High Level Primitives in Byzantine Systems

Alex Auvolat

WIDE Team Seminar, May 16-17, 2019
Byzantine systems: $BAMP_{n,t}[\ldots]$
An example: Trustworthy Asset Transfer (AT2)

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 Λ</td>
<td>100 Λ</td>
<td>10 000 Λ</td>
</tr>
</tbody>
</table>

Guarantees:
- No money is ever created or destroyed
- An account always has a balance $\geq 0$
An example: Trustworthy Asset Transfer (AT2)

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 C</td>
<td>100 C</td>
<td>10 000 C</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 C</td>
<td>-50 C</td>
</tr>
</tbody>
</table>

Guarantees:
- No money is ever created or destroyed
- An account always has a balance $\geq 0$
### An example: Trustworthy Asset Transfer (AT2)

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 Ʌ</td>
<td>100 Ʌ</td>
<td>10 000 Ʌ</td>
</tr>
<tr>
<td>C1</td>
<td>+50 Ʌ</td>
<td>-50 Ʌ</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100 Ʌ</td>
<td>150 Ʌ</td>
<td>9 950 Ʌ</td>
</tr>
</tbody>
</table>

**Guarantees:**
- No money is ever created or destroyed
- An account always has a balance $\geq 0$
An example: Trustworthy Asset Transfer (AT2)

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 Ç</td>
<td>100 Ç</td>
<td>10 000 Ç</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td>100 Ç</td>
<td>+50 Ç</td>
<td>-50 Ç</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>-150 Ç</td>
<td>+150 Ç</td>
<td></td>
</tr>
</tbody>
</table>

Guarantees:

- No money is ever created or destroyed
- An account always has a balance $\geq 0$
### An example: Trustworthy Asset Transfer (AT2)

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ฿</td>
<td>100 ฿</td>
<td>10 000 ฿</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ฿</td>
<td>-50 ฿</td>
</tr>
<tr>
<td></td>
<td>100 ฿</td>
<td>150 ฿</td>
<td>9 950 ฿</td>
</tr>
<tr>
<td>A1</td>
<td>-150 ฿</td>
<td></td>
<td>+150 ฿</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>impossible!</td>
</tr>
</tbody>
</table>

### Guarantees:
- No money is ever created or destroyed
- An account always has a balance $\geq 0$
An example: Trustworthy Asset Transfer (AT2)

AT2: introduced in Guerraoui et al., 2018

Question: general modular approach?
AT2: introduced in Guerraoui et al., 2018

Question: general modular approach?

→ high-level communication abstractions in Byzantine systems
→ consistency criteria in Byzantine systems
System Model

\( n \) nodes (or processes)
System Model

Up to $t$ Byzantine nodes
A Byzantine process may deviate arbitrarily from the spec. It may omit messages or send arbitrary messages.
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.
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 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.
A Byzantine process may not pretend to be another process. The system model guarantees the identity of the sender.
System Model

Authenticated point-to-point links
Authenticated point-to-point links
Asynchronous message passing
One-shot Byzantine Reliable Broadcast: a fundamental primitive.

- **BR-Validity.** If a correct process br-delivers a message $m$ from a correct process $p_i$, then $p_i$ br-broadcasts $m$.

- **BR-Integrity.** A correct process br-delivers at most one message $m$ from a process $p_i$.

- **BR-Termination-1.** If a correct process br-broadcasts a message, it br-delivers it.

- **BR-Termination-2.** If a correct process br-delivers a message $m$ from $p_i$ (possibly Byzantine) then all correct processes eventually br-deliver $m$ from $p_i$. 
- G. Bracha (1984): tolerant to $t < n/3$ Byzantine nodes.
Tolerance to Byzantine Nodes

- G. Bracha (1984): tolerant to $t < n/3$ Byzantine nodes.

- D. Imbs, M. Raynal (2016): lower message complexity. Tolerant to $t < n/5$ Byzantine nodes.
Tolerance to Byzantine Nodes

- G. Bracha (1984): tolerant to $t < n/3$ Byzantine nodes.

- D. Imbs, M. Raynal (2016): lower message complexity. Tolerant to $t < n/5$ Byzantine nodes.

All algorithms we build over BRB have the same requirements as the selected underlying BRB implementation. They may also have their own independent requirements (e.g. $t < n/4$ for SCD broadcast).
Multi-shot BRB

We need multiple instances of BRB so that each process can send several messages.
Multi-shot BRB

We need multiple instances of BRB so that each process can send several messages.

\[ \text{br\_broadcast}(\langle i, sn_i \rangle, m) \]

BR-broadcast of message \( m \) by process \( p_i \) with sequence number \( sn_i \).
Multi-shot BRB

We need multiple instances of BRB so that each process can send several messages.

\[ \text{br\_broadcast(} \langle i, sn_i \rangle, m) \]

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

These instances operate independently.

(no order guarantee between instances)
init init[1..n]: constant array where init[k] is the initial value of pk account;
   hist;[1..n] ← [∅, · · · , ∅]; del;[1..n] ← [0, · · · , 0]; sn; ← 0.

operation transfer(j, ν) is
   (1) if (balance(i) < ν)
   (2) then return (abort)
   (3) else sn; ← sn; + 1; done; ← false;
   (4) br_broadcast (⟨i, sn;⟩, TRANSFER(j, ν));
   (5) wait (done;); return (commit).

when (⟨j, sn⟩, TRANSFER(k, ν)) is br_delivered from p_j do
   (6) wait (balance(j) ≥ ν) ∧ (del;[j] + 1 = sn);
   (7) hist;[j] ← hist;[j] ∪ {⟨k, ν⟩};
   (8) del;[j] ← sn;
   (9) if (j = i) then done; ← true.

internal function balance(j) is
   (10) return (init[j] + ∑ℓ ∑⟨j, νx⟩∈hist;[ℓ] νx − ∑⟨−, νx⟩∈hist;[j] νx).
## AT2 Behaviour

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₪</td>
<td>100 ₪</td>
<td>10 000 ₪</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td>100 ₪</td>
<td>+50 ₪</td>
<td>-50 ₪</td>
</tr>
<tr>
<td></td>
<td>100 ₪</td>
<td>150 ₪</td>
<td>9 950 ₪</td>
</tr>
</tbody>
</table>
## AT2 Behaviour

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₪</td>
<td>100 ₪</td>
<td>10 000 ₪</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td></td>
<td>+50 ₪</td>
<td>-50 ₪</td>
</tr>
<tr>
<td></td>
<td>100 ₪</td>
<td>150 ₪</td>
<td>9 950 ₪</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>-150 ₪</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+150 ₪</td>
<td></td>
</tr>
</tbody>
</table>
### AT2 Behaviour

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₷</td>
<td>100 ₷</td>
<td>10 000 ₷</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td></td>
<td>+50 ₷</td>
<td>-50 ₷</td>
</tr>
<tr>
<td></td>
<td>100 ₷</td>
<td>150 ₷</td>
<td>9 950 ₷</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>-150 ₷</td>
<td>+150 ₷</td>
<td></td>
</tr>
<tr>
<td><strong>on hold...</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## AT2 Behaviour

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 C</td>
<td>100 C</td>
<td>10 000 C</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100 C</td>
<td>150 C</td>
<td>9 950 C</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>-150 C</td>
<td>+150 C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>on hold...</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C2</strong></td>
<td>+50 C</td>
<td></td>
<td>-50 C</td>
</tr>
</tbody>
</table>
### AT2 Behaviour

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ฿</td>
<td>100 ฿</td>
<td>10,000 ฿</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ฿</td>
<td>-50 ฿</td>
</tr>
<tr>
<td></td>
<td>100 ฿</td>
<td>150 ฿</td>
<td>9,950 ฿</td>
</tr>
<tr>
<td>C2</td>
<td>+50 ฿</td>
<td></td>
<td>-50 ฿</td>
</tr>
<tr>
<td>A1</td>
<td>-150 ฿</td>
<td>+150 ฿</td>
<td></td>
</tr>
</tbody>
</table>
### AT2 Behaviour

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₧</td>
<td>100 ₧</td>
<td>10 000 ₧</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ₧</td>
<td>-50 ₧</td>
</tr>
<tr>
<td>A1</td>
<td>-150 ₧</td>
<td>+150 ₧</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>+50 ₧</td>
<td></td>
<td>-50 ₧</td>
</tr>
<tr>
<td></td>
<td>0 ₧</td>
<td>300 ₧</td>
<td>9 900 ₧</td>
</tr>
</tbody>
</table>
Let’s Split It Up

Byzantine systems: $BAMP_{n,t}[\ldots]$

- Specialized AT2 Algorithm
- Causal broadcast
- Reliable broadcast (Bracha, 1984)
  - $BAMP_{n,t}[t < n/3]$ (Causal broadcast)
  - FIFO broadcast
  - $BAMP_{n,t}[t < n/4]$ (SCD broadcast)

Sequential specifications of objects with some restrictions

- Update consistency
  - Generic algorithm
- Snapshot Asset Transfer etc.
- Sequential consistency

Alex Auvolat
High Level Primitives in Byzantine Systems
Byzantine systems: $BAMP_{n,t} [...]$

- Reliable broadcast (Bracha, 1984)
- $BAMP_{n,t}[t < n/3]$
- Causal broadcast
- FIFO broadcast
- $BAMP_{n,t}[t < n/4]$
- SCD broadcast

Sequential specifications of objects with some restrictions

- Update consistency
- Generic algorithm
- Snapshot Asset Transfer etc.
- Sequential consistency
Sequential Specifications of Objects

What? Why? How does it relate to causality?

Key ideas:

1. Specify the behaviour of distributed objects assuming a sequential execution (operations are totally ordered) → sequential specification

2. Relate the behaviour of the actual distributed implementation more or less strongly to the sequential specification → consistency criterion
Sequential Specifications of Objects

To specify an object:

- What are its possible states? ($Q$)
- What is its initial state? ($q_0$)
- What are the possible operations?
- How do operations mutate the state and what do they return?
A First Example: A Stack

$q_0 = \epsilon$

$q \xrightarrow{\text{push}(x)/\text{true}} q.x$

$q.x \xrightarrow{\text{pop}/x} q$

$\epsilon \xrightarrow{\text{pop}/\bot} \epsilon$

Notation: $\epsilon$ is the empty word.
An Example With Permissions

Different process may not be able to do the same operations. We introduce a notion of *permissions*. 

Example: single-writer multi-reader snapshot.

A state \( q \) is a map from processes to values.

\[ q[p] \text{write}(i, x) / \text{true} \rightarrow q[i] \leftarrow x \]

\[ q[p] \text{write}(j, x) / \text{false} \rightarrow q[j] \neq i \rightarrow q \]

Notation: \( q[i] \leftarrow x \) is the map \( q \) modified with \( q[i] = x \).
An Example With Permissions

Different process may not be able to do the same operations. We introduce a notion of *permissions*.

Example: single-writer multi-reader snapshot. A state \( q \) is a map from processes to values.

\[
q \xrightarrow{p_i: \text{write}(i, x) / \text{true}} q[i \leftarrow x]
\]

\[
q \xrightarrow{p_i: \text{write}(j, x) / \text{false}} q \quad \text{if } j \neq i
\]

\[
q \xrightarrow{p: \text{snapshot}/q} q
\]

Notation: \( q[i \leftarrow x] \) is the map \( q \) modified with \( q[i] = x \).
We require that read and write operations be clearly differentiated.
We require that read and write operations be clearly differentiated.

Read operations:
- May return any value
- Do not change the state

\[
q \xrightarrow{p: \text{read op}/r} q
\]
We require that read and write operations be clearly differentiated.

Read operations:
- May return any value
- Do not change the state

\[
q \xrightarrow{p:\text{read op}/r} q
\]

Write operations:
- Either return \texttt{true} and possibly change the state
- Or return \texttt{false} and keep the same state

\[
q \xrightarrow{p:\text{write op}/\texttt{true}} q'
\]
\[
q \xrightarrow{p:\text{write op}/\texttt{false}} q
\]
A state $q$ is a map from accounts $a$ to balances.

An account $a$ may have several owners, noted $\text{owners}(a)$.

\[
\begin{align*}
q & \xrightarrow{p:\text{transfer}(a,b,v)/\text{true}} q \quad \text{if } p \in \text{owners}(a) \text{ and } v \leq q[a] \\
q & \xrightarrow{p:\text{transfer}(a,b,v)/\text{false}} q \quad \text{if } p \notin \text{owners}(a) \text{ or } v > q[a] \\
q & \xrightarrow{p:\text{balance}(a)/q[a]} q
\end{align*}
\]
### Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ₯</td>
<td>100 ₯</td>
<td>10 000 ₯</td>
</tr>
</tbody>
</table>
## Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ฿</td>
<td>100 ฿</td>
<td>10 000 ฿</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td>+50 ฿</td>
<td>-50 ฿</td>
<td></td>
</tr>
<tr>
<td><code>true</code></td>
<td>100 ฿</td>
<td>150 ฿</td>
<td>9 950 ฿</td>
</tr>
</tbody>
</table>
## Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ₣</td>
<td>100 ₣</td>
<td>10 000 ₣</td>
</tr>
<tr>
<td>C1 true</td>
<td>100 ₣</td>
<td>+50 ₣</td>
<td>-50 ₣</td>
</tr>
<tr>
<td></td>
<td>150 ₣</td>
<td>150 ₣</td>
<td>9 950 ₣</td>
</tr>
<tr>
<td>C2 true</td>
<td>+50 ₣</td>
<td></td>
<td>-50 ₣</td>
</tr>
<tr>
<td></td>
<td>150 ₣</td>
<td>150 ₣</td>
<td>9 900 ₣</td>
</tr>
</tbody>
</table>
# Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₷</td>
<td>100 ₷</td>
<td>10 000 ₷</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ₷</td>
<td>-50 ₷</td>
</tr>
<tr>
<td>true</td>
<td>100 ₷</td>
<td>150 ₷</td>
<td>9 950 ₷</td>
</tr>
<tr>
<td>C2</td>
<td>+50 ₷</td>
<td></td>
<td>-50 ₷</td>
</tr>
<tr>
<td>true</td>
<td>150 ₷</td>
<td>150 ₷</td>
<td>9 900 ₷</td>
</tr>
<tr>
<td>A1</td>
<td>-200 ₷</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>impossible!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ฿</td>
<td>100 ฿</td>
<td>10 000 ฿</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ฿</td>
<td>-50 ฿</td>
</tr>
<tr>
<td>true</td>
<td>100 ฿</td>
<td>150 ฿</td>
<td>9 950 ฿</td>
</tr>
<tr>
<td>C2</td>
<td>+50 ฿</td>
<td></td>
<td>-50 ฿</td>
</tr>
<tr>
<td>true</td>
<td>150 ฿</td>
<td>150 ฿</td>
<td>9 900 ฿</td>
</tr>
<tr>
<td>A1</td>
<td>-200 ฿</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>false</strong></td>
<td>impossible!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>+100 ฿</td>
<td>-100 ฿</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>250 ฿</td>
<td>50 ฿</td>
<td>9 900 ฿</td>
</tr>
</tbody>
</table>
## Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₷</td>
<td>100 ₷</td>
<td>10 000 ₷</td>
</tr>
<tr>
<td>C2</td>
<td>+50 ₷</td>
<td></td>
<td>-50 ₷</td>
</tr>
<tr>
<td>true</td>
<td>150 ₷</td>
<td>100 ₷</td>
<td>9 950 ₷</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ₷</td>
<td>-50 ₷</td>
</tr>
<tr>
<td>true</td>
<td>150 ₷</td>
<td>150 ₷</td>
<td>9 900 ₷</td>
</tr>
<tr>
<td>A1</td>
<td>-200 ₷</td>
<td>+200 ₷</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>impossible!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>+100 ₷</td>
<td>-100 ₷</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>250 ₷</td>
<td>50 ₷</td>
<td>9 900 ₷</td>
</tr>
</tbody>
</table>
### Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 C</td>
<td>100 C</td>
<td>10 000 C</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>-200 C</td>
<td>+200 C</td>
<td></td>
</tr>
<tr>
<td><strong>false</strong></td>
<td>impossible!</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td>+50 C</td>
<td>-50 C</td>
<td></td>
</tr>
<tr>
<td><strong>true</strong></td>
<td>100 C</td>
<td>150 C</td>
<td>9 950 C</td>
</tr>
<tr>
<td><strong>C2</strong></td>
<td>+50 C</td>
<td></td>
<td>-50 C</td>
</tr>
<tr>
<td><strong>true</strong></td>
<td>150 C</td>
<td>150 C</td>
<td>9 900 C</td>
</tr>
<tr>
<td><strong>B1</strong></td>
<td>+100 C</td>
<td>-100 C</td>
<td></td>
</tr>
<tr>
<td><strong>true</strong></td>
<td>250 C</td>
<td>50 C</td>
<td>9 900 C</td>
</tr>
</tbody>
</table>
### Order is Not (That) Important

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ฿</td>
<td>100 ฿</td>
<td>10 000 ฿</td>
</tr>
<tr>
<td>C1</td>
<td>100 ฿</td>
<td><strong>+50 ฿</strong></td>
<td>-50 ฿</td>
</tr>
<tr>
<td>true</td>
<td>100 ฿</td>
<td>150 ฿</td>
<td>9 950 ฿</td>
</tr>
<tr>
<td>C2</td>
<td><strong>+50 ฿</strong></td>
<td></td>
<td>-50 ฿</td>
</tr>
<tr>
<td>true</td>
<td>150 ฿</td>
<td>150 ฿</td>
<td>9 900 ฿</td>
</tr>
<tr>
<td>B1</td>
<td><strong>+100 ฿</strong></td>
<td>-100 ฿</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>250 ฿</td>
<td>50 ฿</td>
<td>9 900 ฿</td>
</tr>
<tr>
<td>A1</td>
<td>-200 ฿</td>
<td><strong>+200 ฿</strong></td>
<td></td>
</tr>
<tr>
<td>true</td>
<td><strong>50 ฿</strong></td>
<td>250 ฿</td>
<td>9 900 ฿</td>
</tr>
</tbody>
</table>
Order is Not (That) Important

If:

we can execute write operations in another order
and each operation still returns the same value
(true or false, success or error)

Then:

the final state is the same

We call this property **commit-bound order independance** (CBOI)
A problem with shared account:

<table>
<thead>
<tr>
<th></th>
<th>Alice &amp; Bob</th>
<th>Carol</th>
<th>Dave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ฿</td>
<td>10,000 ฿</td>
<td>100 ฿</td>
</tr>
</tbody>
</table>
A problem with shared account:

<table>
<thead>
<tr>
<th></th>
<th>Alice &amp; Bob</th>
<th>Carol</th>
<th>Dave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ¢</td>
<td>10 000 ¢</td>
<td>100 ¢</td>
</tr>
<tr>
<td>Alice 1</td>
<td>-50 ¢</td>
<td></td>
<td>+50 ¢</td>
</tr>
<tr>
<td>true</td>
<td>50 ¢</td>
<td>10 000 ¢</td>
<td>150 ¢</td>
</tr>
</tbody>
</table>
A problem with shared account:

<table>
<thead>
<tr>
<th></th>
<th>Alice &amp; Bob</th>
<th>Carol</th>
<th>Dave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ₪</td>
<td>10 000 ₪</td>
<td>100 ₪</td>
</tr>
<tr>
<td>Alice 1</td>
<td>-50 ₪</td>
<td></td>
<td>+50 ₪</td>
</tr>
</tbody>
</table>
A problem with shared account:

<table>
<thead>
<tr>
<th></th>
<th>Alice &amp; Bob</th>
<th>Carol</th>
<th>Dave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ฿</td>
<td>10,000 ฿</td>
<td>100 ฿</td>
</tr>
<tr>
<td>Bob 1</td>
<td>-80 ฿</td>
<td>+80 ฿</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>20 ฿</td>
<td>10,080 ฿</td>
<td>100 ฿</td>
</tr>
<tr>
<td>Alice 1</td>
<td>-50 ฿</td>
<td></td>
<td>+50 ฿</td>
</tr>
</tbody>
</table>
A problem with shared account:

<table>
<thead>
<tr>
<th></th>
<th>Alice &amp; Bob</th>
<th>Carol</th>
<th>Dave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ₶</td>
<td>10 000 ₶</td>
<td>100 ₶</td>
</tr>
<tr>
<td>Bob 1</td>
<td>-80 ₶</td>
<td>+80 ₶</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>20 ₶</td>
<td>10 080 ₶</td>
<td>100 ₶</td>
</tr>
<tr>
<td>Alice 1</td>
<td>-50 ₶</td>
<td></td>
<td>+50 ₶</td>
</tr>
<tr>
<td>false</td>
<td>impossible!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We must require an additional property:

If:

- a process executes a write operation \( o \)
  - \( o \) succeeds (returns \texttt{true} )

Then, if:

- we add operations by other processes before \( o \)

Then:

- \( o \) still succeeds

We call this property \texttt{local commit stability} (LC-stability)
In the case of AT2:

At most one process must be allowed to withdraw from any given account

(i.e. at most one owner per account)
In the case of AT2:

At most one process must be allowed to withdraw from any given account

(i.e. at most one owner per account)

CBOI + LC-stable is the condition under which we can implement the object without consensus
In a Distributed System

If:

a correct process $p_i$ invokes an operation $o$
and
that operation succeeds locally (at $p_i$)

Then:

all other correct processes will also be able to
apply $o$ successfully on their local state

Because:

- **Reliable broadcast:** all other processes will eventually see all the
  operations that happened before $o$ at $p_i$
- **LC-stable:** all operations that $p_i$ hadn’t seen yet when it executed $o$
cannot prevent $o$ from succeeding
Thus, the following requirements on the event processing order:

- **FIFO**: the operations of one process must be handled at other processes in the same order as they were invoked (future operations by the same process may prevent the current operation)

- **Causal**: if $p_i$ saw some operations before invoking $o$, then other processes must also process these operations before processing $o$

Total order (i.e. consensus) is not required!
Byzantine systems: $BAMP_{n,t}[\ldots]$
Definition of Causal Order

\[ p_i \rightarrow m \rightarrow m' \rightarrow_M m' \]

\[ p_i \rightarrow m \rightarrow m' \rightarrow_M m' \]
### Definition of Causal Order

- **BC-FIFO.** If a correct process $p_i$ bf-delivers messages $m$ before $m'$ from the same process $p_k$ (possibly Byzantine), then no correct process bf-delivers $m'$ before $m$. Moreover, if $p_k$ is correct, it bf-broadcast $m$ before $m'$.

  and

- **BC-Local-Order.** If a correct process bc-delivers first a message $m$ and later bc-broadcasts a message $m'$, then no correct process bc-delivers $m'$ before $m$. 
Definition of Causal Order

- **BC-FIFO.** If a correct process $p_i$ bf-delivers messages $m$ before $m'$ from the same process $p_k$ (possibly Byzantine), then no correct process bf-delivers $m'$ before $m$. Moreover, if $p_k$ is correct, it bf-broadcasts $m$ before $m'$.

and

- **BC-Local-Order.** If a correct process bc-delivers first a message $m$ and later bc-broadcasts a message $m'$, then no correct process bc-delivers $m'$ before $m$.

No local order guarantee for Byzantine processes!
These situations only make sense for non-Byzantine processes.
Correct sender: the correct processes deliver messages in the order they were sent
Understanding the FIFO property

- **Correct sender:** the correct processes deliver messages in the order they were sent

- **Byzantine sender:** the correct processes deliver messages in a certain order, the same at all correct processes
This simple algorithm implements only the FIFO order property.

\[
\textbf{init} \quad sn_i \leftarrow 0; \quad del_i \leftarrow [0, \ldots, 0].
\]

\textbf{operation} \texttt{bf\_broadcast}(m) \textbf{is}

(1) \quad sn_i \leftarrow sn_i + 1;
(2) \quad \texttt{br\_broadcast}(\langle i, sn_i \rangle, m).

\textbf{when} \quad (\langle j, sn \rangle, m) \textbf{is} \texttt{br\_delivered} \textbf{from} \ p_j \ \textbf{do}

(3) \quad \texttt{wait}(sn = del_i[j] + 1);
(4) \quad \texttt{bf\_delivery} \texttt{of} \ m \textbf{from} \ p_j;
(5) \quad del_i[j] \leftarrow del_i[j] + 1.
Implementation of causal order: using the causal barrier set.

\[ cb(m_2) = \{ id(m), id(m'), id(m'') \} \]
Causal Order Graph

Also called set of immediate causal predecessors.

\[ cb(m_4) = \{ id(m_2), id(m_3) \} \]
init $cb_i \leftarrow \emptyset$; $sn_i \leftarrow 0$; $del_i \leftarrow [0, \ldots, 0]$.

operation bc_broadcast($m$) at $p_i$ is
(1) $sn_i \leftarrow sn_i + 1$;
(2) br_broadcast($\langle i, sn_i \rangle, cb(m), m$) where $cb(m) = cb_i$;
(3) $cb_i \leftarrow \emptyset$.

when ($\langle j, sn \rangle, cb(m), m$) is br_delivered from $p_j$ at $p_i$
(4) wait($($sn = del_i[j] + 1$) \land (\forall \langle k, sn' \rangle \in cb(m) : del_i[k] \geq sn'$)$);
(5) $cb_i \leftarrow (cb_i \setminus cb(m)) \cup \{\langle j, sn \rangle\}$;
(6) local bc_delivery of $m$ from $p_j$;
(7) $del_i[j] \leftarrow del_i[j] + 1$. 

Byzantine processes can always lie on their causal barrier, e.g. by pretending that they haven't yet received a previous message.
Byzantine Causal Broadcast Algorithm

\begin{align*}
\text{init} & \quad \text{\texttt{cb}}_i \leftarrow \emptyset; \text{\texttt{sn}}_i \leftarrow 0; \text{\texttt{del}}_i \leftarrow [0, \ldots, 0]. \\
\text{operation} \quad & \text{bc\_broadcast}(m) \text{ at } p_i \text{ is} \\
(1) & \quad \text{\texttt{sn}}_i \leftarrow \text{\texttt{sn}}_i + 1; \\
(2) & \quad \text{br\_broadcast}(\langle i, \text{\texttt{sn}}_i \rangle, \text{\texttt{cb}}(m), m) \text{ where } \text{\texttt{cb}}(m) = \text{\texttt{cb}}_i; \\
(3) & \quad \text{\texttt{cb}}_i \leftarrow \emptyset. \\
\text{when} \quad & (\langle j, \text{\texttt{sn}} \rangle, \text{\texttt{cb}}(m), m) \text{ is br\_delivered from } p_j \text{ at } p_i \\
(4) & \quad \text{\texttt{wait}}((\text{\texttt{sn}} = \text{\texttt{del}}_i[j] + 1) \land (\forall \langle k, \text{\texttt{sn}}' \rangle \in \text{\texttt{cb}}(m) : \text{\texttt{del}}_i[k] \geq \text{\texttt{sn}}')); \\
(5) & \quad \text{\texttt{cb}}_i \leftarrow (\text{\texttt{cb}}_i \setminus \text{\texttt{cb}}(m)) \cup \{\langle j, \text{\texttt{sn}} \rangle\}; \\
(6) & \quad \text{local bc\_delivery of } m \text{ from } p_j; \\
(7) & \quad \text{\texttt{del}}_i[j] \leftarrow \text{\texttt{del}}_i[j] + 1.
\end{align*}

Byzantine processes can always lie on their causal barrier, e.g. by pretending that they haven’t yet received a previous message.
A First Generic Algorithm

Byzantine systems: $BAMP_{n,t}[\ldots]$

- Reliable broadcast (Bracha, 1984)
- $BAMP_{n,t}[t < n/3]$ Causal broadcast
- $BAMP_{n,t}[t < n/4]$ FIFO broadcast
- SCD broadcast

Sequential specifications of objects with some restrictions

- Update consistency
- Generic algorithm
- Snapshot Asset Transfer etc.
- Sequential consistency
Update consistency:

The local state of a process (on which it executes read operations) must be the result of applying a certain set of write operations following the sequential specification starting at $q_0$ (with no requirement on the order in which different processes see different operations).

Strong update consistency:

(the interesting one)

Same, and also:

All write operations must be eventually processed at all nodes.
Causal: if \( p_i \) saw some operations before invoking \( o \), then other processes must also process these operations before processing \( o \)

This is not even a strong requirement!
<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ø</td>
<td>100 ø</td>
<td>10 000 ø</td>
</tr>
<tr>
<td>C1</td>
<td>100 ø</td>
<td>+50 ø</td>
<td>-50 ø</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>100 ø</th>
<th>150 ø</th>
<th>9 950 ø</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### On Causality

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ₧</td>
<td>100 ₧</td>
<td>10 000 ₧</td>
</tr>
<tr>
<td>C1</td>
<td>100 ₧</td>
<td>+50 ₧</td>
<td>-50 ₧</td>
</tr>
<tr>
<td></td>
<td>100 ₧</td>
<td>150 ₧</td>
<td>9 950 ₧</td>
</tr>
<tr>
<td>A1</td>
<td>-150 ₧</td>
<td>+150 ₧</td>
<td></td>
</tr>
</tbody>
</table>
## On Causality

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>100 ₪</td>
<td>100 ₪</td>
<td>10 000 ₪</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ₪</td>
<td>-50 ₪</td>
</tr>
<tr>
<td></td>
<td>100 ₪</td>
<td>150 ₪</td>
<td>9 950 ₪</td>
</tr>
<tr>
<td>A1</td>
<td>-150 ₪</td>
<td>+150 ₪</td>
<td></td>
</tr>
<tr>
<td>on hold...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## On Causality

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₣</td>
<td>100 ₣</td>
<td>10 000 ₣</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td></td>
<td>+50 ₣</td>
<td>-50 ₣</td>
</tr>
<tr>
<td></td>
<td>100 ₣</td>
<td>150 ₣</td>
<td>9 950 ₣</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>-150 ₣</td>
<td>+150 ₣</td>
<td></td>
</tr>
<tr>
<td><strong>on hold...</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C2</strong></td>
<td></td>
<td>+50 ₣</td>
<td>-50 ₣</td>
</tr>
</tbody>
</table>
## On Causality

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ᶾ</td>
<td>100 ᶾ</td>
<td>10 000 ᶾ</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td>+50 ᶾ</td>
<td>-50 ᶾ</td>
</tr>
<tr>
<td></td>
<td>100 ᶾ</td>
<td>150 ᶾ</td>
<td>9 950 ᶾ</td>
</tr>
<tr>
<td>C2</td>
<td>+50 ᶾ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>-150 ᶾ</td>
<td>+150 ᶾ</td>
<td></td>
</tr>
</tbody>
</table>
# On Causality

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td>100 ₣</td>
<td>100 ₣</td>
<td>10 000 ₣</td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td></td>
<td>+50 ₣</td>
<td>-50 ₣</td>
</tr>
<tr>
<td></td>
<td>100 ₣</td>
<td>150 ₣</td>
<td>9 950 ₣</td>
</tr>
<tr>
<td><strong>C2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td></td>
<td>+150 ₣</td>
<td>-50 ₣</td>
</tr>
<tr>
<td></td>
<td>0 ₣</td>
<td>300 ₣</td>
<td>9 900 ₣</td>
</tr>
</tbody>
</table>
A First Algorithm

\begin{align*}
\text{init} \quad & \text{state}_i \leftarrow q_0; \quad \text{del}_i[1..n] \leftarrow [0, \ldots, 0]; \quad s_i \leftarrow 0. \\
\text{operation } o \in W \text{ at } p_i \text{ is} \quad & \text{– } o \text{ is a write operation} \\
(1) \quad & \text{if } (\exists q' : \text{state}_i \xrightarrow{p_i:o/true} q') \\
(2) \quad & \text{then } s_i \leftarrow s_i + 1; \quad \text{done}_i \leftarrow \text{false}; \\
(3) \quad & \text{br_broadcast } (i, s_i, o); \\
(4) \quad & \text{wait } (\text{done}_i); \quad \text{return}(\text{commit}); \\
(5) \quad & \text{else return}(\text{abort}). \\
\text{operation } o \in R \text{ at } p_i \text{ is} \quad & \text{– } o \text{ is a read operation} \\
(6) \quad & \text{let } r \text{ such that } \text{state}_i \xrightarrow{p_i:o/r} \text{state}_i; \\
(7) \quad & \text{return}(r). \\
\text{when } (\langle j, s_i \rangle, o) \text{ is br_delivered from } p_j \text{ do} \\
(8) \quad & \text{wait } (\exists q' : \text{state}_i \xrightarrow{p_j:o/true} q') \land (\text{del}_i[j] + 1 = s_i); \\
(9) \quad & \text{state}_i \leftarrow q'; \quad \text{del}_i[j] \leftarrow s_i; \\
(10) \quad & \text{if } (j = i) \text{ then } \text{done}_i \leftarrow \text{true}.
\end{align*}
A First Algorithm

- A direct adaptation of the first algorithm given for AT2
- Guarantees Strong Update Consistency for any CBOI LC-stable sequential object
A First Algorithm

- A direct adaptation of the first algorithm given for AT2

- Guarantees Strong Update Consistency for any CBOI LC-stable sequential object

- Issue: a Byzantine process may send an invalid operation and the network will never reject it, it will be stuck forever!
If we want correct processes to be able to reject some operations, they must agree on which operations to accept or to reject.

- **Solution 1**: use a consensus algorithm
  (bad solution: consensus requires additional computing power, equivalent to total order!)
If we want correct processes to be able to reject some operations, they must agree on which operations to accept or to reject.

- **Solution 1**: use a consensus algorithm
  (bad solution: consensus requires additional computing power, equivalent to total order!)

- **Solution 2**: use **Byzantine causal broadcast** and leverage the causality information
  (the values of $cb$ associated with each message)
All processes see the same causality graph:

<table>
<thead>
<tr>
<th>Alice</th>
<th>Carol</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 C</td>
<td>10 000 C</td>
<td>100 C</td>
</tr>
</tbody>
</table>

- $\text{tx}(C, A, 50C)$ ✓
- $\text{tx}(C, B, 50C)$ ✓
- $\text{tx}(A, B, 150C)$ ×
- $\text{tx}(B, A, 150C)$ ✓
- $\text{tx}(C, A, 50C)$ ✓
Algorithm:

[omitted]
Algorithm:

[omitted]

When an operation is invoked: same as previously, except that we use **BC-broadcast** instead of **BR-broadcast**
Algorithm:

omitted

When an operation is invoked: same as previously, except that we use BC-broadcast instead of BR-broadcast

When an operation is received from another process:

1. Extract the set of operations that are predecessors in the causal graph
Algorithm:

When an operation is invoked: same as previously, except that we use **BC-broadcast** instead of **BR-broadcast**

When an operation is received from another process:

1. Extract the set of operations that are predecessors in the causal graph
2. Apply them in a topological sort order following the sequential specification starting from $q_0$, leading to a state $q$
A Better Algorithm with Byzantine Causal Broadcast

Algorithm:

[omitted]

When an operation is invoked: same as previously, except that we use **BC-broadcast** instead of **BR-broadcast**

When an operation is received from another process:

1. Extract the set of operations that are predecessors in the causal graph
2. Apply them in a topological sort order following the sequential specification starting from $q_0$, leading to a state $q$
3. If the new operation can be applied successfully at state $q$, apply it on current $state_i$
Algorithm:

When an operation is invoked: same as previously, except that we use **BC-broadcast** instead of **BR-broadcast**

When an operation is received from another process:

1. Extract the set of operations that are predecessors in the causal graph
2. Apply them in a topological sort order following the sequential specification starting from $q_0$, leading to a state $q$
3. If the new operation can be applied successfully at state $q$, apply it on current state $i$
4. Otherwise, reject the operation
Byzantine systems: $BAMP_{n,t}[\ldots]$

Reliable broadcast (Bracha, 1984)

$BAMP_{n,t}[t < n/3]$

Causal broadcast

FIFO broadcast

$BAMP_{n,t}[t < n/4]$

SCD broadcast

Sequential specifications of objects with some restrictions

Update consistency
Generic algorithm

Snapshot Asset Transfer etc.

Sequential consistency
Set-Constraint Delivery Broadcast

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

**Example:** \( \{ m_1, m_2 \}, \{ m_3 \}, \{ m_4, m_5, m_6 \}, \ldots \)
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 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 \( m_1 \).
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 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:**

  \(p_i: \quad \{m_1, m_2\}, \{m_3\}, \{m_4, m_5, m_6\}, \ldots\)

  \(p_j: \quad \{m_1, m_2, m_3\}, \{m_4\}, \{m_5, m_6\}, \ldots\)
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 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
  \]
Algorithm:

[omitted, really a bit complex, check the paper]
Algorithm:

[omitted, really a bit complex, check the paper]

Main idea: wait for a majority of processes to agree that \( m_1 \) comes before \( m_2 \) if we want to bscd-deliver \( m_1 \) before \( m_2 \).
BSCD-Broadcast Algorithm

Algorithm:

[omitted, really a bit complex, check the paper]

Main idea: wait for a majority of processes to agree that $m_1$ comes before $m_2$ if we want to bscd-deliver $m_1$ before $m_2$.

- Does not require consensus: BSCD-broadcast is strictly weaker than total order broadcast;
BSCD-Broadcast Algorithm

Algorithm:

[omitted, really a bit complex, check the paper]

Main idea: wait for a majority of processes to agree that $m_1$ comes before $m_2$ if we want to bscd-deliver $m_1$ before $m_2$.

- Does not require consensus: BSCD-broadcast is strictly weaker than total order broadcast;
- Our algorithm requires $t < n/4$. 
Sequential Consistency

**Update consistency:**

The local state of a process (on which it executes read operations) must be the result of applying a certain set of write operations following the sequential specification starting at $q_0$ (with no requirement on the order in which different processes see different operations).
Update consistency:

The local state of a process (on which it executes read operations) must be the result of applying a certain set of write operations following the sequential specification starting at $q_0$ (with no requirement on the order in which different processes see different operations).

Sequential consistency:

There exists a total order of operations (not necessarily known to processes) such that processes get the same return values to the operations they invoke as if all the operations were executed in that order following the sequential specification.
Sequential Consistency with SCD

$\pi_i$

$\{w_1, w_2\}$

$\{w_3\}$

$\pi_j$

$\{w_1\}$

$\{w_2, w_3\}$

Sequential Order

$r_1$ $r'_1$ $w_1$ $r'_2$ $w_2$ $r_2$ $w_3$ $r'_3$ $r_3$
Computing Power: the Snapshot Object

\begin{verbatim}
init \textit{reg}_i \leftarrow [\bot, \ldots, \bot]; \textit{wsn}_i \leftarrow [0, \ldots, 0].

\textbf{operation \textit{snapshot}() is}
(1) \textit{done}_i \leftarrow \text{false}; \text{bscd\_broadcast} \text{SYNC(); \text{wait}(\text{done}_i));}
(2) \text{return}(\textit{reg}_i[1..n]).

\textbf{operation \textit{write}(v) is}
(3) \textit{done}_i \leftarrow \text{false}; \text{bscd\_broadcast} \text{WRITE(v); \text{wait}(\text{done}_i)).

\textbf{when} \textit{ms} = \{ \langle j_1, \textit{sn}_1, \text{WRITE}(v_1) \rangle, \ldots, \langle j_x, \textit{sn}_x, \text{WRITE}(v_x) \rangle, \\
\langle j_{x+1}, \textit{sn}_{x+1}, \text{SYNC()} \rangle, \ldots, \langle j_y, \textit{sn}_y, \text{SYNC()} \rangle \} \text{ is bscd-delivered do}
(4) \text{for each message} \langle j, \textit{sn}_j, \text{WRITE}(v) \rangle \in \textit{ms} \text{ do}
(5) \quad \text{if} (\textit{wsn}_i[j] < \textit{sn}_j) \text{ then} \textit{reg}_i[j] \leftarrow v; \textit{wsn}_i[j] \leftarrow \textit{sn}_j \text{ end if}
(6) \text{end for};
(7) \text{if} \exists \ell : j_\ell = i \text{ then} \textit{done}_i \leftarrow \text{true} \text{ end if}.
\end{verbatim}

A linearizable Byzantine-tolerant SWMR snapshot object.
We can build a generic algorithm provided that:

- Write operations commute
- Write operations always apply their update, they cannot fail/be refused
We can build a generic algorithm provided that:

- Write operations commute
- Write operations always apply their update, they cannot fail/be refused

A generic algorithm for commit/abort sequential specs: probably not so simple.
Conclusion

- Broadcast abstractions: powerful primitives
- Hierarchy: BRB < BFIFO < (BC, BSCD), BC ⊥ BSCD
- Sequential specifications of objects
- How broadcast primitives relate to consistency criteria
- AT2: causality + FIFO is the necessary condition, not total order like Blockchain (too strong)
- Update consistency ↔ BC broadcast, generic algorithm
- No generic algorithm for sequential consistency (yet)
- SCD ↔ atomic read/write registers