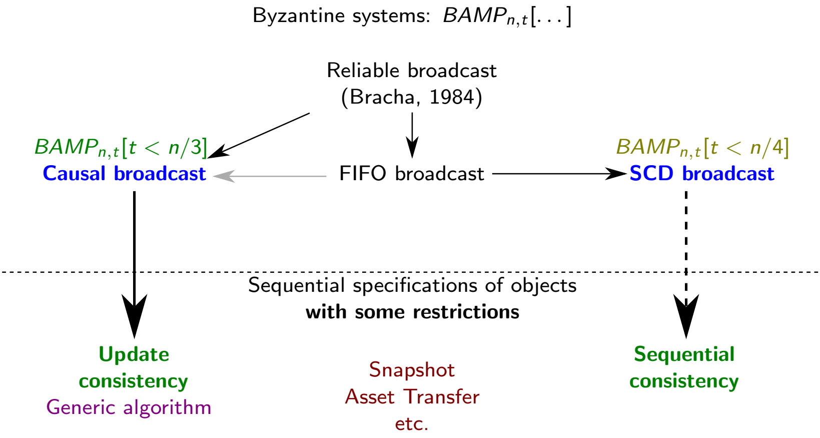
High Level Primitives in Byzantine Systems

Alex Auvolat

WIDE Team Seminar, May 16-17, 2019

Outline


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

An example: Trustworthy Asset Transfer (AT2)

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵

Guarantees:

- No money is ever created or destroyed
- An account always has a balance ≥ 0

An example: Trustworthy Asset Transfer (AT2)

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵

Guarantees:

- No money is ever created or destroyed
- An account always has a balance ≥ 0

An example: Trustworthy Asset Transfer (AT2)

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵

Guarantees:

- No money is ever created or destroyed
- An account always has a balance ≥ 0

An example: Trustworthy Asset Transfer (AT2)

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	

Guarantees:

- No money is ever created or destroyed
- An account always has a balance ≥ 0

An example: Trustworthy Asset Transfer (AT2)

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
~~A1~~	~~-150 ₵~~	~~+150 ₵~~	
	impossible!		

Guarantees:

- No money is ever created or destroyed
- An account always has a balance ≥ 0

An example: Trustworthy Asset Transfer (AT2)

AT2: introduced in *Guerraoui et al., 2018*

Question: general modular approach?

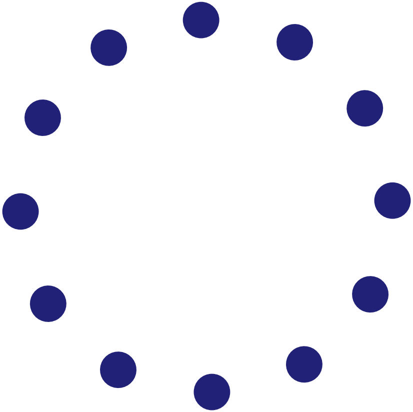
An example: Trustworthy Asset Transfer (AT2)

AT2: introduced in *Guerraoui et al., 2018*

Question: general modular approach?

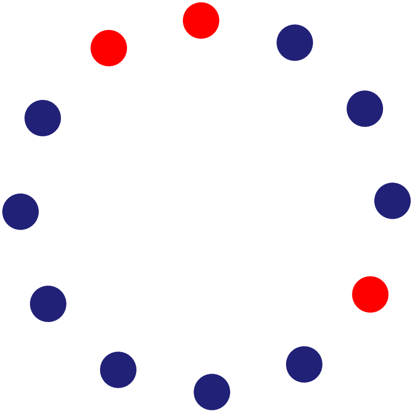
$\rightarrow$ high-level communication abstractions in Byzantine systems
$\rightarrow$ consistency criteria in Byzantine systems

System Model



n nodes (or processes)

System Model



Up to t Byzantine nodes

Byzantine Behaviour

- 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.

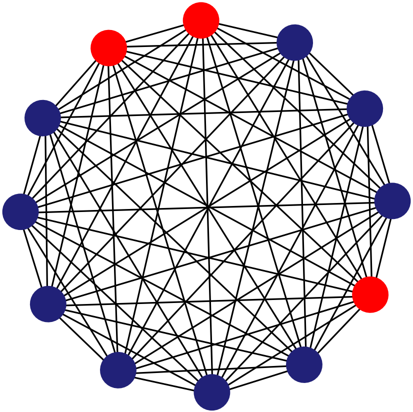
Byzantine Behaviour

- A Byzantine process <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.

Byzantine Behaviour

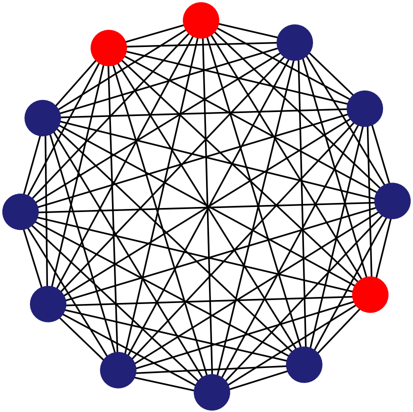
- A Byzantine process may not pretend to be another process.
  The system model guarantees the identity of the sender.

System Model



Authenticated point-to-point links

System Model



Authenticated point-to-point links
Asynchronous message passing

Byzantine Reliable Broadcast

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-broadcast 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.

Tolerance to Byzantine Nodes

- 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 multple instances of BRB so that each process can send several messages.

Multi-shot BRB

We need multple 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 multple 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)

AT2 over BRB

init $init[1..n]$: constant array where $init[k]$ is the initial value of p_k account;
$hist_i[1..n] \leftarrow [\emptyset, \cdots, \emptyset]$; $del_i[1..n] \leftarrow [0, \cdots, 0]$; $sn_i \leftarrow 0$.

operation transfer(j, v) **is**
(1) **if** (balance$(i) < v$)
(2) **then return**(`abort`)
(3) **else** $sn_i \leftarrow sn_i + 1$; $done_i \leftarrow$ `false`;
(4) br_broadcast $(\langle i, sn_i \rangle, \text{TRANSFER}(j, v))$;
(5) **wait** $(done_i)$; **return**(`commit`).

when $(\langle j, sn \rangle, \text{TRANSFER}(k, v))$ **is** br_delivered **from** p_j **do**
(6) **wait** (balance$(j) \geq v) \wedge (del_i[j] + 1 = sn)$;
(7) $hist_i[j] \leftarrow hist_i[j] \cup \{\langle k, v \rangle\}$;
(8) $del_i[j] \leftarrow sn$;
(9) **if** $(j = i)$ **then** $done_i \leftarrow$ `true`.

internal function balance(j) **is**
(10) **return**$(init[j] + \sum_\ell \sum_{\langle j, v_x \rangle \in hist_i[\ell]} v_x - \sum_{\langle -, v_x \rangle \in hist_i[j]} v_x)$.

AT2 Behaviour

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵

AT2 Behaviour

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	

AT2 Behaviour

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	
	on hold...		

AT2 Behaviour

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	
	on hold...		
C2	+50 ₵		-50 ₵

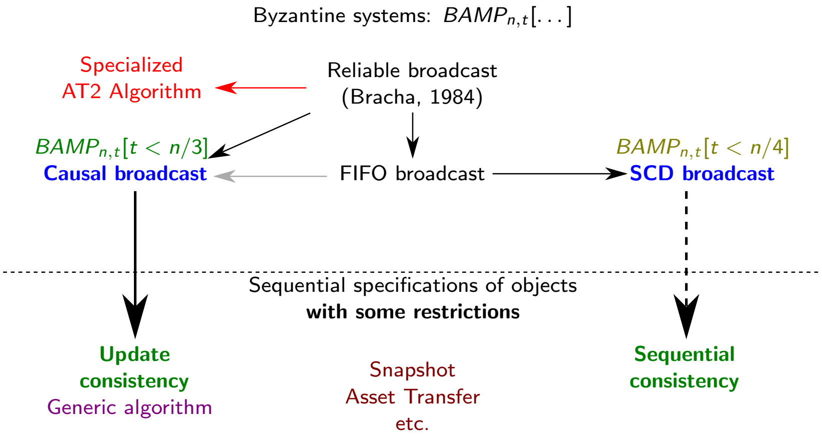
AT2 Behaviour

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50 ₵
A1	-150 ₵	+150 ₵	

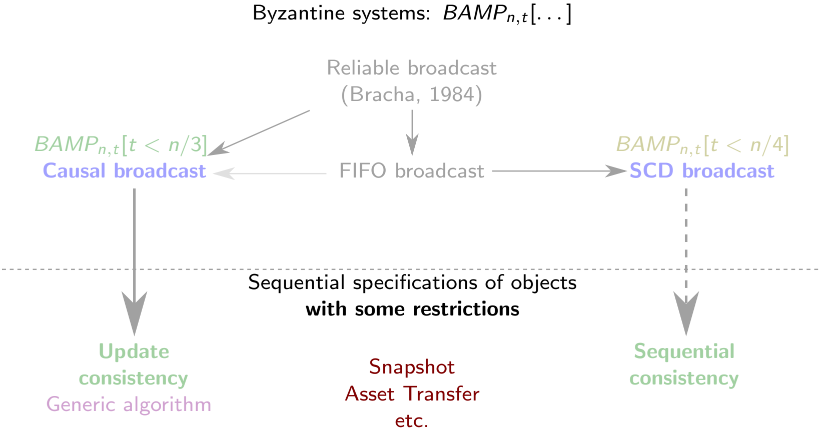
AT2 Behaviour

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50 ₵
A1	-150 ₵	+150 ₵	
	0 ₵	300 ₵	9 900 ₵

Let's Split It Up



Let's Split It Up



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)
   $\rightarrow$ sequential specification

2. Relate the behaviour of the actual distributed implementation more or less strongly to the sequential specification
   $\rightarrow$ 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

$$q_0 = \epsilon$$

$$\begin{array}{ccc} q & \xrightarrow{\text{push}(x)/\texttt{true}} & q.x \\ q.x & \xrightarrow{\text{pop}/x} & q \\ \epsilon & \xrightarrow{\text{pop}/\bot} & \epsilon \end{array}$$

Notation: ϵ 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*.

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.

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

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

$$q \xrightarrow{p:\text{snapshot}/q} q$$

Notation: $q[i \leftarrow x]$ is the map q modified with $q[i] = x$.

Read/Write Commit/Abort

We require that read and write operations be clearly differentiated.

Read/Write Commit/Abort

We require that read and write operations be clearly differentiated.

Read operations:

- May return any value
- Do not change the state

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

Read/Write Commit/Abort

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 `true` and possibly change the state
- Or return `false` and keep the same state

$$q \xrightarrow{p\text{:write op}/\texttt{true}} q'$$
$$q \xrightarrow{p\text{:write op}/\texttt{false}} q$$

Example: Multi-AT2 Specification

A state q is a map from accounts a to balances.

An account a may have several owners, noted owners(a).

$$q \xrightarrow[\text{if } p\in\text{owners}(a) \text{ and } v\leq q[a]]{p\text{:transfer}(a,b,v)/\texttt{true}} q\begin{bmatrix} a\leftarrow q[a]-v \\ b\leftarrow q[b]+v \end{bmatrix}$$

$$q \xrightarrow[\text{if } p\notin\text{owners}(a) \text{ or } v>q[a]]{p\text{:transfer}(a,b,v)/\texttt{false}} q$$

$$q \xrightarrow{p\text{:balance}(a)/q[a]} q$$

Order is Not (That) Important

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵

Order is Not (That) Important

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
`true`	100 ₵	150 ₵	9 950 ₵

Order is Not (That) Important

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
`true`	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50₵
`true`	150 ₵	150 ₵	9 900 ₵

Order is Not (That) Important

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
`true`	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50₵
`true`	150 ₵	150 ₵	9 900 ₵
~~A1~~	~~-200 ₵~~	~~+200 ₵~~	
`false`	**impossible!**		

Order is Not (That) Important

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
`true`	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50₵
`true`	150 ₵	150 ₵	9 900 ₵
~~A1~~	~~-200 ₵~~	~~+200 ₵~~	
`false`	**impossible!**		
B1	+100 ₵	-100 ₵	
`true`	250 ₵	50 ₵	9 900 ₵

Order is Not (That) Important

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C2	+50 ₵		-50 ₵
`true`	150 ₵	100 ₵	9 950 ₵
C1		+50 ₵	-50₵
`true`	150 ₵	150 ₵	9 900 ₵
~~A1~~	~~-200 ₵~~	~~+200 ₵~~	
`false`	**impossible!**		
B1	+100 ₵	-100 ₵	
`true`	250 ₵	50 ₵	9 900 ₵

Order is Not (That) Important

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
~~A1~~	~~-200 ₵~~	~~+200 ₵~~	
`false`	**impossible!**		
C1		+50 ₵	-50₵
`true`	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50 ₵
`true`	150 ₵	150 ₵	9 900 ₵
B1	+100 ₵	-100 ₵	
`true`	250 ₵	50 ₵	9 900 ₵

Order is Not (That) Important

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
`true`	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50₵
`true`	150 ₵	150 ₵	9 900 ₵
B1	+100 ₵	-100 ₵	
`true`	250 ₵	50 ₵	9 900 ₵
A1	-200 ₵	+200 ₵	
`true`	**50** ₵	**250** ₵	9 900 ₵

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)

Problem

A problem with shared account:

	Alice & Bob	**Carol**	**Dave**
Initial	100 ₵	10 000 ₵	100 ₵

Problem

A problem with shared account:

	Alice & Bob	**Carol**	**Dave**
Initial	100 ₵	10 000 ₵	100 ₵
Alice 1	-50 ₵		+50 ₵
`true`	50 ₵	10 000 ₵	150 ₵

Problem

A problem with shared account:

	Alice & Bob	**Carol**	**Dave**
Initial	100 ₵	10 000 ₵	100 ₵
Alice 1	-50 ₵		+50 ₵

Problem

A problem with shared account:

	Alice & Bob	**Carol**	**Dave**
Initial	100 ₵	10 000 ₵	100 ₵
Bob 1	-80 ₵	+80 ₵	
`true`	20 ₵	10 080 ₵	100 ₵
Alice 1	-50 ₵		+50 ₵

Problem

A problem with shared account:

	Alice & Bob	**Carol**	**Dave**
Initial	100 ₵	10 000 ₵	100 ₵
Bob 1	-80 ₵	+80 ₵	
`true`	20 ₵	10 080 ₵	100 ₵
~~Alice 1~~	~~-50 ₵~~		~~+50 ₵~~
`false`	**impossible!**		

Solution

We must require an additionnal property:

If:

a process executes a write operation o
o succeeds (returns `true`)

Then, if:

we add operations by other processes before o

Then:

o still succeeds

We call this property local commit stability (LC-stability)

Solution

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)

Solution

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 concensus

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 happenned 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

Order of Operations

Thus, the following requirements on the event processing order:

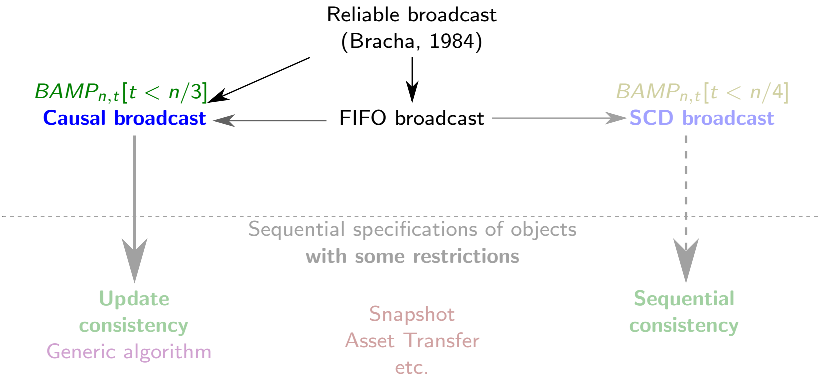
- FIFO: the operations of one process must be handled at other processes in the same order as they were invoked
  (future operations by the same process may prevent the current operation)

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

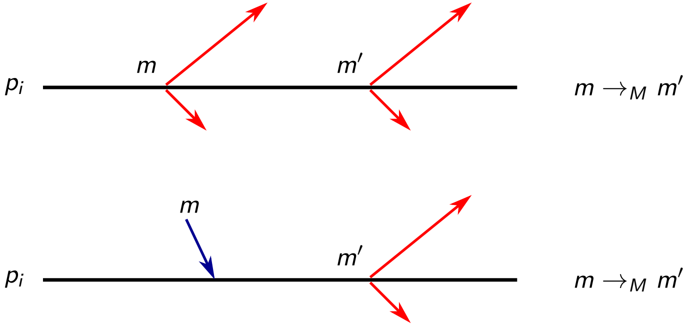
Total order (i.e. concensus) is not required!

Back to Communication Primitives

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



Definition of Causal Order



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

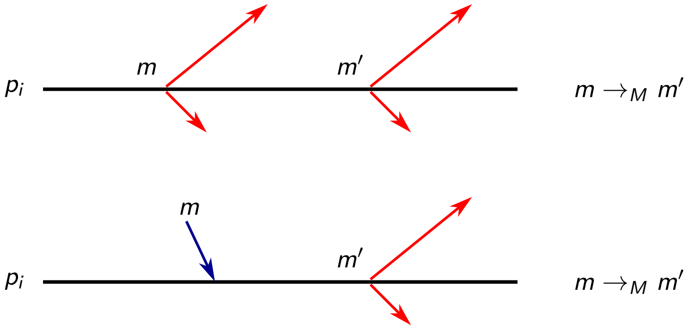
- BC-FIFO. If a correct process p_i bf-delivers messages m before m' from the same process p_k (possibly Byzantine), then no correct process bf-delivers m' before m.
  Moreover, if p_k is correct, it bf-broadcast m before m'.

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.

No local order guarantee for Byzantine processes!

Definition of Causal Order



These situations only make sense for non-Byzantine processes.

Understanding the FIFO property

- 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

BF-broadcast algorithm

This simple algorithm implements only the FIFO order property.

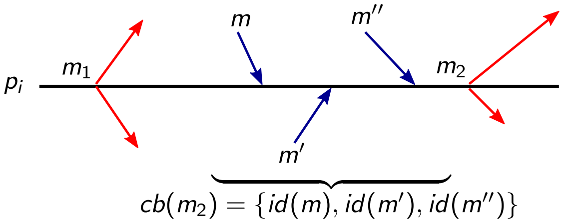
init $sn_i \leftarrow 0$; $del_i \leftarrow [0, \ldots, 0]$.

operation bf_broadcast(m) **is**
(1) $sn_i \leftarrow sn_i + 1$;
(2) br_broadcast($\langle i, sn_i \rangle, m$).

when ($\langle j, sn \rangle, m$) **is** br_delivered **from** p_j **do**
(3) **wait**$\big(sn = del_i[j] + 1\big)$;
(4) bf_delivery of m **from** p_j;
(5) $del_i[j] \leftarrow del_i[j] + 1$.

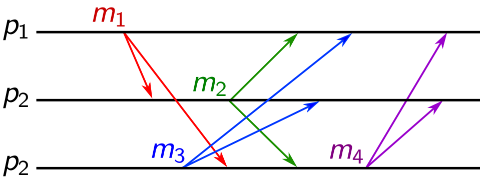
Causal Barrier

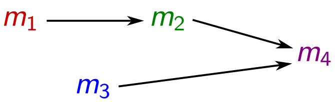
Implementation of causal order: using the causal barrier set.



Causal Order Graph

Also called set of immediate causal predecessors.





$$cb(m_4) = \{id(m_2), id(m_3)\}$$

Byzantine Causal Broadcast Algorithm

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**$\big((sn = del_i[j] + 1) \wedge (\forall \langle k, sn' \rangle \in cb(m) :\ del_i[k] \geq sn')\big)$;
(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 Causal Broadcast Algorithm

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**$\big((sn = del_i[j] + 1) \wedge (\forall \ \langle k, sn' \rangle \in cb(m) : \ del_i[k] \geq sn')\big)$;

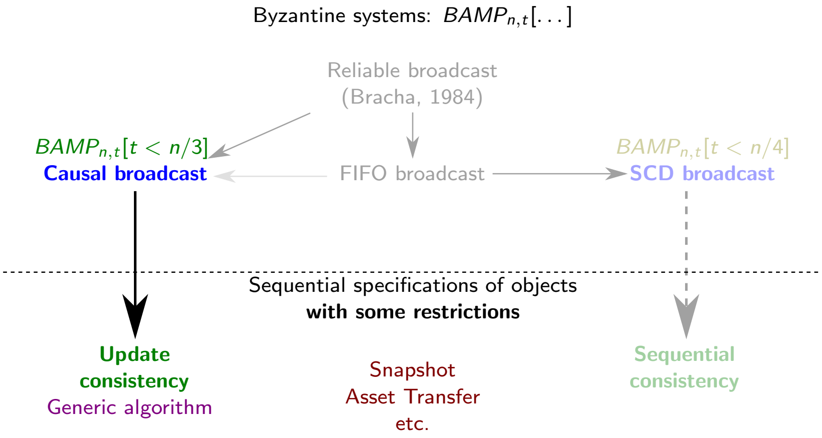
(5) $cb_i \leftarrow (cb_i \setminus cb(m)) \cup \{\langle j, sn \rangle\}$;

(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

A First Generic Algorithm



Update 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

On Causality

- 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!

On Causality

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵

On Causality

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	

On Causality

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	
	on hold...		

On Causality

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
A1	-150 ₵	+150 ₵	
	on hold...		
C2	+50 ₵		-50 ₵

On Causality

	Alice	**Bob**	**Carol**
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50 ₵
A1	-150 ₵	+150 ₵	

On Causality

	Alice	Bob	Carol
Initial	100 ₵	100 ₵	10 000 ₵
C1		+50 ₵	-50 ₵
	100 ₵	150 ₵	9 950 ₵
C2	+50 ₵		-50 ₵
A1	-150 ₵	+150 ₵	
	0 ₵	300 ₵	9 900 ₵

init $state_i \leftarrow q_0$; $del_i[1..n] \leftarrow [0, \cdots, 0]$; $sn_i \leftarrow 0$.

operation $o \in W$ **at** p_i **is** — *o is a write operation*

if $(\exists q' : state_i \xrightarrow{p_i:o/\texttt{true}} q')$
  then $sn_i \leftarrow sn_i + 1$; $done_i \leftarrow \texttt{false}$;
    br_broadcast $(\langle i, sn_i \rangle, o)$;
    wait $(done_i)$; **return**$(\texttt{commit})$;
  else return$(\texttt{abort})$.

operation $o \in R$ **at** p_i **is** — *o is a read operation*

let r **such that** $state_i \xrightarrow{p_i:o/r} state_i$;
return(r).

when $(\langle j, sn \rangle, o)$ **is** br_delivered **from** p_j **do**

wait $(\exists q' : state_i \xrightarrow{p_j:o/\texttt{true}} q') \wedge (del_i[j] + 1 = sn)$;
$state_i \leftarrow q'$; $del_i[j] \leftarrow sn$;
if $(j = i)$ **then** $done_i \leftarrow \texttt{true}$.

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!

A Better Algorithm

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 concensus algorithm
  (bad solution: concensus requires additionnal computing power, equivalent to total order!)

A Better Algorithm

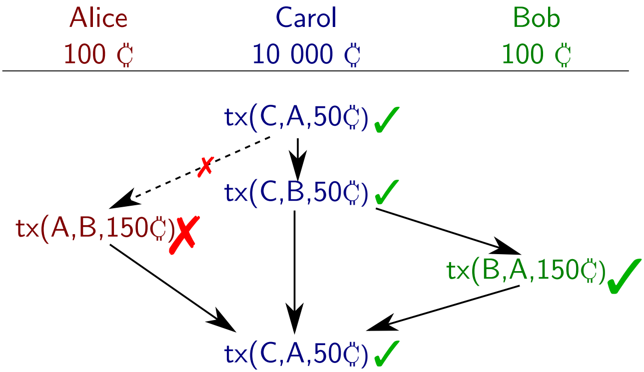
If we want correct processes to be able to reject some operations, they must agree on which operations to accept or to reject.

- Solution 1: use a concensus algorithm
  (bad solution: concensus requires additionnal 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)

A Better Algorithm with Byzantine Causal Broadcast

All processes see the same causality graph:



A Better Algorithm with Byzantine Causal Broadcast

Algorithm:

[omitted]

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**

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

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

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 $state_i$

A Better Algorithm with Byzantine Causal Broadcast

Algorithm:

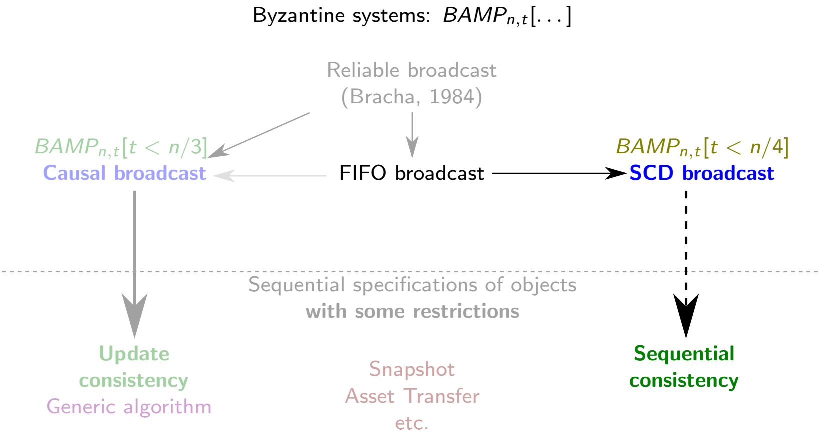
[omitted]

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 $state_i$
4. Otherwise, reject the operation

Set-Constraint Delivery Broadcast



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

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

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

BSCD-Broadcast Algorithm

Algorithm:

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

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.

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 concensus: 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 concensus: 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)

Sequential Consistency

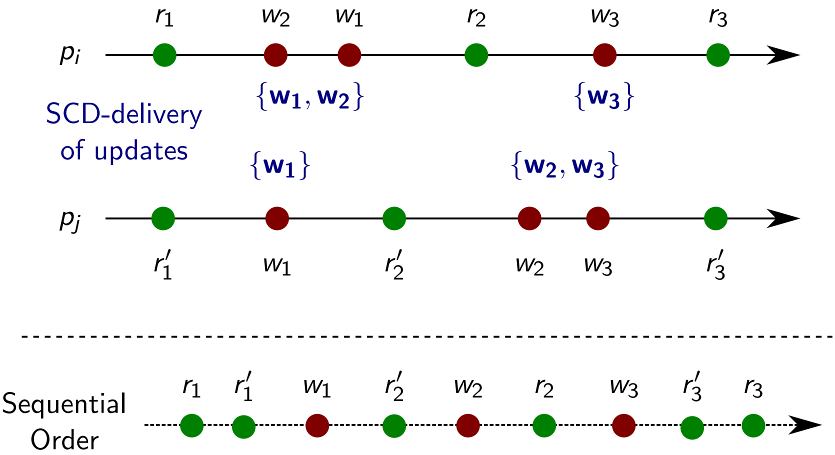
Update consistency:

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)

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



Computing Power: the Snapshot Object

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

operation snapshot() **is**
(1) $done_i \leftarrow$ `false`; bscd_broadcast SYNC(); **wait**($done_i$);
(2) **return**($reg_i[1..n]$).

operation write(v) **is**
(3) $done_i \leftarrow$ `false`; bscd_broadcast WRITE(v); **wait**($done_i$).

when $ms = \{ \langle j_1, sn_1, \text{WRITE}(v_1)\rangle, \ldots, \langle j_x, sn_x, \text{WRITE}(v_x)\rangle,$
$\langle j_{x+1}, sn_{x+1}, \text{SYNC}()\rangle, \ldots, \langle j_y, sn_y, \text{SYNC}()\rangle \}$
is bscd-delivered do
(4) **for each message** $\langle j, snj, \text{WRITE}(v)\rangle \in ms$ **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** $done_i \leftarrow$ `true` **end if**.

A linearizable Byzantine-tolerant SWMR snapshot object.

Generic Algorithm

We can build a generic algorithm provided that:

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

Generic Algorithm

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$\perp$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 $\leftrightarrow$ BC broadcast, generic algorithm
- No generic algorithm for sequential consistency (yet)
- SCD $\leftrightarrow$ atomic read/write registers