Causal Message Delivery, Cooked Three Ways

Lindsey Kuper
UC Santa Cruz
IFP WG 2.16 (Language Design), January 2023
Let’s meet in Claremont to talk about gradual types!
Let's meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey – what’s it about?
Let's meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you're meeting with Lindsey — what's it about?

...what meeting with Lindsey? 😊
Let's meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you're meeting with Lindsey — what's it about?

...what meeting with Lindsey? 😔
Let's meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you're meeting with Lindsey — what's it about?

...what meeting with Lindsey? 😁
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey — what’s it about?

...what meeting with Lindsey?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
$\text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2)$

(total order of events on process $p$)
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey — what’s it about?

...what meeting with Lindsey?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$, 
$\text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2)$

$(\text{total order of events on process } p)$
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey — what’s it about?

Causal delivery:
For all messages \( m_1 \) and \( m_2 \) delivered at process \( p \),
\[
\text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2)
\]

(total order of events on process \( p \))
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey – what’s it about?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$, $send(m_1) \rightarrow send(m_2) \Rightarrow deliver(m_1) \rightarrow_p deliver(m_2)$

Receiver-side enforcement of causal delivery

(total order of events on process $p$)
Let’s meet in Claremont to talk about gradual types!
Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey — what’s it about?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$, $\text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2)$

(total order of events on process $p$)
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey — what’s it about?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
\[ \text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2) \]

(total order of events on process $p$)
Let’s meet in Claremont to talk about gradual types!

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
\[ \text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2) \]

(total order of events on process $p$)
Let’s meet in Claremont to talk about gradual types!

I heard you’re meeting with Lindsey — what’s it about?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
send($m_1$) $\rightarrow$ send($m_2$) $\Rightarrow$ deliver($m_1$) $\rightarrow_p$ deliver($m_2$)

(totally order of events on process $p$)
Let’s meet in Claremont to talk about gradual types!

I heard you’re meeting with Lindsey — what’s it about?

Could you join a meeting with me and Sam in Claremont?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$, $\text{send}(m_1) \rightarrow \text{send}(m_2)$ $\Rightarrow \text{deliver}(m_1) \rightarrow \text{deliver}(m_2)$

(send($m_1$) $\rightarrow$ send($m_2$) $\Rightarrow$ deliver($m_1$) $\rightarrow$ deliver($m_2$))

(total order of events on process $p$)

I heard you’re meeting with Lindsey — what’s it about?

ack("Let’s meet...")
Let’s meet in Claremont to talk about gradual types!

Could you join a meeting with me and Sam in Claremont?

I heard you’re meeting with Lindsey — what’s it about?

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
send($m_1$) → send($m_2$) ⇒ deliver($m_1$) →$_p$ deliver($m_2$)

( total order of events on process $p$ )
Let’s meet in Claremont to talk about gradual types!

I heard you’re meeting with Lindsey — what’s it about?

Could you join a meeting with me and Sam in Claremont?

Causal delivery:
For all messages \( m_1 \) and \( m_2 \) delivered at process \( p \),
\[
send(m_1) \rightarrow send(m_2) \Rightarrow deliver(m_1) \rightarrow_p deliver(m_2)
\]

(total order of events on process \( p \))

Sender-side enforcement of causal delivery

I heard you’re meeting with Lindsey — what’s it about?
A receiver-side protocol
[Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]

\[ \text{DELIV}_1 = [\emptyset, \emptyset, \emptyset] \]
A receiver-side protocol
[Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]
A receiver-side protocol

[Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]
A receiver-side protocol

[Raynal et al., 1991]
A receiver-side protocol [Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $\text{SENT}_m$ is deliverable at process $i$ if:

for all $k$, $\text{DELIV}_i[k] \geq \text{SENT}_m[k,i]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $\text{SENT}_m$ is deliverable at process $i$ if:

\[
\forall k, \text{DELIV}_i[k] \geq \text{SENT}_m[k,i]
\]
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $SENT_m$ is **deliverable** at process $i$ if:

for all $k$, $DELIV_i[k] \geq SENT_m[k,i]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $SENT_m$ is **deliverable** at process $i$ if:

for all $k$, $DELIV_i[k] \geq SENT_m[k,i]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $\text{SENT}_m$ is deliverable at process $i$ if:

for all $k$, $\text{DELIV}_i[k] \geq \text{SENT}_m[k,i]$
A receiver-side protocol  
[Raynal et al., 1991]

A message with metadata $\text{SENT}_m$ is deliverable at process $i$ if:

for all $k$, $\text{DELIV}_i[k] \geq \text{SENT}_m[k,i]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $SENT_m$ is deliverable at process $i$ if:

$$\forall k. \text{ DELIV}_i[k] \geq SENT_m[k,i]$$

increment $SENT_1[1,3]$ and merge with $m_1$'s metadata

increment $SENT_1[1,2]$ and merge with $m_2$'s metadata

increment $SENT_2[1,2]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $\text{SENT}_m$ is deliverable at process $i$ if:

\[
\forall k. \text{DELIV}_i[k] \geq \text{SENT}_m[k,i]
\]

and merge with $m_2$'s metadata

increment $\text{SENT}_2[1,2]$

increment $\text{SENT}_1[2,3]$

increment $\text{SENT}_1[1,3]$

increment $\text{SENT}_1[1,2]$
A receiver-side protocol

[Raynal et al., 1991]

A message with metadata $SENT_m$ is **deliverable** at process $i$ if:

$$\forall k. \text{DELIV}_i[k] \geq SENT_m[k,i]$$

A receiver-side protocol

[Raynal et al., 1991]
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $\text{SENT}_m$ is deliverable at process $i$ if:

for all $k$, $\text{DELIV}_i[k] \geq \text{SENT}_m[k,i]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $SENT_m$ is deliverable at process $i$ if:

for all $k$, $DELIV_i[k] \geq SENT_m[k,i]$
A receiver-side protocol
[Raynal et al., 1991]

A message with metadata $\textit{SENT}_m$ is deliverable at process $i$ if:

for all $k$, $\textit{DELIV}_i[k] \geq \textit{SENT}_m[k,i]$
A sender-side protocol
[Mattern and Fünfrocken, 1995]
A sender-side protocol
[Mattern and Fünfrocken, 1995]
A sender-side protocol
[Mattern and Fünfrocken, 1995]
A sender-side protocol  
[Mattern and Fünfrocken, 1995]

**output buffer algorithm:**
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

**output buffer algorithm:**
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

**Output buffer algorithm:**

1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

m1  m2

ack(m1)
A sender-side protocol
[Mattern and Fünfrocken, 1995]

**output buffer algorithm:**
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

output buffer

$\text{ack}(m_1)$
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1
A sender-side protocol
[Mattern and Fünfrocken, 1995]

**output buffer algorithm:**
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

**Causal delivery:**
For all messages $m_1$ and $m_2$ delivered at process $p$,
$\text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow \text{deliver}(m_2)$

(total order of events on process $p$)

output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$, \(\text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow \text{deliver}(m_2)\)

(output buffer)
output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
send($m_1$) $\rightarrow$ send($m_2$) $\Rightarrow$ deliver($m_1$) $\rightarrow$ deliver($m_2$)

A sender-side protocol
[Mattern and Fünfrocken, 1995]
approximates
synchronous
communication

Output buffer algorithm:
1. wait for a message to appear
2. dequeue & transmit next message
3. wait for acknowledgment
4. go to step 1

Causal delivery:
For all messages $m_1$ and $m_2$ delivered at process $p$,
\[ \text{send}(m_1) \rightarrow \text{send}(m_2) \Rightarrow \text{deliver}(m_1) \rightarrow_p \text{deliver}(m_2) \]

(send $m_1$) → (send $m_2$) ⇒ (deliver $m_1$) $\rightarrow_p$ (deliver $m_2$)

(total order of events on process $p$)
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
Idea: the “can you keep a secret?” protocol
some design considerations for the “can you keep a secret?” protocol
some design considerations for the “can you keep a secret?” protocol

- Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
some design considerations for the “can you keep a secret?” protocol

• Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  • Right away. In fact, for FIFO delivery, the sender *shouldn’t* send now-you-can-tell until the eager send is ack’d!
some design considerations for the “can you keep a secret?” protocol

- Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  - Right away. In fact, for FIFO delivery, the sender *shouldn’t* send now-you-can-tell until the eager send is ack’d!

- Can you do an eager send as a result of an eager send?
some design considerations for the “can you keep a secret?” protocol

- Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  - Right away. In fact, for FIFO delivery, the sender shouldn’t send now-you-can-tell until the eager send is ack’d!

- Can you do an eager send as a result of an eager send?
  - No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!
some design considerations for the “can you keep a secret?” protocol

- Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  - Right away. In fact, for FIFO delivery, the sender *shouldn’t* send now-you-can-tell until the eager send is ack’d!

- Can you do an eager send as a result of an eager send?
  - No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!

- When a process wants to send a message as a result of an eager send it previously received, can the new message go into the output buffer while waiting for the now-you-can-tell message, or should it not even be buffered yet?
**some design considerations for the “can you keep a secret?” protocol**

- Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  - Right away. In fact, for FIFO delivery, the sender *shouldn’t* send now-you-can-tell until the eager send is ack’d!

- Can you do an eager send as a result of an eager send?
  - No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!

- When a process wants to send a message as a result of an eager send it previously received, can the new message go into the output buffer while waiting for the now-you-can-tell message, or should it not even be buffered yet?
  - I don’t know!
some design considerations for the “can you keep a secret?” protocol

• Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  • Right away. In fact, for FIFO delivery, the sender shouldn’t send now-you-can-tell until the eager send is ack’d!

• Can you do an eager send as a result of an eager send?
  • No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!

• When a process wants to send a message as a result of an eager send it previously received, can the new message go into the output buffer while waiting for the now-you-can-tell message, or should it not even be buffered yet?
  • I don’t know!

• (my excuse for giving a distributed systems talk at WGLD) How about some kind of information flow analysis?
some design considerations for the “can you keep a secret?” protocol

• Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  • Right away. In fact, for FIFO delivery, the sender shouldn’t send now-you-can-tell until the eager send is ack’d!

• Can you do an eager send as a result of an eager send?
  • No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!

• When a process wants to send a message as a result of an eager send it previously received, can the new message go into the output buffer while waiting for the now-you-can-tell message, or should it not even be buffered yet?
  • I don’t know!

• (my excuse for giving a distributed systems talk at WGLD) How about some kind of information flow analysis?
  • if an analysis shows $m_2$ is in fact not related to $m_1$, I can send $m_2$ without regard to whether I’ve heard back about $m_1$
some design considerations for the “can you keep a secret?” protocol

• Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  • Right away. In fact, for FIFO delivery, the sender shouldn’t send now-you-can-tell until the eager send is ack’d!

• Can you do an eager send as a result of an eager send?
  • No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!

• When a process wants to send a message as a result of an eager send it previously received, can the new message go into the output buffer while waiting for the now-you-can-tell message, or should it not even be buffered yet?
  • I don’t know!

• *(my excuse for giving a distributed systems talk at WGLD)* How about some kind of information flow analysis?
  • if an analysis shows $m_2$ is in fact not related to $m_1$, I can send $m_2$ without regard to whether I’ve heard back about $m_1$
  • happens-before is too coarse; we use it because it’s an overapproximation of actual causality that’s easy to compute
some design considerations for the “can you keep a secret?” protocol

• Should a recipient acknowledge eager sends right away, or wait until it gets the now-you-can-tell message?
  • Right away. In fact, for FIFO delivery, the sender shouldn’t send now-you-can-tell until the eager send is ack’d!

• Can you do an eager send as a result of an eager send?
  • No. E.g., if Ron can eagerly send $m_3$, then it can overtake $m_1$ on the way to Sam, which was the original problem!

• When a process wants to send a message as a result of an eager send it previously received, can the new message go into
  the output buffer while waiting for the now-you-can-tell message, or should it not even be buffered yet?
  • I don’t know!

• (my excuse for giving a distributed systems talk at WGLD) How about some kind of information flow analysis?
  • if an analysis shows $m_2$ is in fact not related to $m_1$, I can send $m_2$ without regard to whether I’ve heard back about $m_1$
  • happens-before is too coarse; we use it because it’s an overapproximation of actual causality that’s easy to compute
  • it’s 2023 and we’re PL people, dang it; let’s use language-level techniques that weren’t available when these protocols
    were originally designed in the ’90s