Ordered Multicast

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo
Causal and Total Orderings

We previously saw these definitions:

Causal ordering preserves the causal relationship between messages.

Formally:
If \( \text{MSend}(m, G) \rightarrow \text{MSend}(m', G) \), then every correct process that delivers \( m' \) must have already delivered \( m \).

Total ordering preserves the order of all messages across all processes.

Formally:
If any correct process delivers \( m \) before \( m' \), then every correct process that delivers \( m' \) must have already delivered \( m \).
The ISIS System

The ISIS system defined causally and totally ordered multicast [1, 2].

It uses vector clocks for causal ordering.

Causal ordering is imposed on multicast message delivery only.

It uses a two-phase protocol for total ordering.

Processes cooperatively arrive at a total ordering for each message.
Safety and Liveness

These protocols maintain two properties [1]:

- **Safety**: The protocol never delivers messages in an order that violates the ordering constraints.
- **Liveness**: The protocol never delays a message indefinitely.

The latter requires that every message is delivered.

This can be accomplished via, e.g., \( R_{MCAST} \).
ISIS VT Protocol

ISIS defines several protocols for causal ordering.

The VT protocol [1] addresses causal ordering with static group membership.

More complicated ISIS protocols handle:

- Dynamic group membership (processes joining and leaving)
- Overlapping groups with causal relationships
- Causal total ordering
Vector Timestamps

The VT protocol uses **vector timestamps**.

Every message is transmitted with its timestamp.

These timestamps **look just like** our FIFO timestamps!

However, **vector entries are causally updated** like vector clocks.

Messages must be **held back** if they do not arrive in causal order.
VT Protocol Vector Timestamps

The VT protocol maintains a vector $VT$ for:

- Every message $m$: $VT(m) = \langle 1, \ldots, n \rangle$
- Every process $p$: $VT(p) = \langle 1, \ldots, n \rangle$

Each entry in the vector represents process $p_i$ for $0 \leq i < n$ processes.

Every process maintains its own vector.

Every process increments only its own timestamp.
VT Protocol Methods

**VT_Send**($m$, $G$) at $p_i$:
- Increment $VT(p_i)[i]$
- $VT(m) = VT(p_i)$
- $R_MCast(VT(p_i) || m, G)$

**VT_Deliver**($m$) from $p_i$ at $p_j \neq p_i$:
- Increment $VT(p_j)[i]$
- Deliver($m$)
- Run hold back queue

**VT_Recv**($m$) from $p_i$ at $p_j \neq p_i$:
- If $VT(m) = VT(p_j)[i] + 1$ and
- $\forall k \neq i : VT(m)[k] \leq VT(p_j)[k]$
  - $VT_Deliver(m)$
- Else:
  - Hold back $m$
VT Example

$p_3$ queue:

$p_1$ sends a message with timestamp $\langle 1, 0, 0 \rangle$. 
VT Example

$p_1$ queue: $\langle 1, 0, 0 \rangle$

$p_2$ queue: $\langle 1, 0, 0 \rangle$

$p_3$ queue: $\langle 1, 0, 0 \rangle$

$p_2$ receives and delivers message $\langle 1, 0, 0 \rangle$. 
VT Example

$p_3$ queue:

Now

$p_2$ sends a message $\langle 1, 1, 0 \rangle$. 
VT Example

$p_3$ queue: $\langle 1, 1, 0 \rangle$

Now

$p_1$ receives and delivers $\langle 1, 1, 0 \rangle$.
$p_3$ holds back $\langle 1, 1, 0 \rangle$. 
VT Example

$p_3$ queue: $\langle 1, 1, 0 \rangle$

$p_3$ receives and delivers message $\langle 1, 0, 0 \rangle$.
$p_3$ delivers held back message $\langle 1, 1, 0 \rangle$. 
Total Ordering with a Sequencer

Total ordering can be achieved through a sequencer.

Each time a process $p_i$ wants to send a message:

1. $p_i$ sends $m$ to the sequencer
2. The sequencer sends $m$ with FIFO multicast

All messages are received FIFO from the sequencer.

What are the disadvantages of this?
ISIS ABCAST Protocol


It doesn’t require a central sequencer!

It uses a two phase protocol.

Each message is:

- Transmitted without ordering
- Ordered and delivered

Messages are queued but undeliverable until ordered.

The ordering of each message is managed by its sender.
Intuition

Every host $p_i$ in ABCAST maintains a logical clock $T_i$.

Every message has two associated timestamps:

- A proposed timestamp $T^p_m$, set when it is transmitted
- An ordered timestamp $T^o_m$, set when it is deliverable

The ordered timestamp of a message is the maximum clock on all processes when its proposal was received.

The clock ticks for:

- Sending an unordered message
- Receiving an unordered message
ABCAST Phase 1

In the first phase of message transmission:

1. Process $p_i$ increments its local clock.
2. Process $p_i$ adds $m$ to its queue as undeliverable at priority $T_{m}^p = T_i$.
3. Process $p_i$ multicasts $m$ with timestamp $T_{m}^p$ from its local clock.

Every process $p_j, j \neq i$ eventually receives $m$ and:

1. $p_j$ sets its local timestamp to $\text{MAX}(T_j, T_{m}^p)$.
2. $p_j$ increments $T_j$.
3. $p_j$ adds $m$ to its queue as undeliverable at priority $T_j$.
4. $p_j$ sends an acknowledgment for $m$ with timestamp $T_j$ to $p_i$. 
Phase 1 Example

$p_1$ sends message $m_1$ with timestamp 1.
Phase 1 Example

$p_2$ receives $m_1$. 
Phase 1 Example

\( p_2 \) returns an acknowledgment for \( m \) with timestamp 2.
Phase 1 Example

$p_1$ receives $p_2$'s acknowledgment.
Phase 1 Example

$p_3$ receives $m_1$. 

Now
Phase 1 Example

$p_3$ returns an acknowledgment for $m$ with timestamp 2.
Phase 1 Example

$p_1$ receives $p_3$'s acknowledgment.
ABCAST Phase 2

In the **second phase** of message $m$ transmission from $p_i$:

$p_i$ performs the following steps:

1. compute the maximum timestamp $T_m^o$ from all acknowledgments of $m$
2. multicast the ordered message $m$ with timestamp $T_m^o$

Each process $p_j, j \neq i$ eventually receives the ordered $m$ and:

1. marks $m$ as deliverable
2. delivers all deliverable messages **at the front** of its queue
Tie Breaking

There’s one wrinkle: What if two processes propose the same max timestamp for different messages?

Those messages are tie broken by appending the process ID.

If timestamp $T_i = k$, it is treated as $k.i$; for example:
Timestamp 3 at $p_2$ is 3.2.

We will elide this suffix when it is irrelevant.
ABCAST Example

Now

Queue:

\( m_{1:1} \)

\( p_1 \)

\( m_{1:1} \)

\( p_2 \)

\( m_{1:1} \)

\( p_3 \)
ABC\textsc{CAST} Example

\begin{itemize}
\item $p_2$ receives $m_1$ and enqueues it at priority 2.
\end{itemize}
ABCAST Example

Queues:

\(m_1:1\)

\(m_1:2\)
ABCAST Example

$p_1$ receives $A_1:2$ from $p_2$ and takes the max priority for $m_1$.

$p_2$ sends $m_2$ with $T^p_2 = 3$. 
ABCAST Example

Now

$p_1$ and $p_3$ enqueue $m_2$ with priority 4.
ABCAST Example

 Queues:

\[ m_1:2 \quad m_2:4 \]

\[ m_1:2 \quad m_2:3 \]

\[ m_2:4 \]
$p_2$ receives $A_2:4$ from $p_3$ and takes the max priority for $m_2$. $p_3$ receives $m_1$ and enqueues it at priority 5.
ABCAST Example

$p_2$ receives the final ack for $m_2$ and orders it at 4.
ABCAST Example

Queues:

- $m_1:2$
- $m_2:4$
- $m_1:5$
- $m_2:4$
ABCAST Example

$p_1$ receives the final ack for $m_1$ and orders it at 5.
ABCAST Example

$\rho_1$ receives the ordering for $m_2$ and delivers $m_2$ then $m_1$. $\rho_3$ receives the ordering for $m_2$ and delivers it.
**ABCAST Example**

$p_3$ receives the ordering for $m_1$ and delivers it.
ABCAST Example

$p_2$ receives the ordering for $m_1$ and delivers $m_2$ then $m_1$. 
ABCAST Example

Queue:

All processes delivered $m_2$ followed by $m_1$. 
Sketch for Correctness

Why does this work?

Every process knows that $m$ will be delivered no earlier than its acknowledged ordering.

The sequencer for $m$ takes the maximum observed timestamp.

When a deliverable message is in the queue the local process will never propose an earlier sequence!
Summary

- **Safety** means constraints will never be violated
- **Liveness** means every message is eventually delivered
- ISIS provides **causally** and **totally** ordered multicast
- The VT protocol uses **vector clocks** to causally order
- ISIS ABCAST uses **distributed sequencing** to totally order
References

Required Readings


Optional Readings

References II

Copyright 2023 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.