\}\);

![Diagram showing the execution of the algorithm with SCD Broadcast](image)
Definition – Set-Constraint Delivery Broadcast

Interface

operation: scd-broadcast \((m)\) \hspace{1cm} event: scd-deliver \((mset)\)

Properties

Validity: \(p_i\) scd-delivers \(m \in mset\) \(\Rightarrow\) some \(p_j\) scd-broadcast \(m\)

Integrity: \(m\) is scd-delivered at most once by \(p_i\)

MS-Ordering: \(p_i\) scd-delivers \(m \in mset\) and later \(m' \in mset'\)

\(\Downarrow\)

impossible that
\(p_j\) scd-delivers \(m' \in mset'\) and later \(m \in mset\)

Termination-1: If a non-faulty \(p_i\) scd-broadcasts \(m\), it terminates its scd-broadcast invocation and scd-delivers \(m \in mset\)

Termination-2: \(p_i\) scd-delivers \(m\)

\(\Rightarrow\) every non-faulty \(p_j\) scd-delivers \(m \in mset\)
Definition – Set-Constraint Delivery Broadcast

Interface

operation: scd-broadcast \((m)\) \quad \text{event: scd-deliver} \((mset)\)

Properties

Validity: \(p_i\) scd-delivers \(m \in mset\) \(\Rightarrow\) some \(p_j\) scd-broadcast \(m\)

Integrity: \(m\) is scd-delivered at most once by \(p_i\)

MS-Ordering: \(p_i\) scd-delivers \(m \in mset_i\) and later \(m' \in mset'_i\)

\[
\downarrow
\]

impossible that \(p_j\) scd-delivers \(m' \in mset'_j\) and later \(m \in mset_j\)

Termination-1: If a non-faulty \(p_i\) scd-broadcasts \(m\), it terminates its scd-broadcast invocation and scd-delivers \(m \in mset\)

Termination-2: \(p_i\) scd-delivers \(m\)

\(\Rightarrow\) every non-faulty \(p_j\) scd-delivers \(m \in mset\)
Definition – Set-Constraint Delivery Broadcast

Interface

operation: scd-broadcast \((m)\) \hspace{1cm} \text{event: scd-deliver} \((mset)\)

Properties

Validity: \(p_i\) scd-delivers \(m \in mset\) \(\Rightarrow\) some \(p_j\) scd-broadcast \(m\)

Integrity: \(m\) is scd-delivered at most once by \(p_i\)

MS-Ordering: \(p_i\) scd-delivers \(m \in mset_i\) and later \(m' \in mset'_i\)

\[\downarrow\]

impossible that \(p_j\) scd-delivers \(m' \in mset'_j\) and later \(m \in mset_j\)

Termination-1: If a non-faulty \(p_i\) scd-broadcasts \(m\), it terminates its scd-broadcast invocation and scd-delivers \(m \in mset\)

Termination-2: \(p_i\) scd-delivers \(m\)

\(\Rightarrow\) every non-faulty \(p_j\) scd-delivers \(m \in mset\)
Definition – Set-Constraint Delivery Broadcast

Interface

operation: scd-broadcast \((m)\)  
event: scd-deliver \((mset)\)

Properties

Validity: \(p_i\) scd-delivers \(m \in mset\) ⇒ some \(p_j\) scd-broadcast \(m\)

Integrity: \(m\) is scd-delivered at most once by \(p_i\)

MS-Ordering: \(p_i\) scd-delivers \(m \in mset_i\) and later \(m' \in mset'_i\)

\[\downarrow\]

impossible that
\(p_j\) scd-delivers \(m' \in mset'_j\) and later \(m \in mset_j\)

Termination-1: If a non-faulty \(p_i\) scd-broadcasts \(m\), it terminates its scd-broadcast invocation and scd-delivers \(m \in mset\)

Termination-2: \(p_i\) scd-delivers \(m\)

⇒ every non-faulty \(p_j\) scd-delivers \(m \in mset\)
Definition – MS-Ordering examples

Messages SCD-broadcast by processes:

\[ m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8 \]

Correct SCD-deliveries

at \( p_1 \): \{ \{m_1, m_2\}, \{m_3, m_4, m_5\}, \{m_6\}, \{m_7, m_8\} \}

at \( p_2 \): \{ \{m_1\}, \{m_2, m_3\}, \{m_4, m_5, m_6\}, \{m_7\}, \{m_8\} \}

at \( p_3 \): \{ \{m_1, m_2, m_3\}, \{m_4, m_5, m_6\}, \{m_7\}, \{m_8\} \}

Incorrect SCD-deliveries

at \( p_1 \): \{ \{m_1, m_2\}, \{m_3, m_4, m_5\}, \{m_6\}, \{m_7, m_8\} \}

at \( p_2 \): \{ \{m_1, m_3\}, \{m_2\}, \{m_6, m_4, m_5\}, \{m_7\}, \{m_8\} \}
Definition – MS-Ordering examples

Messages SCD-broadcast by processes:

\[ m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8 \]

Correct SCD-deliveries

at \( p_1 \):  \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \}

at \( p_2 \):  \{ m_1 \}, \{ m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \}

at \( p_3 \):  \{ m_1, m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \}

Incorrect SCD-deliveries

at \( p_1 \):  \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \}

at \( p_2 \):  \{ m_1, m_3 \}, \{ m_2 \}, \{ m_6, m_4, m_5 \}, \{ m_7 \}, \{ m_8 \}
Definition – MS-Ordering examples

Messages SCD-broadcast by processes:

\[ m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8 \]

Correct SCD-deliveries

at \( p_1 \): \( \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \} \)
at \( p_2 \): \( \{ m_1 \}, \{ m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \} \)
at \( p_3 \): \( \{ m_1, m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \} \)

Incorrect SCD-deliveries

at \( p_1 \): \( \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \} \)
at \( p_2 \): \( \{ m_1, m_3 \}, \{ m_2 \}, \{ m_6, m_4, m_5 \}, \{ m_7 \}, \{ m_8 \} \)
Definition – MS-Ordering examples

Messages SCD-broadcast by processes:

\[ m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8 \]

Correct SCD-deliveries

at \( p_1 \): \( \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \} \)

at \( p_2 \): \( \{ m_1 \}, \{ m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \} \)

at \( p_3 \): \( \{ m_1, m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \} \)

Incorrect SCD-deliveries

at \( p_1 \): \( \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \} \)

at \( p_2 \): \( \{ m_1, m_3 \}, \{ m_2 \}, \{ m_6, m_4, m_5 \}, \{ m_7 \}, \{ m_8 \} \)
Definition – MS-Ordering examples

Messages SCD-broadcast by processes:

\[ m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8 \]

Correct SCD-deliveries

at \( p_1 \): \( \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \} \)

at \( p_2 \): \( \{ m_1 \}, \{ m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \} \)

at \( p_3 \): \( \{ m_1, m_2, m_3 \}, \{ m_4, m_5, m_6 \}, \{ m_7 \}, \{ m_8 \} \)

Incorrect SCD-deliveries

at \( p_1 \): \( \{ m_1, m_2 \}, \{ m_3, m_4, m_5 \}, \{ m_6 \}, \{ m_7, m_8 \} \)

at \( p_2 \): \( \{ m_1, m_3 \}, \{ m_2 \}, \{ m_6, m_4, m_5 \}, \{ m_7 \}, \{ m_8 \} \)
Definition – Propositions

Graph interpretation

- Local SCD-delivery order: \( m \rightarrow_i m' \)
  - \( p_i \) delivers \( m \) in a message set \( mset \)
  - later \( p_i \) delivers \( m' \) in another message set \( mset' \)
- Global SCD-delivery order: \( \rightarrow = \bigcup_{i=1}^{n} \rightarrow_i \)
  - \( \rightarrow \) is a partial order
- Let \( \leq \) be some total order extending \( \rightarrow \)
  - processes scd-deliver sections of \( \leq \)

A containment property

- let \( ms_i^x \) the \( x \)-th message set scd-delivered by \( p_i \)
- let \( MS_i^x = ms_i^1 \cup \cdots \cup ms_i^x \)
- \( \forall i, j, x, y, (MS_i^x \subseteq MS_j^y) \lor (MS_j^y \subseteq MS_i^x) \)
Graph interpretation

- **Local SCD-delivery order:** $m \xrightarrow{i} m'$
  - $p_i$ delivers $m$ in a message set $mset$
  - later $p_i$ delivers $m'$ in an other message set $mset'$
- **Global SCD-delivery order:** $\xrightarrow{} = \bigcup_{i=1}^{n} \xrightarrow{i}$
  - $\xrightarrow{}$ is a partial order
- Let $\leq$ be some total order extending $\xrightarrow{}$
  - processes scd-deliver sections of $\leq$

A containment property

- let $ms_i^x$ the $x$-th message set scd-delivered by $p_i$
- let $MS_i^x = ms_i^1 \cup \cdots \cup ms_i^x$
- $\forall i, j, x, y, (MS_i^x \subseteq MS_j^y) \lor (MS_j^y \subseteq MS_i^x)$
Operation `read()`:
1. return `state`;

Operation `insert(v)`:
2. `scd-broadcast I(v); wait local delivery;`

Event `scd-deliver`:
3. `state ← state ∪ {v_1, ..., v_k}`
1 operation read():
  scd-broadcast Sync; wait local delivery;
  return state;

4 operation insert(v):
  scd-broadcast I(v); wait local delivery;

6 event scd-deliver (\{I(v_1), \ldots, I(v_k), Sync_1, \ldots, Sync_l\}): 
  state ← state ∪ \{v_1, \ldots, v_k\}
The MWMR snapshot object

abstract state: an array of registers

write\( (x, v) \): write \( v \) in register \( x \)

snapshot(): returns the whole array

1 operation snapshot():
   return Regs;

3 operation write\( (x, v) \):
   let \( \langle sn, j \rangle \leftarrow tsa[x] \);
   scd-broadcast Write\( (x, v, \langle sn + 1, i \rangle) \); wait local delivery;

6 event scd-deliver \((mset)\):
   foreach Write\( (x, v, ts) \in mset \text{ s.t. } ts > tsa[x] \) do
      Regs\[x\] \leftarrow v; tsa[x] \leftarrow ts;
The MWMR snapshot object

abstract state: an array of registers

write\((x, v)\): write \(v\) in register \(x\)

snapshot(): returns the whole array

1 operation snapshot():
2 return RRegs;

3 operation write\((x, v)\):
4 let \(\langle sn, j \rangle \leftarrow tsa[x]\);
5 scd-broadcast \(Write(x, v, \langle sn + 1, i \rangle)\); wait local delivery;

6 event scd-deliver \((mset)\):
7 foreach \(Write(x, v, ts) \in mset s.t. ts > tsa[x]\) do
8 Regs\([x]\) ← \(v\); \(tsa[x]\) ← \(ts\);
1 operation snapshot():
2   scd-broadcast Sync; wait local delivery;
3   return Regs;

4 operation write(x, v):
5   scd-broadcast Sync; wait local delivery;
6   let ⟨sn, j⟩ ← tsa[x];
7   scd-broadcast Write(x, v, ⟨sn + 1, i⟩); wait local delivery;

8 event scd-deliver (mset):
9   foreach Write(x, v, ts) ∈ mset s.t. ts > tsa[x] do
10      Regs[x] ← v; tsa[x] ← ts;
Power – Atomic snapshot object

1 operation snapshot():
2    scd-broadcast Sync; \textbf{wait} local delivery;
3    \textbf{return} Regs;

4 operation write(x, v):
5    scd-broadcast Sync; \textbf{wait} local delivery;
6    \textbf{let} \langle sn, j \rangle \leftarrow tsa[x];
7    scd-broadcast Write(x, v, \langle sn + 1, i \rangle); \textbf{wait} local delivery;

8 event scd-deliver (mset):
9    \textbf{foreach} Write(x, v, ts) \in mset \textbf{s.t.} ts > tsa[x] \textbf{do}
10       \textbf{Regs}[x] \leftarrow v; tsa[x] \leftarrow ts;
Remarks for software engineers

1. **operation** `snapshot()`:  
   - `scd-broadcast Sync; wait local delivery;`  
   - `return Regs;`

2. **operation** `write(x, v)`:  
   - `scd-broadcast Sync; wait local delivery;`  
   - `let \( \langle sn, j \rangle \leftarrow tsa[x] \);`  
   - `scd-broadcast Write(x, v, \langle sn + 1, i \rangle); wait local delivery;`

3. **event** `scd-deliver (mset)`:  
   - `foreach Write(x, v, ts) \in mset s.t. ts > tsa[x] do`  
   - `Regs[x] \leftarrow v; tsa[x] \leftarrow ts;`

Observations

- No quorum at this abstraction level!
- Each element plays its role:
  - structure, sequential consistency, overwriting, real-time
- Works for all objects with commutative/overwriting operations
Implementation – Shared memory

1. **Operation** SCD-broadcast\((m)\): \(\text{Reg}[i] \leftarrow \text{Reg}[i] \cdot m\);

2. **Regularly do:**
   
3. \(\text{regs} \leftarrow \text{Reg.snapshot}()\);

4. \(S \leftarrow \bigcup_{j=1}^{n} \text{regs}[j] \setminus \text{delivered}\);

5. **If** \(S \neq \emptyset\) **then** SCD-deliver\((S)\);

Consequences

- From sequential consistency to linearizability
- Equivalence SCD-broadcast/atomic register
- Consensus Number = 1
- Message-passing implementation: \(t < \frac{n}{2}\)
Implementation – Shared memory

1 operation SCD-broadcast($m$): $\text{Reg}[i] \leftarrow \text{Reg}[i] \cdot m$;
2 Regularly do:
3 \[
\begin{align*}
\text{regs} &\leftarrow \text{Reg.snapshot}(); \\
S &\leftarrow \bigcup_{j=1}^{n} \text{regs}[j] \setminus \text{delivered}; \\
\text{if } S \neq \emptyset \text{ then } &\text{SCD-deliver}(S);
\end{align*}
\]

Consequences

- From sequential consistency to linearizability
- Equivalence SCD-broadcast/atomic register

<table>
<thead>
<tr>
<th>SCD broadcast</th>
<th>Previous slide</th>
<th>Snapshot object</th>
<th>(projection)</th>
<th>Atomic register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above</td>
<td>Afek et Al.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Consensus Number = 1
- Message-passing implementation: $t < \frac{n}{2}$
Implementation –  Message-passing

Process $p_{sd}$ SCD-broadcasts $m$

- each process $p_f$ fifo-broadcasts $forward(m, sd, sn_{sd}, f, sn_f)$

Dependencies

- $p_f$ views $m$ before $m'$ if
  - $p_f$ sends $forward(m, \cdot, \cdot, f, sn_f)$ and $forward(m', \cdot, \cdot, f, sn_f')$
  - $sn_f < sn_f'$

- $p_i$ knows that $p_f$ has viewed $m$ before $m'$ if $p_i$ received either
  - $forward(m, \cdot, \cdot, f, \cdot)$ but no $forward(m', \cdot, \cdot, f, \cdot)$
  - $forward(m, \cdot, \cdot, f, sn_f)$ and $forward(m', \cdot, \cdot, f, sn_f')$, $sn_f < sn_f'$

- $m$ depends on $m'$ (according to $p_i$) unless $p_i$ knows that:
  - a majority of processes have viewed $m$ before $m'$

Delivery condition

- $p_i$ can scd-deliver $mset$ if for all $m \in mset$
  - $p_i$ received $forward$ messages from a majority
  - $mset$ contains all non-delivered dependencies of $m$
Implementation – Message-passing

Process \( p_{sd} \) SCD-broadcasts \( m \)

- each process \( p_f \) fifo-broadcasts \( \text{forward}(m, sd, sn_{sd}, f, sn_f) \)

Dependencies

- \( p_f \) views \( m \) before \( m' \) if
  - \( p_f \) sends \( \text{forward}(m, \cdot, \cdot, f, sn_f) \) and \( \text{forward}(m', \cdot, \cdot, f, sn'_f) \)
  - \( sn_f < sn'_f \)

- \( p_i \) knows that \( p_f \) has viewed \( m \) before \( m' \) if \( p_i \) received either
  - \( \text{forward}(m, \cdot, \cdot, f, \cdot) \) but no \( \text{forward}(m', \cdot, \cdot, f, \cdot) \)
  - \( \text{forward}(m, \cdot, \cdot, f, sn_f) \) and \( \text{forward}(m', \cdot, \cdot, f, sn'_f) \), \( sn_f < sn'_f \)

- \( m \) depends on \( m' \) (according to \( p_i \)) unless \( p_i \) knows that:
  - a majority of processes have viewed \( m \) before \( m' \)

Delivery condition

- \( p_i \) can scd-deliver \( mset \) if for all \( m \in mset \)
  - \( p_i \) received \( \text{forward} \) messages from a majority
  - \( mset \) contains all non-delivered dependencies of \( m \)
Implementation – Message-passing

Process $p_{sd}$ SCD-broadcasts $m$

- each process $p_f$ fifo-broadcasts $\text{forward}(m, sd, sn_{sd}, f, sn_f)$

Dependencies

- $p_f$ views $m$ before $m'$ if
  - $p_f$ sends $\text{forward}(m, \cdot, \cdot, f, sn_f)$ and $\text{forward}(m', \cdot, \cdot, f, sn'_f)$
  - $sn_f < sn'_f$

- $p_i$ knows that $p_f$ has viewed $m$ before $m'$ if $p_i$ received either
  - $\text{forward}(m, \cdot, \cdot, f, \cdot)$ but no $\text{forward}(m', \cdot, \cdot, f, \cdot)$
  - $\text{forward}(m, \cdot, \cdot, f, sn_f)$ and $\text{forward}(m', \cdot, \cdot, f, sn'_f)$, $sn_f < sn'_f$

- $m$ depends on $m'$ (according to $p_i$) unless $p_i$ knows that:
  - a majority of processes have viewed $m$ before $m'$

Delivery condition

- $p_i$ can scd-deliver $mset$ if for all $m \in mset$
  - $p_i$ received $\text{forward}$ messages from a majority
  - $mset$ contains all non-delivered dependencies of $m$
Implementation – Message-passing

Process $p_{sd}$ SCD-broadcasts $m$

- each process $p_f$ fifo-broadcasts $\text{forward}(m, sd, sn_{sd}, f, sn_f)$

Dependencies

- $p_f$ views $m$ before $m'$ if
  - $p_f$ sends $\text{forward}(m, \cdot, \cdot, f, sn_f)$ and $\text{forward}(m', \cdot, \cdot, f, sn'_f)$
  - $sn_f < sn'_f$

- $p_i$ knows that $p_f$ has viewed $m$ before $m'$ if $p_i$ received either
  - $\text{forward}(m, \cdot, \cdot, f, \cdot)$ but no $\text{forward}(m', \cdot, \cdot, f, \cdot)$
  - $\text{forward}(m, \cdot, \cdot, f, sn_f)$ and $\text{forward}(m', \cdot, \cdot, f, sn'_f)$, $sn_f < sn'_f$

- $m$ depends on $m'$ (according to $p_i$) unless $p_i$ knows that:
  - a majority of processes have viewed $m$ before $m'$

Delivery condition

- $p_i$ can scd-deliver $mset$ if for all $m \in mset$
  - $p_i$ received $\text{forward}$ messages from a majority
  - $mset$ contains all non-delivered dependencies of $m$
Implementation – Complexity

### SCD-broadcast

- **# msgs:** \( n^2 \)
- **latency:** \( 2\Delta \) (\( \Delta \): network delay)

#### Snapshot object

<table>
<thead>
<tr>
<th></th>
<th>Read / Snapshot latency</th>
<th>Write latency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># msgs</strong></td>
<td><strong># msgs</strong></td>
<td><strong># msgs</strong></td>
</tr>
<tr>
<td>ABD</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>( 4\Delta )</td>
<td>( 2\Delta )</td>
</tr>
<tr>
<td>ABD + AR</td>
<td>( O(n^2 \log n) )</td>
<td>( O(n^2 \log n) )</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>( O(n \log n \Delta) )</td>
<td>( O(n \log n \Delta) )</td>
</tr>
<tr>
<td>DGFRR</td>
<td>( O(n^3) )</td>
<td>( O(n \Delta) )</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>( O(n \log n \Delta) )</td>
<td>( O(n \log n \Delta) )</td>
</tr>
<tr>
<td>SCD-Atomic</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>( 2\Delta )</td>
<td>( 4\Delta )</td>
</tr>
<tr>
<td>PPMJ</td>
<td>0</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>( 0 - 4\Delta )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>SCD-Sequential</td>
<td>0</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td><strong>latency</strong></td>
<td>0</td>
<td>( 2\Delta )</td>
</tr>
</tbody>
</table>

References:


## Implementation – Complexity

### SCD-broadcast

- **# msgs:** $n^2$
- **latency:** $2\Delta$ ($\Delta$: network delay)

### Snapshot object

<table>
<thead>
<tr>
<th></th>
<th>Read / Snapshot</th>
<th></th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># msgs</td>
<td>latency</td>
<td># msgs</td>
</tr>
<tr>
<td>ABD</td>
<td>$\mathcal{O}(n)$</td>
<td>$4\Delta$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>ABD + AR</td>
<td>$\mathcal{O}(n^2 \log n)$</td>
<td>$\mathcal{O}(n \log n\Delta)$</td>
<td>$\mathcal{O}(n^2 \log n)$</td>
</tr>
<tr>
<td>DGFRR</td>
<td>$\mathcal{O}(n^3)$</td>
<td>$\mathcal{O}(n\Delta)$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td>SCD-Atomic</td>
<td>$\mathcal{O}(n^2)$</td>
<td>$2\Delta$</td>
<td>$\mathcal{O}(n^2)$</td>
</tr>
<tr>
<td>PPMJ</td>
<td>0</td>
<td>$0 - 4\Delta$</td>
<td>$\mathcal{O}(n^2)$</td>
</tr>
<tr>
<td>SCD-Sequential</td>
<td>0</td>
<td>0</td>
<td>$\mathcal{O}(n^2)$</td>
</tr>
</tbody>
</table>

Can we limit the size of the message sets?

**k-SCD broadcast**

**Definition:** All message sets contain at most $k$ messages

**Observation:** $1$-SCD broadcast $\simeq$ Atomic broadcast

**k-set agreement**

Extension of consensus

**Termination:** Each non-faulty process eventually decides a value

**Validity:** All decided values have been proposed

**k-Agreement:** At most $k$ different values are decided

**Theorem**

$k$-SCD broadcast $\simeq$ SCD broadcast + $k$-set agreement

Perspectives – $k$-SCD broadcast

Can we limit the size of the message sets?

$k$-SCD broadcast

Definition: All message sets contain at most $k$ messages

Observation: $1$-SCD broadcast $\simeq$ Atomic broadcast

$k$-set agreement

Extension of consensus

Termination: Each non-faulty process eventually decides a value

Validity: All decided values have been proposed

$k$-Agreement: At most $k$ different values are decided

Theorem

$k$-SCD broadcast $\simeq$ SCD broadcast + $k$-set agreement

Some operations do not commute

**Monotonic Generic Broadcast**

- Based on a *conflict* relation (like generic broadcast)
- Conflicting operations ordered inside message sets

**Specific cases**

- No conflicts: SCD-broadcast
- Only conflicts: Atomic broadcast

Consensus: only when necessary