Causality

Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu
MESSAGE ORDERING

- Assume messages received reliably but not necessarily in order
- Communication is direct (P2P)
MULTICAST

User Process

Message

User Process

Message

User Process

Same Message directed multiple processes

Receiver of multicast message can also multicast

Consistency?

Consistency in 2-computer case?

communicator.toOthers(new ARemoteInput(theNextInput));
Unicast

User Process

Message

directed to a single process

User Process

communicator.toUser("alice", new AFloorControlRequest());
Decoupling Reliability and Order

User Process

Message

User Process

Sliding window ensures in-order processing and reliable delivery
Assume reliable delivery
How in-order processing?
Not practical to decouple, but will help us draw out principles for the N-Computer, multicast case, where reliability assumed

`communicator.toUser("alice", new AFloorControlRequest());`
OUT OF ORDER UNICAST

PC 1

Done?
Lunch?
Yes, No

PC 2

Divergent state!
Sequence numbers!

Lunch?
Done?
Yes, No

Yes, No
Yes, No
Out of Order Unicast (Review)

PC 1

- Done?
- Lunch?
- Yes, No

PC 2

- Lunch?
- Done?
- Yes, No

Divergent state!
Sequence numbers!
**Unicast Sequence Numbers**

Each computer pair keeps count of #messages sent to other party.

Send message: increment and attach local count as time stamp.

Each computer keeps last processed remote # and ordered buffer for other party.

When message received, put message in ordered buffer:

1. If buffer empty or message# != successor (remote#) return.
2. Remove message from buffer, process it.
3. remote# = message#.
4. Go to 1.

PC1 Buffer:
- Done? 0
- Lunch? 1

PC2 Buffer:
- Done? 1
- Lunch? 2

PC1:
- Local 2
- Remote 0

PC2:
- Local 0
- Remote 2
N-USER UNICAST

Supports pairwise connections (IMs)

Group IM?
OUT OF ORDER MULTICAST

PC 1
 Done?
 Lunch?
 Yes, No

PC 2
 Done?
 Lunch?
 Yes, No

PC 3
 Lunch?
 Done?
 Yes, No

Causal relations

M₁ causes (<) M₂ if M₂ sent from site S after M₁ received or generated at Site S

Causal multicast: At all sites, if M₁ < M₂, M₂ should be processed after M₁

How to detect and ensure causal broadcast?
**Real-Time Scalar Stamp**

- **PC 1**: Done?, 10:01
- **PC 2**: Lunch?, 10:02
- **PC 3**: Yes, No, 10:04

- **Lunch?**
  - PC 1: Yes, No
  - PC 2: Yes, No
  - PC 3: Yes, No

- **Done?**
  - PC 1: Yes, No
  - PC 2: Yes, No
  - PC 3: Yes, No

**M_2 caused by M_1 → RTS (M_2) > RTS (M_1)**

- Process received messages in order of RTS

- **Not RTS (M_2) > RTS (M_1)** → M_2 caused by M_1

- Do not know if there is an in-transit previous message, how long to wait?

- Clocks at different sites not synchronized!
**Message History Stamp**

- **Stamp**: Global message id + history of ids of sent/received msgs
- **Global id**: unique site it + sequence number
- **History can get large and compression needed**
- **Simpler scheme possible if message not multicast to arbitrary user set**

**Diagram**:
- PC 1: Local 1, Done?, (1,1), {}
- PC 2: Local 1, Lunch?, (2,1), {(1,1)}
- PC 3: Local 1, Lunch?, (3,1), {(1,1), (2,1)}

- Questions:
  - Done?
  - Lunch?
  - Yes, No
MESSAGE HISTORY STAMP (REVIEW)

Stamp: Global message id + history of ids of sent/received msgs

Global id: unique site it + sequence number

History can get large and compression needed

Simpler scheme possible if message not multicast to arbitrary user set

Done?
Lunch?
Yes, No

Done?
Lunch?
Yes, No

Done?
Lunch?
Yes, No

PC 1
Local 1

PC 2

PC 3
Local 1

Yes, No, (3,1), {(1,1), (2,1)}

Lunch?, (2, 1), {(1,1)}

Done?, (1, 1), {}
**Global Scalar Id: Logical Clock, Assuming All Messages Broadcast**

Every site keeps a global id initialized to 0.

When a site generates a message, it increments id and time stamps message with it.

A site delivers a message if its global id is the successor of current global id; otherwise, it buffers the message to be delivered later.

On delivering/processing a received message, a site sets its global id to the message id.

It should allow detection of concurrent messages as soon as they arrive.

It should not deliver a message before its cause.

If no concurrent messages ever occur, this scheme should work.

Causal broadcast does not indicate what should happen with concurrent messages – immediate delivery, (fatal) error.
Every site keeps a receive count for each site.

When a site generates a message, it sends the sender and count of the last message it received.

A site delivers a message if the received count for the site is the same as its count for that site; otherwise it buffers the message for later delivery.

On delivering/processing a received message, a site increments local count for that site.
Every site keeps a receive count for each site.

When a site generates a message, it sends the sender and count of the last message it received.

A site delivers a message if the received count for the site is the same as its count for that site; otherwise, it buffers the message for later delivery.

On delivering/processing a received message, a site increments its local count for that site.

A message may have multiple causes, and this scheme sends only the most recent cause.
**From History to Vector Timestamps**

Assume: each message broadcast to all other users in an app session.

IM and many other apps follow this assumption.

Counts of sent/received messages replace history.

---

**PC 1**

Done?, (1, 1), {}  
Lunch?, (2, 1), {(1,1)}

**PC 2**

Yes, No, (3,1), {(1,1), (2,1)}

**PC 3**

Local 0

Local 1

Local 1
EXTENSION OF TWO-USER UNICAST

- Local: 2
- Remote: 0

- PC 1
  - Done? 1
  - Lunch? 2

- PC 2
  - Local: 0
  - Remote: 2

Number for each user: vector timestamp
Message has vector timestamp
Ordered buffer for messages arriving early

Need:
< and == for sorting buffer
increment operation before sending message and after receiving message
succ function for picking next received msg
VECTOR TIME STAMPS

\[ v = (x_1, \ldots, x^n) \text{ at Site } S_j \to \]

\[ v^1 = (a_1, \ldots, a^n) \]

\[ = \]

\[ v^2 = (b_1, \ldots, b^n) \]

\[ v^1 = (a_1, \ldots, a^n) \]

\[ < \]

\[ v^2 = (b_1, \ldots, b^n) \]

Site \( S^i \) has broadcast \( x^i \) messages to other sites and for all \( 1 \leq i \leq n, i \neq j \) Site \( S^i \) has received \( x^i \) messages from Site \( S^i \)

for all \( 1 \leq i \leq n, a^i = b^i \)

for all \( 1 \leq i \leq n, a^i \leq b^i \)

for some \( 1 \leq i \leq n, a^i < b^i \)

Vector time stamps do not create total order

Possible that \( a^i < b^i \) and \( a^i > b^i \) for some \( 1 \leq i, j \leq n \) \( \to \) concurrent message, \( v^1 \ || \ v^2 \)

Causal broadcast does not impose order on concurrent messages

For causal broadcast, will assume no concurrent messages are generated.
EXTENSION OF TWO-USER UNICAST

Number for each user: vector timestamp

Message has vector time stamp

Ordered buffer for messages arriving early

Need:

< and == for sorting buffer

increment operation before sending message and after receiving message

succ function for picking next received msg

Done?

Lunch?

Local 2
Remote 0
PC 1

Local 0
Remote 2
PC 2

PC2 Buffer

Done? 1
Lunch? 2

Done? 0
Lunch? 1
**INCREMENT AND SUCCESSOR**

\[ v_1 = (a_1, \ldots, a_n) \]

is a successor of

\[ v_2 = (b_1, \ldots, b_n) \]

There exists \( 1 \leq i \leq n, a_i = 1 + b_i \)

for all \( j \neq i, a_j = b_j \)

\[ \text{inc}(i, v = (a_1, \ldots, a_i, \ldots, a_n)) \rightarrow v = (a_1, \ldots, a_i + 1, \ldots, a_n) \]

A message has multiple successors

Inc with respect to a site
**Unicast vs. Multicast**

Each pair of communicating computers keeps a count of how many messages it has sent to other party and next expected remote# for other party.

**Send message:** attach and increment local count.

Each site keeps ordered buffer for other party.

When message received, put message in ordered buffer.

1. If buffer empty or message# != successor (remote#) return.
2. Remove message from buffer, process it.
3. remote# ← message#
4. Go to 1.

Each site\(^i\) keeps a local vector time stamp, \(v^i = (i^1, \ldots, i^n)\).

**Send message:** increment \(i^i\) and attach vector time stamp.

Each site\(^i\) keeps ordered buffer\(^i\) for all parties.

When message received from site \(i\), put message in ordered buffer.

1. If buffer empty or message TS \(!=\) successor (local TS) return.
2. Remove message from buffer, process it.
3. Local TS\(^i\) ← message TS\(^i\)
4. Go to 1.
Causal Multicast

Received message is put in ordered buffer

1. If buffer empty or message TS != successor (local TS) return

2. Remove message from buffer and process it

3. Local TS\textsuperscript{i} \leftarrow msg TS\textsuperscript{i}

4. Go to 1

Send message: increment \( i^i \) and attach vector time stamp

Done?

PC 1

\( v_1 \)

1 0 0

Lunch?

PC 2

\( v_2 \)

1 1 0

Done?

PC 3

\( v_3 \)

1 1 0

Buffer\textsubscript{1}

Done?

1 0 0

Buffer\textsubscript{2}

Lunch?

1 1 0

Buffer\textsubscript{3}

Lunch?

1 1 0

Done?
**Extra Steps for Implementing Causal Multicast?**

- Implement vector time stamps
- Implement buffer
- Change message sends and receives

**Buffer 1**

- Done? 1 0 0
- Lunch? 1 0 0

**Buffer 2**

- Done? 1 0 0
- Lunch? 1 1 0

**Buffer 3**

- Done? 1 0 0
- Lunch? 1 1 0
SOFTWARE ARCHITECTURE

- Put causal semantics in model?
  - Model has to do the extra steps mentioned in previous slide
  - Model may not want overhead and delay of causality in certain situations
  - Causality not an issue when communication is relayed and model is unaware of routing

- Put causal semantics in communication infrastructure?
  - May want causality in replicated window systems or some other model
SOFTWARE ARCHITECTURE REQUIREMENTS?

- Causality concepts independent of app and comm. infrastructure
- Separation of concerns
- Application code unaware of causality code
- Communication infrastructure unaware of causality
- Can dynamically add, remove, change causality implementation
- Some general pattern beyond causality?
Causality Architecture (Review)

- Communicating app and communication system unaware of causality
- Optional, substitutable intermediary causality module
- Communication system must allow interception and interjection of messages

Causality – Unaware Application

<table>
<thead>
<tr>
<th>Causality-aware proxy</th>
<th>Received messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Done?</td>
</tr>
<tr>
<td></td>
<td>Lunch?</td>
</tr>
</tbody>
</table>

| Done?     | 1 | 0 | 0 |
| Lunch?    | 1 | 1 | 0 |

Causality-unaware communication system

<table>
<thead>
<tr>
<th>Send/Receive</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Done? 1 0 0

Done? 1 0 0

Done? 1 0 0
**INTERJECTION/INTERCEPTION OF MESSAGES**

- **Causality – Unaware Application**
- **Send Filter**
- **Receive Filter**
- **Causality-unaware communication system**

**Interjection/Interception of Messages**

- A sent/received message goes through a send/receive filter in send/receive pipeline
- Default filter simply forwards message to the next stage
- Need a way to replace default filter with custom filters
# Delivery: Un-Filtered or Filtered

<table>
<thead>
<tr>
<th>Demo</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Demo 1: Multi-View Single-User Version of the IM/Editing Tool</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>2</td>
<td>Demo 2: Multi-User IM/Editing Tool</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>3</td>
<td>Demo 3: Replicated Windows</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>4</td>
<td>Demo 4: Jitter and Telepointer Trails</td>
<td>Buffered</td>
</tr>
<tr>
<td>5</td>
<td>Demo 5: Causality</td>
<td>Reordered</td>
</tr>
<tr>
<td>6</td>
<td>Demo 6: Operation Transformation</td>
<td>Transformed</td>
</tr>
<tr>
<td>7</td>
<td>Demo 7: Locking</td>
<td>Unfiltered</td>
</tr>
</tbody>
</table>
Filtering and Extensibility

Sender

SentMessage Filter

processMessage

Sender

Communicator

toOthers

Receiver

Received MessageFilter

objectReceived

Received Message Processor

processMessage

Received Message Processor

filterMessage

Filter interface(s)?
**MESSAGE FILTER INTERFACE**

```java
public interface MessageFilter<MessageType> {
    public void setMessageProcessor(MessageProcessor<MessageType> newVal);
    public void filterMessage(MessageType message);
}
```

- **Called by communication system when new message to be filtered available**
- **Next stage in pipeline, processing the filtered message**
- **ReceivedMessage or SentMessage**
- **Called by communication system when pipeline setup**
public interface MessageProcessor<MessageType> {
    public void processMessage(MessageType theMessage);
}

- Called by message filterer to process message
- ReceivedMessage or SentMessage
- Sent message processor (and succeeding pipeline stages) broadcasts message
- Received message processor (and succeeding pipeline stages) delivers to listeners
FILTERING AND EXTENSIBILITY

Sender

SentMessage Filter

Filter

SentMessage Processor

Communicator

toOthers

processMessage

ReceivedMessage Filter

ReceivedMessage Processor

objectReceived

processMessage

filterMessage

Unfiltered case?
**Default Parameterized Message Filter**

```java
public class AMessageForwarder<MessageType> implements MessageFilter<MessageType> {
    MessageProcessor<MessageType> messageProcessor;

    public void filterMessage(MessageType sentMessage) {
        messageProcessor.processMessage(sentMessage);
    }

    public void setMessageProcessor(MessageProcessor<MessageType> newVal) {
        messageProcessor = newVal;
    }
}
```

- Simply forwards the message
- Instantiated as both sent and receive filter

Can be replaced with custom received and sent filters that modify, buffer and/or reorder messages: e.g. MySentMessageFilter, MyReceivedMessageFilter
**MESSAGE-SPECIFIC FILTERS**

```java
public class AMessageForwarder<MessageType> implements MessageFilter<MessageType> {
    MessageProcessor<MessageType> messageProcessor;

    public void filterMessage(MessageType sentMessage) {
        messageProcessor.processMessage(sentMessage);
    }

    public void setMessageProcessor(MessageProcessor<MessageType> newVal) {
        messageProcessor = newVal;
    }
}
```

- **Instantiated as both sent and receive filter**
- **Simply forwards the message**
- **Can be replaced with custom received and sent filters that modify, buffer and/or reorder messages: e.g. MySentMessageFilter, MyReceivedMessageFilter**
- **Typically different actions for sent and receive filtering (e.g. add time stamp, remove time stamp)**

Message type matters, must know receive and send message types implemented by the communicator.
**GROUPMESSAGE AND SENTMESSAGE**

```java
public interface GroupMessage extends Serializable {
    String getApplicationName();
    Object getUserMessage();
    boolean isUserMessage();
}
```

```java
public interface SentMessage extends GroupMessage{
    ...
}
```

Sent message filter must implement MessageFilter&lt;SentMessage&gt;

If (isUserMessage()) then getUserMessage() is the object sent by remote site

User object will be replaced with a time stamped object by filter

System messages such as client joins and leave status update messages
**ReceivedMessage**

```java
public interface ReceivedMessage extends GroupMessage {
  String getClientName();
  ...
}
```

Receive message filter must implement MessageFilter<ReceivedMessage>

GroupMessage unites SendMessage and ReceiveMessage

If (isUserMessage()) then getUserMessage() is the object sent by remote site

User object will be actual user object extracted from timestamped message

ggetClientName() needed for timestamp-based processing
INTERJECTION/INTERCEPTION OF MESSAGES

A sent/received message goes through a send/receive filter in send/receive pipeline

Default filter simply forwards message to the next stage

Shared data between filters?

Causality – Unaware Application

Send Filter

Receive Filter

Causality-unaware communication system
**Causal Multicast**

1. If buffer empty or message TS != successor (local TS) return
2. Remove message from buffer and process it
3. Local $T_{S_i} \leftarrow$ msg $T_{S_i}$
4. Go to 1

Send message: increment $i^i$ and attach vector time stamp

Received message is put in ordered buffer

**Buffer**

- **Buffer$_1$**
  - Done? 1 0 0
  - Lunch? 1 0 0
- **Buffer$_2$**
  - Done? 1 0 0
  - Lunch? 1 1 0
- **Buffer$_3$**
  - Done? 1 1 0
  - Lunch? 1 1 0

$v_1$

PC 1

$v_2$

PC 2

$v_3$

PC 3
Shared Filter State

Causality – Unaware Application

Send Filter \[\rightarrow\] Causality Manager \[\rightarrow\] Receive Filter

Causality-unaware communication system

Bulk of the work done by shared causality manager

Dynamic steps: intercepting messages

Static steps?

Creation of vector time stamp of correct length
Causality - Unaware Application

Send Filter → Causality Manager

Causality Manager → Receive Filter

clientJoined()

Causality-unaware communication system

When to add component to vector

Creation of vector time stamp of correct length
LISTENING TO CLIENT JOINS

public interface SessionMessageListener {
    void clientJoined(String aClientName, String anApplicationName,
                       String aSessionName, boolean isNewSession, boolean isNewApplication,
                       Collection<String> allUsers);
    void clientLeft(String aClientName, String anApplicationName);
}

communicator.addSessionMessageListener(causalityManager);

Assume first message sent after all members of the session have
joined and no message sent after the first user leaves

Dynamic session changes in causal communication requires
latecomer messages
How to Switch?

Causality – Unaware Application

Forwarder

Causality-unaware communication system

Forwarder
**How to Switch?**

- **Causality–Unaware Application**
  - Send Filter
  - Causality Manager
  - Receive Filter
  - Causality-unaware communication system

- Communicator could provide an API to switch filters
- Communicator is aware of filters and any other component that needs to be switched
- Filter created using factory
- Shared objects can be passed to factories created by programmer
- Can register custom factory with abstract factory to create custom filter

- How to share objects between filters
- How to switch factories?
**FACTORY INTERFACE**

```java
public interface MessageFilterCreator<MessageType> {
    MessageFilter<MessageType> getMessageFilter();
}
```

- Returns object to be created
- Can create a new object each time or return a singleton object
- Common interface for creating sent and receive filters
**Default Parameterized Message Filter Factory**

```java
public class AMessageForwarderCreatorCreator<MessageType> implements MessageFilterCreator<MessageType>{
    public MessageFilter<MessageType> getMessageFilter() {
        return new AMessageForwarder<MessageType>();
    }
}
```

- Can be replaced with custom factories (e.g. MySendFilterCreator, MyReceiveFilterCreator)
- Instantiated as both sent and receive filter factory
**Send Filter (Factory) Selector or Abstract Factory**

```java
public class SentMessageFilterSelector {
    static MessageFilterCreator<SentMessage> filterFactory =
        new AMessageForwarderCreator<SentMessage>;
    public static MessageFilterCreator<SentMessage> getMessageFilterCreator() {
        return filterFactory;
    }
    public static void setMessageFilterCreator(MessageFilterCreator<SentMessage> theFactory) {
        filterFactory = theFactory;
    }
}
```

- Default factory
- Called during construction of send pipeline
- Can be assigned custom send factory

*(SentMessageFilterSelector.setMessageFilterCreator(new MySendFilterCreator()) (before communicator is created))*
Receive Filter (Factory) Selector

```java
public class ReceivedMessageFilterSelector {
    static MessageFilterCreator<ReceivedMessage> filterFactory = 
        new AMessageForwarderCreator<ReceivedMessage>();
    public static MessageFilterCreator<ReceivedMessage> getMessageFilterCreator(){
        return queuerFactory;
    }
    public static void setMessageFilterCreator(MessageFilterCreator<ReceivedMessage> theFactory){
        queuerFactory = theFactory;
    }
}
```

Default factory

Called during construction of receive pipeline

Can be assigned custom receive factory

```
ReceivedMessageFilterSelector.setMessageFilterCreator( new MyReceiveFilterCreator() )
```

(before communicator is created )
**Filters**

Causality module composed of send and receive filters.

Factories for returning filters

Filters can share common state such as site vector time stamp.

Common state can be passed as parameters to factory and filter constructors

**Causality – Unaware Application**

Send/Receive

**Causality-aware proxy**

**Causality-unaware communication system**

**Received messages**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Done?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lunch?</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Done? 1 0 0
UNICAST vs. MULTICAST (Review)

Each pair of communicating computers keeps a count of how many messages it has sent to other party and next expected remote# for other party.

Send message: attach and increment local count.

Each site keeps ordered buffer for other party.

When message received, put message in ordered buffer.

1. If buffer empty or message# != successor (remote#) return.
2. Remove message from buffer, process it.
3. remote# ← message#
4. Go to 1.

Each site\textsuperscript{i} keeps a local vector time stamp, \( v^i = (i^1, \ldots, i^n) \).

Send message: increment \( i^i \) and attach vector time stamp.

Each site\textsuperscript{i} keeps ordered buffer\textsuperscript{i} for all parties.

When message received from site \( i \), put message in ordered buffer.

1. If buffer empty or message TS \( \neq \) successor (local TS) return.
2. Remove message from buffer, process it.
3. Local TS\textsubscript{i} ← message TS\textsubscript{i}
4. Go to 1.
Causality - Unaware Communication System

- Causality module composed of send and receive filters.
- Factories for returning filters
- Filters can share common state such as site vector time stamp.
- Common state can be passed as parameters to factory and filter constructors

Causality-unaware Communication System

Send Filter → Causality Manager → Receive Filter
How to Test System

Must ask communication library to use direct (peer to peer) communication

How to ensure that message from PC 1 takes longer to reach PC 2 than PC 3?

Buffer₁

Buffer₂

Buffer₃

Done? 1 0 0
Lunch? 1 1 0
Nodes labeled in terms of their users

Actual delay maybe larger because of scheduling and network delays

```java
static void setDelaysAlice(Communicator communicator) {
    communicator.setMinimumDelayToPeer("cathy", 20000);
}
```
myTimeStamp.inc();
timestampedMessage.setTimeStamp(myTimeStamp);
messageProcessor.processMessage();

Incrementing the time stamp may change time stamps of previous unsent messages!
DEEP COPY

Timestamper → Delayer → Broadcaster

Ok 3 1 0  
Done? 2 0 0  
Lunch? 1 0 0  

3 1 0  Site time stamp

myTimeStamp.inc();
timestampedMessage.timeStamp = myTimeStamp.deepCopy();
mESSAGEProcessor.processMessage();
VectorTimeStamp deepCopy(VectorTimeStamp original) {
    return (VectorTimeStamp) Misc.deepCopy(original);
}

Uses Java’s ability to automatically serialize objects

Returns original if object is not serializable
Causality Architecture: Traceable Algorithm

Causality – Unaware Application

Send Filter -> Causality Manager

Receive Filter -> Causality Manager

Causality-unaware communication system
Peer Traceable Algorithm: Pre Communication Steps

- **Init**
- **VectorTimeStampCreated**

**Join Messages**

For each new user U, **UserAddedToVectorTimeStamp()**.
**Send Traceable Steps**

**Send Filter**

For each sent user message M

LocalCountIncrementedInSiteVectorTimeStamp

VectorTime Stam pedMessageSent through message processor

**Send Filter**

For each non-user message M

Pass unfiltered message to message processor
**Receive Traceable Steps**

**Receive Filter**

For each `VectorTimeStampedMessageReceived`

If `isConcurrent(M)` `ConcurrentVectorTimeStampedMessageDetected` ... return

`VectorTimeStampedMessageBuffered`

If `(isSuccessorNextBufferedMessage)`

`VectorTimeStampedMessageRemovedFromBuffer` and `VectorTimeStampedMessageDelivered`

**Receive Filter**

For each non-user message `M`

Pass unfiltered message to message processor

Handling Concurrent Messages?
**Immediately Delivering Concurrent Messages**

- When a message arrives see if its vector time stamp > the vector time stamp, put in the buffer and process buffer.
- Otherwise deliver immediately (optimistically assuming no conflict)
  - Update time stamp
    - Subsequent causal messages wrt to previous messages will not be processed
  - Do not update time stamp
    - Subsequent causal messages wrt to this message not processed
IMMEDIATELY DELIVERING CONCURRENT MESSAGES

- A tree of message paths exists
- Create vector time stamp and buffer for each leaf in the path
- When a message arrives see if its vector time stamp > one of the vector time stamps, put in the buffer for that vector time stamp
- Otherwise create a new vector time stamp and buffer (VectorTimeStampCopiedAndNewBufferCreated) and deliver the message after flagging concurrency
SUMMARY

- Assume reliable delivery
- Send logical timestamp with message
- If message received out of order, buffer it until preceding messages received
- In multi-party messages, vector timestamp
- Send and receive filters to make causality and application independent
- Bulk of work done by shared causality manager, which listens to join operations
- (Abstract) Factories to instantiate filters, which can be used to share objects between filters