Byzantine-Tolerant Set-Constrained Delivery Broadcast

Alex Auvolat, Michel Raynal, François Taïani

December 17th, 2019 @ OPODIS, Neuchâtel

Motivation

- Investigate broadcast primitives as high-level **abstractions** for implementing distributed objects
- Byzantine-tolerant algorithms have critical applications (cryptocurrencies, smart contracts, ...)
- Byzantine consensus is a complex and costly primitive
- Set Constrained Delivery (SCD) broadcast is **less costly than consensus**, yet it allows easy construction of **linearizable objects**

Set-Constraint Delivery Broadcast

- We no longer deliver single messages, but **sets of messages**.

  Example: $\{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots$

Set-Constraint Delivery Broadcast

- We no longer deliver single messages, but **sets of messages**.

  Example: $\{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \dots$

- Order property: if a process scd-delivers a set ms_1 containing m_1 and later a set ms_2 containing m_2, then no process scd-delivers a set ms'_1 containing m_2 and later a set ms'_2 containing m_1.

Set-Constraint Delivery Broadcast

- We no longer deliver single messages, but **sets of messages**.

  Example: $\{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots$

- Order property: if a process scd-delivers a set ms_1 containing m_1 and later a set ms_2 containing m_2, then no process scd-delivers a set ms_1' containing m_2 and later a set ms_2' containing m_1.

  Correct: p_i : $\{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots$
  p_j : $\{m_1, m_2, m_3\}, \{m_4\}, \{m_5, m_6\}, \ldots$

Set-Constraint Delivery Broadcast

- We no longer deliver single messages, but **sets of messages**.

  Example: $\{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots$

- Order property: if a process scd-delivers a set ms_1 containing m_1 and later a set ms_2 containing m_2, then no process scd-delivers a set ms_1' containing m_2 and later a set ms_2' containing m_1.

Correct:

$$\begin{aligned} p_i &: \quad \{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots \\ p_j &: \quad \{m_1, m_2, m_3\}, \{m_4\}, \{m_5, m_6\}, \ldots \end{aligned}$$

Incorrect:

$$\begin{aligned} p_i &: \quad \{m_1, {\color{red}m_2}\}, \{{\color{red}m_3}\}, \{m_4, m_5, m_6\}, \ldots \\ p_j &: \quad \{m_1, {\color{red}m_3}\}, \{{\color{red}m_2}, m_4\}, \{m_5, m_6\}, \ldots \end{aligned}$$

SCD-broadcast in crash-prone systems

D. Imbs, A. Mostéfaoui, M. Perrin and M. Raynal:
Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits, ICDCN 2018

- Definition of SCD broadcast
- Algorithm in crash-prone systems with $t < n/2$
  t: number of crashed processes, n: total number of processes
- Programming power: snapshot object, counter object, lattice agreement
- Computability limits: equivalent to read/write registers
  (consensus number 1)

System model

Process model:

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

Communication model:

- complete point-to-point network
- asynchronous messages with finite (unbounded) delays
- reliable point-to-point links:
  no loss, creation, duplication or alteration of messages

System model

Failure model: up to t processes exhibit **Byzantine behaviour**

- A Byzantine process may deviate arbitrarily from the spec.
  It may omit messages or send arbitrary messages.

System model

Failure model: up to t processes exhibit **Byzantine behaviour**

- A Byzantine process may deviate arbitrarily from the spec.
  It may omit messages or send arbitrary messages.
- A Byzantine process may send messages with arbitrary delays.

System model

Failure model: up to t processes exhibit **Byzantine behaviour**

- A Byzantine process may deviate arbitrarily from the spec.
  It may omit messages or send arbitrary messages.
- A Byzantine process may send messages with arbitrary delays.
- A Byzantine process may also behave like a correct process.

System model

Failure model: up to t processes exhibit **Byzantine behaviour**

- A Byzantine process may deviate arbitrarily from the spec.
  It may omit messages or send arbitrary messages.
- A Byzantine process may send messages with arbitrary delays.
- A Byzantine process may also behave like a correct process.
- Byzantine processes may coordinate their malicious actions.

System model

Failure model: up to t processes exhibit **Byzantine behaviour**

- A Byzantine process <u>may</u> deviate arbitrarily from the spec.
  It may omit messages or send arbitrary messages.
- A Byzantine process <u>may</u> send messages with arbitrary delays.
- A Byzantine process <u>may also</u> behave like a correct process.
- Byzantine processes <u>may</u> coordinate their malicious actions.
- A Byzantine process <u>may not</u> pretend to be another process.
  The system model guarantees the identity of the sender.

Main difficulties

- We cannot control the behaviour of Byzantine processes
- Correct processes collectively ensure properties on message deliveries no matter what Byzantine processes do
- Sender can never be trusted: validation logic on the receiver end at each correct process

Definition of Byzantine SCD broadcast

Two operations:

- bscd_broadcast(m): broadcast a message m
- bscd_deliver(): returns a non-empty set of messages

Definition of Byzantine SCD broadcast

Five properties:

- *Validity.* If a correct process bscd-delivers a message m from a correct process p_i, then p_i bscd-broadcast m.
- *Integrity.* A message is bscd-delivered at most once by each correct process.

Definition of Byzantine SCD broadcast

Five properties:

- *Validity.* If a correct process bscd-delivers a message m from a correct process p_i, then p_i bscd-broadcast m.
- *Integrity.* A message is bscd-delivered at most once by each correct process.
- *Ordering.* Let p_i be a correct process that first bscd-delivers a set of messages ms_i and later bscd-delivers a set of messages ms'_i. For any messages $m \in ms_i, m' \in ms'_i$, no correct process bscd-delivers first a set containing m' and later a set containing m.

Definition of Byzantine SCD broadcast

Five properties:

- *Validity.* If a correct process bscd-delivers a message m from a correct process p_i, then p_i bscd-broadcast m.
- *Integrity.* A message is bscd-delivered at most once by each correct process.
- *Ordering.* Let p_i be a correct process that first bscd-delivers a set of messages ms_i and later bscd-delivers a set of messages ms_i'. For any messages $m \in ms_i, m' \in ms_i'$, no correct process bscd-delivers first a set containing m' and later a set containing m.
- *Termination-1.* If a correct process bscd-broadcasts a message m, it bscd-delivers a message set containing m.
- *Termination-2.* If a correct process bscd-delivers a message set containing m, every correct processes bscd-delivers a message set containing m.

Definition of Byzantine SCD broadcast

- *Validity.* If a correct process bscd-delivers a message m from a correct process p_i, then p_i bscd-broadcast m.
- *Integrity.* A message is bscd-delivered at most once by each correct process.

- *Termination-1.* If a correct process bscd-broadcasts a message m, it bscd-delivers a message set containing m.
- *Termination-2.* If a correct process bscd-delivers a message set containing m, every correct processes bscd-delivers a message set containing m.

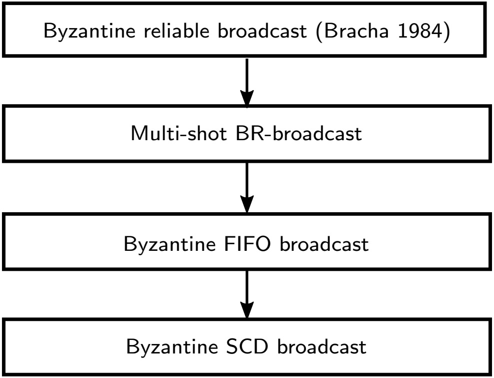
Definition of Byzantine SCD broadcast

- *Validity.* If a correct process bscd-delivers a message m from a correct process p_i, then p_i bscd-broadcast m.
- *Integrity.* A message is bscd-delivered at most once by each correct process.

this is Byzantine Reliable Broadcast

- *Termination-1.* If a correct process bscd-broadcasts a message m, it bscd-delivers a message set containing m.
- *Termination-2.* If a correct process bscd-delivers a message set containing m, every correct processes bscd-delivers a message set containing m.

A Modular Approach



Multi-shot BRB

One-shot Byzantine Reliable Broadcast:

G. Bracha: *Asynchronous Byzantine agreement protocols* (1987)

Multi-shot BRB

One-shot Byzantine Reliable Broadcast:

G. Bracha: *Asynchronous Byzantine agreement protocols* (1987)

Multi-shot BRB: processes may call Byzantine Reliable Broadcast multiple times, each time with a different sequence number:

$$\text{br\_broadcast}(sn_i, m)$$

BR-broadcast of message m by process p_i with sequence number sn_i.

<u>These instances operate independently.</u>

Sequence numbers are just tags on messages that do not induce any ordering.

A Simple Sub-Protocol: Byzantine FIFO Broadcast

FIFO delivery is hard to define in the case of Byzantine systems:

- If a correct process bfifo-broadcasts m before m', then all correct processes bfifo-deliver m before m'.

A Simple Sub-Protocol: Byzantine FIFO Broadcast

FIFO delivery is hard to define in the case of Byzantine systems:

- If a correct process bfifo-broadcasts m before m', then all correct processes bfifo-deliver m before m'.

What if the sender is Byzantine?

A Simple Sub-Protocol: Byzantine FIFO Broadcast

FIFO delivery is hard to define in the case of Byzantine systems:

- If a correct process bfifo-broadcasts m before m', then all correct processes bfifo-deliver m before m'.

What if the sender is Byzantine?

- If a correct process p_i bfifo-delivers m before m' both from the same *possibly Byzantine* process p_k, then no correct process bfifo-delivers m' before m.

An order is decided by the correct processes even if the sender is Byzantine.

Byzantine-tolerant FIFO broadcast algorithm

However, the algorithm is extremely simple:

init $sn_i \leftarrow 0$; $fifo\_del_i \leftarrow [0, \ldots, 0]$.

operation bfifo_broadcast(m) **at** p_i **is**
(1) $sn_i \leftarrow sn_i + 1$;
(2) br_broadcast(sn_i, m).

when $\langle j, sn, m \rangle$ **is** br_delivered **at** p_i **do**
(3) **wait** $\big(sn = fifo\_del_i[j] + 1\big)$;
(4) bfifo_deliver $\langle j, sn, m \rangle$;
(5) $fifo\_del_i[j] \leftarrow fifo\_del_i[j] + 1$.

A Reminder: Byzantine SCD broadcast

Order property: if a correct process bscd-delivers a set ms_1 containing m_1 and later a set ms_2 containing m_2, then no correct process bscd-delivers a set ms'_1 containing m_2 and later a set ms'_2 containing m_1.

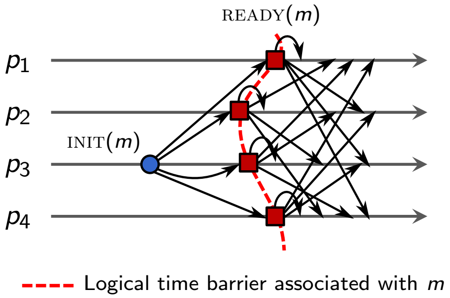
Correct:
$$p_i: \quad \{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots$$
$$p_j: \quad \{m_1, m_2, m_3\}, \{m_4\}, \{m_5, m_6\}, \ldots$$

Incorrect:
$$p_i: \quad \{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots$$
$$p_j: \quad \{m_1, m_3\}, \{m_2, m_4\}, \{m_5, m_6\}, \ldots$$

BSCD-Broadcast Algorithm

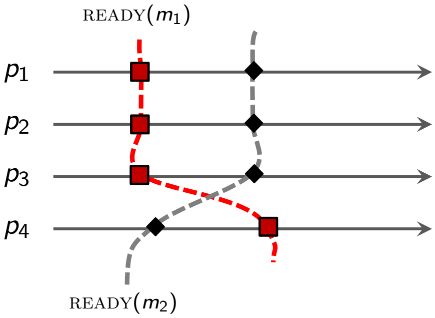
- Processes announce the time (local sequence number) at which they receive messages using Byzantine FIFO-broadcast
- Thanks to Byzantine FIFO-broadcast properties, **all correct processes receive the same sequence of acknowledgements from any other process.**
- Main idea: a correct process may not deliver m_1 before m_2 if it does not know that a majority of processes have seen m_1 before m_2.

Message echo mechanism



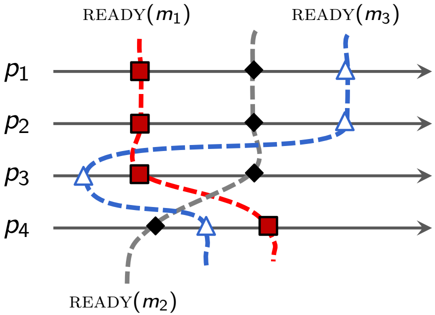
Each READY message has a FIFO sequence number which cannot be faked: all correct processes see the same logical time barrier (as defined by sequence numbers)

Message echo mechanism: example 1

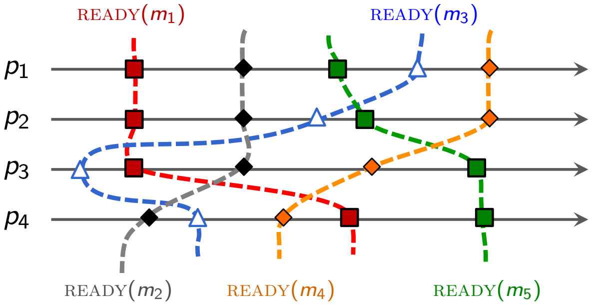


A correct process that has received all the READY messages will know that it is safe to bscd-deliver m_1 before m_2.

Message echo mechanism: example 2



In this case, the three messages must always be bscd-delivered simultaneously.



every message of $\{\mathbf{m_1}, \mathbf{m_2}, \mathbf{m_3}\}$ comes before every message of $\{\mathbf{m_4}, \mathbf{m_5}\}$

A correct process may deliver $\{\mathbf{m_1}, \mathbf{m_2}, \mathbf{m_3}\}$ and then $\{\mathbf{m_4}, \mathbf{m_5}\}$.

Byzantine SCD Broadcast Algorithm

Difficulties in the Byzantine setting:

- Ensure that Byzantine processes cannot prevent correct processes from seeing the same order (BFIFO broadcast)
- **Ensure that Byzantine processes cannot create an infinite set of messages that block one another**

In our paper:

- The complete algorithm for $t < n/4$
- Full proof of the algorithm

Cost of Byzantine SCD Broadcast

Byzantine SCD Broadcast of a single message:
$O(n)$ BRB invocations in two sequential steps

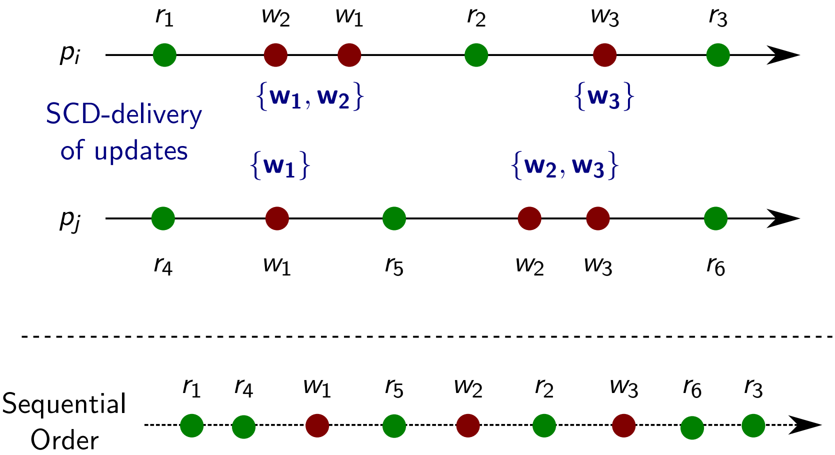
- G. Bracha: *Asynchronous Byzantine agreement protocols* (1987)
  $2n$ messages, 3 sequential communication steps

  $\rightarrow$ $\underline{2n^2 \text{ messages, 6 sequential communication steps}}$

- **For $\mathbf{t < n/5}$:** D. Imbs, M. Raynal: *Trading t-resilience for efficiency in asynchronous Byzantine reliable broadcast* (2016)
  n messages, 2 sequential communication steps

  $\rightarrow$ $\underline{n^2 \text{ messages, 4 sequential communication steps}}$

Sequential Consistency with SCD



Computing Power: the Snapshot Object

init $reg_i \leftarrow [\perp, \ldots, \perp]$; $wsn_i \leftarrow [0, \ldots, 0]$.

operation snapshot() **is**
(1) $done_i \leftarrow \texttt{false}$; bscd_broadcast $\textsc{sync}()$; **wait**$(done_i)$;
(2) **return**$(reg_i[1..n])$.

operation write(v) **is**
(3) $done_i \leftarrow \texttt{false}$; bscd_broadcast $\textsc{write}(v)$; **wait**$(done_i)$.

when $ms = \{ \langle j_1,\ sn_1,\ \textsc{write}(v_1)\rangle, \ldots, \langle j_x,\ sn_x,\ \textsc{write}(v_x)\rangle, \langle j_{x+1},\ sn_{x+1},\ \textsc{sync}()\rangle, \ldots, \langle j_y,\ sn_y,\ \textsc{sync}()\rangle \}$
is bscd-delivered do
(4) **for each message** $\langle j, snj, \textsc{write}(v)\rangle \in ms$ **do**
(5) **if** $(wsn_i[j] < snj)$ **then** $reg_i[j] \leftarrow v$; $wsn_i[j] \leftarrow snj$ **end if**
(6) **end for**;
(7) **if** $\exists \ell\ :\ j_\ell = i$ **then** $done_i \leftarrow \texttt{true}$ **end if**.

A linearizable Byzantine-tolerant SW/MR snapshot object.

Conclusion

- Powerful abstraction for sequentially consistent or linearizable read/write objects such as snapshots

Conclusion

- Powerful abstraction for sequentially consistent or linearizable read/write objects such as snapshots
- SCD broadcast algorithm for $t < n/4$, but can we do better? Or is this a tight bound for the problem?

Conclusion

- Powerful abstraction for sequentially consistent or linearizable read/write objects such as snapshots
- SCD broadcast algorithm for $t < n/4$, but can we do better? Or is this a tight bound for the problem?
- Other potential applications: lattice agreement, ...