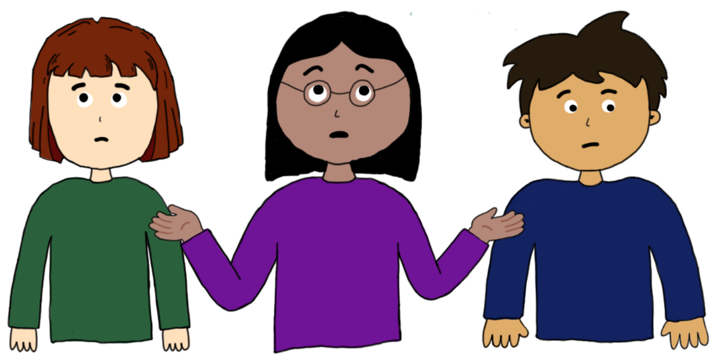
Carol's Causal Conundrum



by Ayush Manocha
and Lindsey Kuper

Welcome!

This zine is an introduction to *causally ordered message delivery* (or just *causal delivery* for short). Causal delivery is a widely used building block of distributed systems made up of communicating components (like email or peer-to-peer chat!). We'll talk about what exactly it is, what problem it solves, and some ways to implement it.

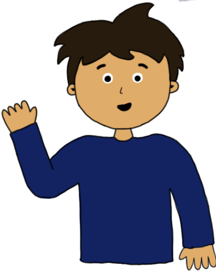


Table of Contents

Why is Carol confused? 1

Processes and messages 3

The happens-before relation 6

Causal delivery 8

A solution to Carol's conundrum: receiver-side enforcement ... 12

Another solution: sender-side enforcement 16

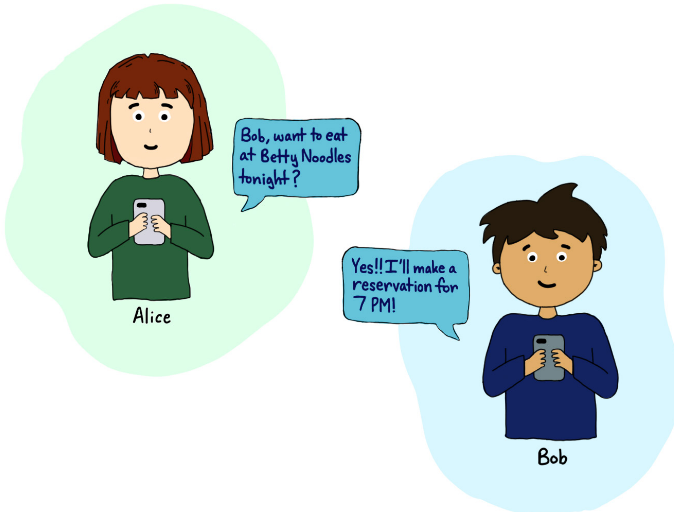
Can you keep a secret? 19

Takeaways 22

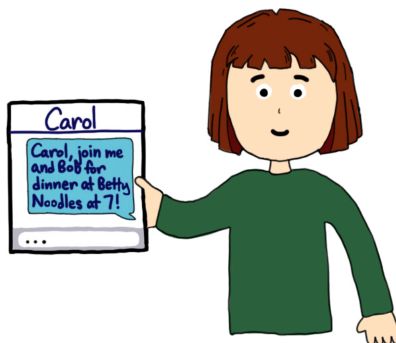
References 24

Why is Carol confused?

One day, Alice decides she wants to have dinner at her favorite restaurant, Betty Noodles. She messages her friend Bob on SlugMessage asking him if he's up for noodles tonight. Bob is always up for noodles, so he excitedly replies back, saying that he'll make a reservation.

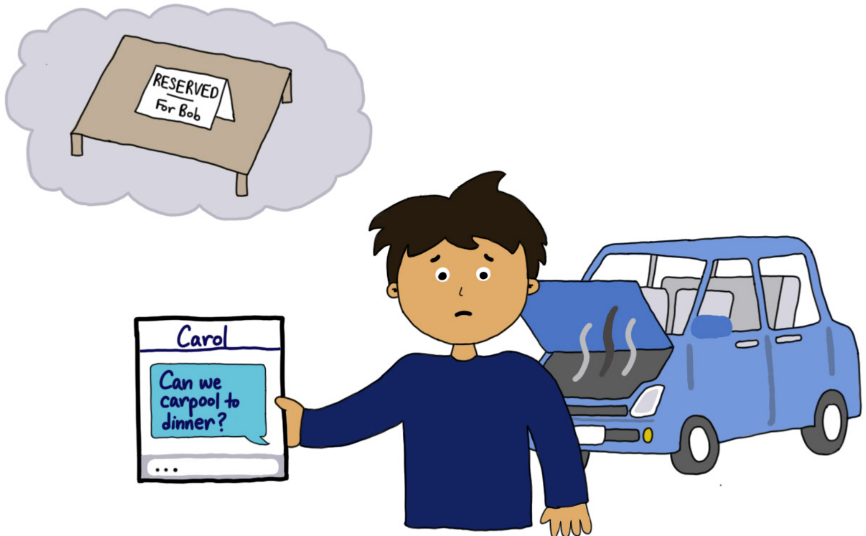


Alice then invites their mutual friend Carol to the dinner through SlugMessage as well. The thing about SlugMessage is that sometimes messages take quite a while to get to their destination...

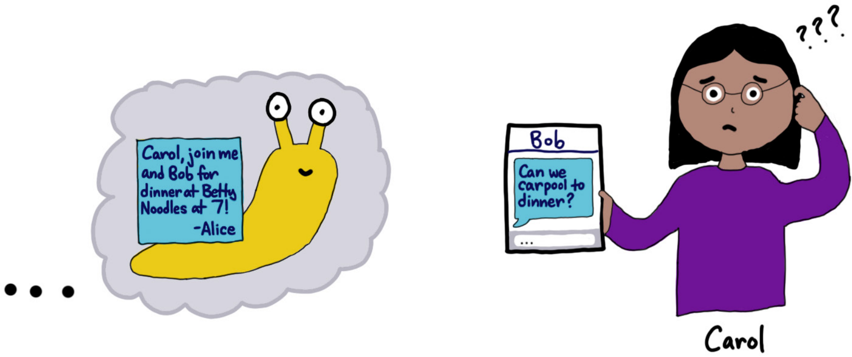


After that, Alice lets Bob know that Carol will now be joining and that the reservation will need to be for the three of them.

Bob then calls Betty Noodles and makes the reservation for 7 pm. He decides to run a couple errands before the dinner, so he hops in his car, only to find that his car won't start! After some poking around, he realizes he likely isn't going to fix it in time. So, he messages Carol asking if they could carpool to dinner.



Unfortunately, Alice's invite message to Carol happens to be moving quite slowly through SlugMessage's notoriously sluggish network. So, when Carol sees Bob's message asking to carpool, she hasn't received Alice's invitation yet, leaving her quite confused by his message.



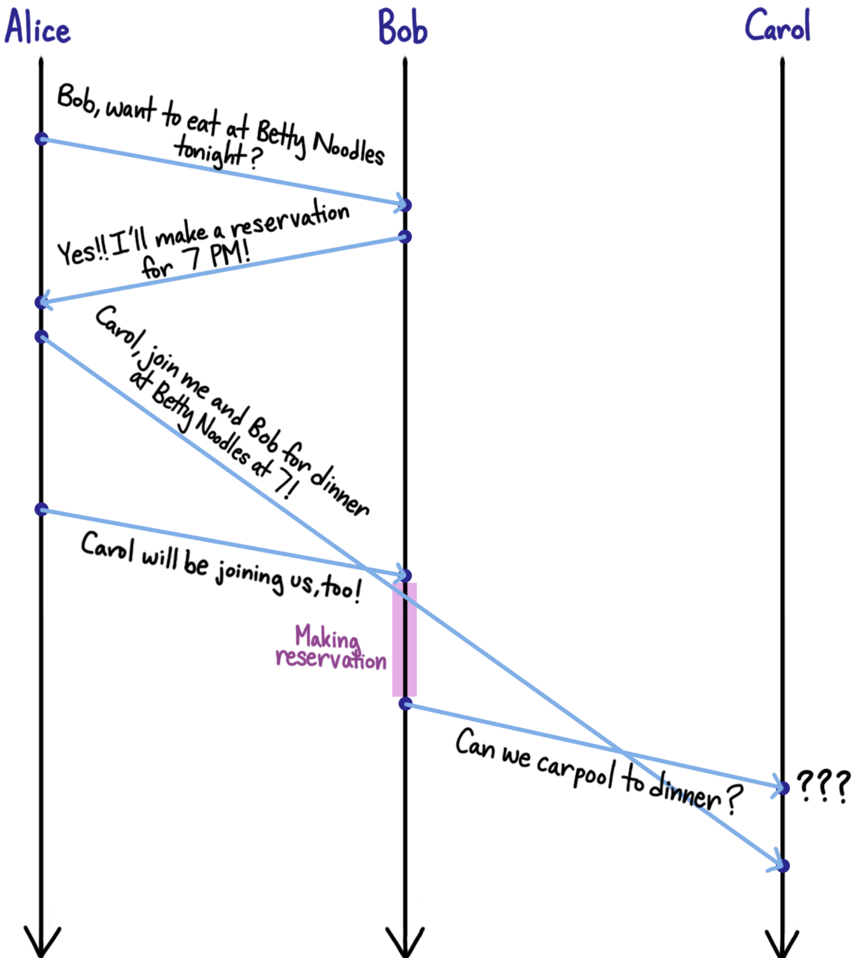
What exactly went wrong here, and how can it be fixed?

Processes and messages

We can think of Alice, Bob, and Carol as independent participants that communicate with each other by sending and receiving messages. In distributed computing, these participants are traditionally known as *processes*.

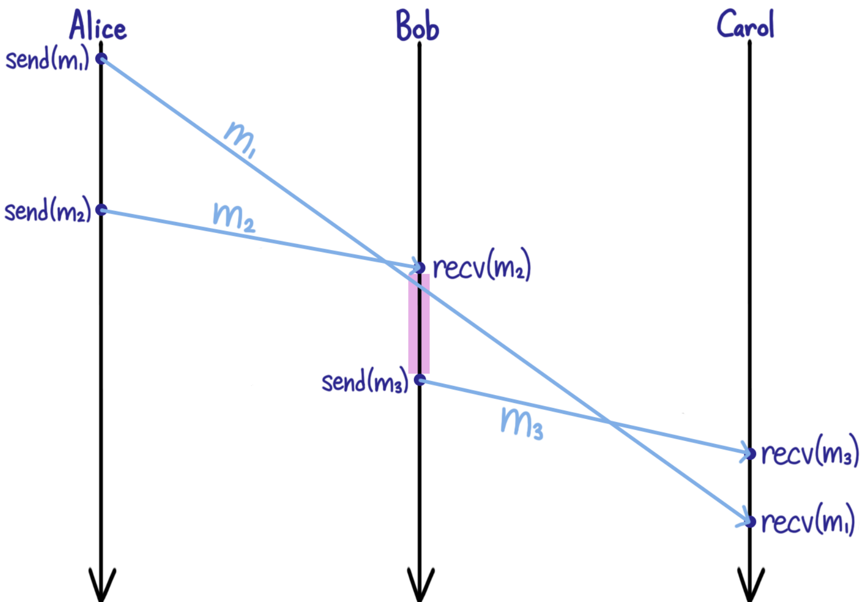
Processes don't share memory, so the only way any process can find out about what's happening on some other process is by receiving a message from it. (So, for example, if Alice and Bob exchange messages among themselves, Carol won't be aware of that unless Alice or Bob tell her.)

Let's visualize an execution involving our three processes in the below diagram, where the vertical axis represents time. Each process is represented by its own vertical line. The arrows between processes represent the messages being sent, and the dots on each process represent events occurring on that process. Here, the events we care about are the sending and reception of messages.



Alice, Bob, and Carol are communicating over an *asynchronous* network, which means that there's no limit on how long it takes for messages to get from place to place. So a message might be slow, like Alice's message to Carol was, leading Carol to see Bob's message first even though Alice's message was sent earlier!

Asynchrony in distributed systems can lead to confusion and bugs. To help tame asynchrony, we turn to *causality* — the principle that an effect cannot happen before its cause! To understand where causality comes into play, let's label the events and messages in our diagram. We will also omit the first two messages between Alice and Bob and zoom into the latter half of the execution, starting from Alice's invite to Carol, as that is where the issues arise.



We can see that Bob sent a message (m_3 = "Can we carpool to dinner?") to Carol *because* Alice previously sent a message (m_2 = "Carol will be joining us too!") to Bob. And Alice did *that* because she wanted Carol to come to dinner, which is why she sent m_1 = "Carol, join me..."! So we can think of the send events for messages m_1 and m_2 as being part of the causal history of the send event of m_3. *In other words, if it hadn't been for* m_1 and m_2, *then* m_3 *never would have happened.*

But when m_3 shows up at Carol, she doesn't know about that causal history — *in particular, she doesn't know about* m_1, *even though it was sent to her! That's why she's confused.*

In fact, the pattern of messages in the execution shown on the previous page is the canonical example of a violation of causal message delivery!

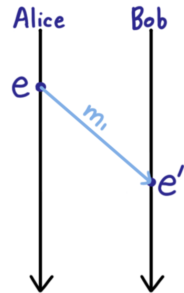
The happens-before relation

Fortunately, distributed systems folks have developed some useful tools for talking about *what caused what* in a distributed system, and helping Carol be less confused.

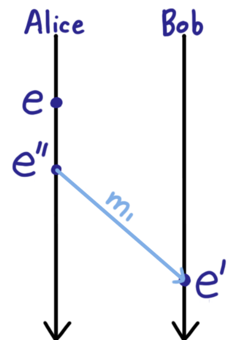
The _happens-before_ relation [Lamport 1978] lets us reason about the ordering of events in an execution of a distributed system. Consider a particular pair of events in an execution, which we'll call **e** and **e'** (for lack of better names!). We can say that event **e** *happens before* event **e'** in the execution, written as **e** → **e'**, if one of the following is true:

1. **e** occurs before **e'** **on the same process.** (In Lamport's model, events on a single process always happen one after another, never at the same time.)

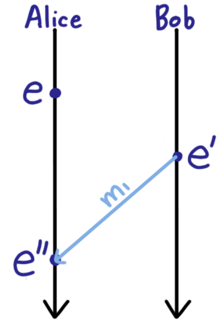
2. **e** is a **send** event and **e'** is the corresponding **receive** event (on some other process).



And there's one more important thing about the happens-before relation: we can chain together happens-before relationships. So, if we know that **e** happens before some other event **e''** (**e** → **e''**), and **e''** happens before **e'** (**e''** → **e'**), then we can combine those two facts to say that **e** happens before **e'**. This last part of the definition is what makes happens-before a _transitive_ relation!



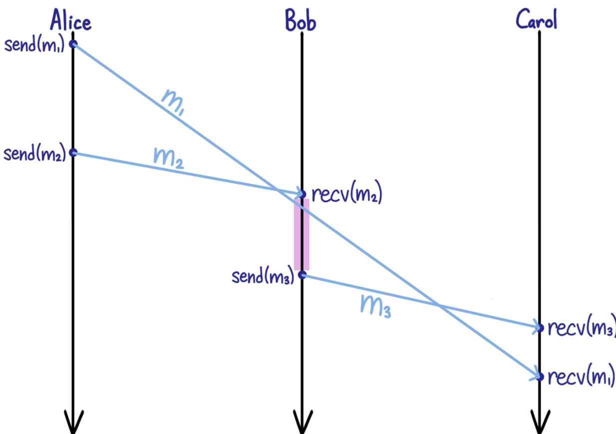
Some pairs of events in an execution may not have an order! In that case, we say that the events are *concurrent*. For example, in the diagram to the right, events **e** and **e'** are concurrent, and we can say neither that **e** → **e'** nor that **e'** → **e**.



Once we've figured out which events are ordered by the happens-before relation, we know that if **e** → **e'**, then **e** might have caused **e'**. The relationship is just *potential* causality, since we only know that **e** might have caused **e'**, not that it *did* cause **e'**. But distributed systems folks usually say "causality" as an abbreviation for "potential causality".

Causal delivery

Let's come back to Carol's confusion with the happens-before relation in hand. We can see that in this execution, the send event for Alice's message m_1 to Carol happens before the send event for Alice's message m_2 to Bob. Those are both events on the same process (Alice), so they're related: **send**(m_1) → **send**(m_2).



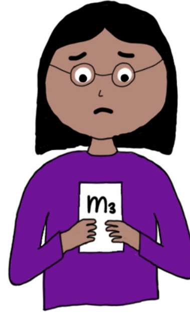
Then, the send event of Alice's message m_2 to Bob happens before the reception of that same message on Bob's process. This is a send-receive pair of events, so they're related by happens-before: **send**(m_2) → **recv**(m_2). And after receiving m_2, Bob sends m_3; since those are events on the same process, they're related by happens-before too: **recv**(m_2) → **send**(m_3).

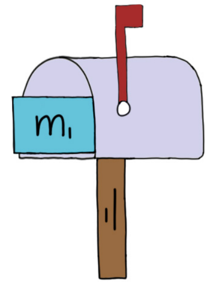
We have a chain of happens-before relationships: **send**(m_1) → **send**(m_2) → **recv**(m_2) → **send**(m_3). So, because the happens-before relation is transitive, **send**(m_1) → **send**(m_3). Unfortunately, that's not the order that Carol actually saw m_1 and m_3 in – she saw them in the opposite order, which is why she's so confused!

To precisely describe what went wrong, we need one more piece of terminology. Let's say that there's a mail clerk sitting on Carol's process who intercepts all incoming messages, stores them temporarily in the mail room, and then delivers them to Carol.

When Carol's mail clerk *delivers* a message to Carol for her to read, we say that a *deliver* event occurs at Carol's process. The mail clerk may choose to deliver the messages to Carol in a different order than the order in which they are received! (More about that later.) So, message delivery should be its own kind of event, separate from a receive event. We use the notation **deliver**$_{\text{Carol}}$(m) for that delivery event, and we say "m is *delivered at* Carol." Then, because Carol read m_3 before m_1, we have **deliver**$_{\text{Carol}}$(m_3) → **deliver**$_{\text{Carol}}$(m_1).

We can now see that the source of the confusion was that **deliver**$_{Carol}$(m_3) → **deliver**$_{Carol}$(m_1), despite **send**(m_1) → **send**(m_3). This is a violation of *causal delivery*.

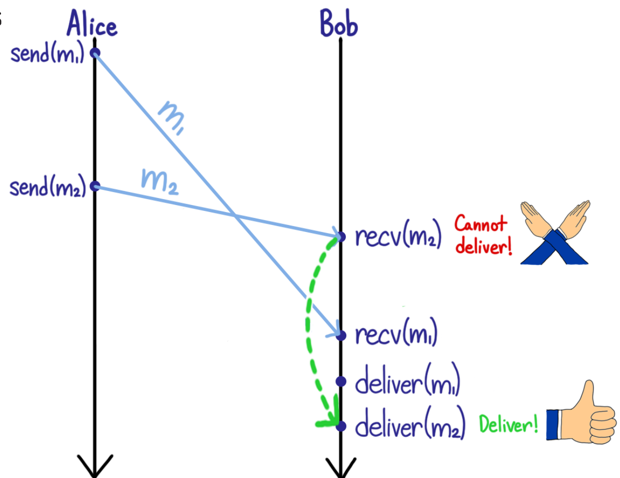




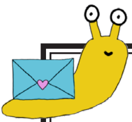
Causal delivery is a desirable property that states that the above execution **cannot happen**. More specifically, causal delivery says that for all messages m_1 and m_2 delivered at the same process p, then no matter who sent them,

if **send**(m_1) → **send**(m_2), then **deliver**$_p$(m_1) → **deliver**$_p$(m_2).

It's okay if messages are *received* at a process in an order that's different from the causal order, as long as they are *delivered* at that process in an order that's consistent with the causal order. This is important, because a process can't control when it receives messages! Since we have an asynchronous network, messages could show up quickly or slowly. Causal delivery mechanisms (like our imaginary mail clerk) are there to help tame that asynchrony.

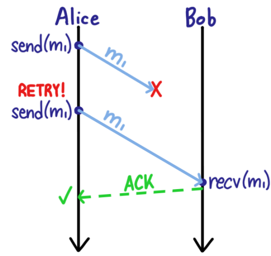


We're making an important simplifying assumption here: the assumption of _reliable delivery_. This means that we assume every message will **eventually** make it to its intended recipient at some point (though not in any particular guaranteed order). In other words, there will never be a message that is entirely lost and never reaches its intended recipient. This simplifies the problem: now, instead of worrying about whether messages will arrive at all, we just have to worry about the *order* in which they arrive.



Reliable delivery

Under the hood, reliable delivery could be accomplished by having the recipient of a message inform the sender that the message is received by sending an _acknowledgment_ message. If no acknowledgment is received,



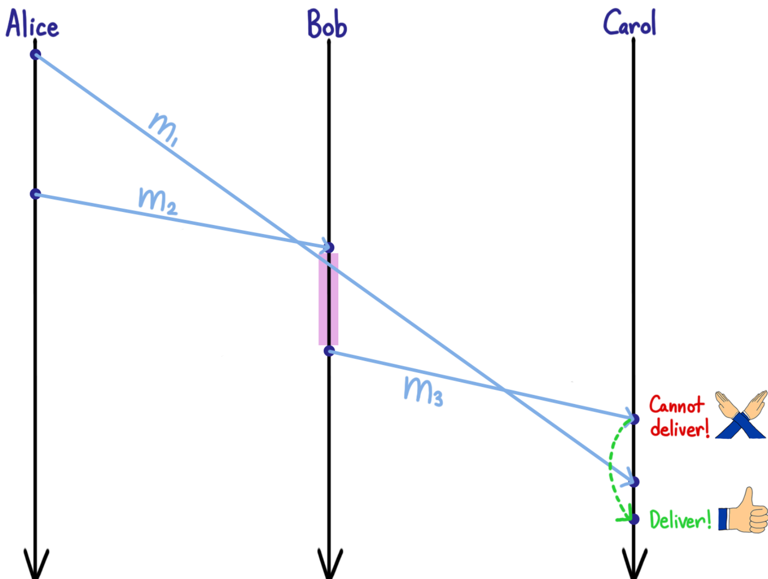
the sender would re-send the message after some time. The details are out of scope for this zine, but this is more or less how real protocols that guarantee reliable delivery, such as TCP and RUDP, would go about it. (By the way, TCP also happens to enforce a helpful ordering property, known as _ordered delivery_ or _FIFO delivery_, which ensures that messages from the same sender are delivered in the order they are sent. But that property by itself still wouldn't be enough to solve Carol's conundrum! For that, only causal delivery will do the job.)

A solution to Carol's conundrum: receiver-side enforcement

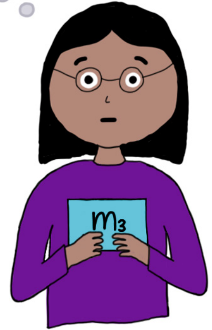
So, we've established the property we want to enforce: causal delivery. Here's the definition again:

> *Causal delivery:*
>
> For all messages m_1 and m_2 delivered at the same process p, if **send**(m_1) → **send**(m_2), then **deliver**$_p$(m_1) → **deliver**$_p$(m_2).

How do we actually ensure that this property holds? One classic and widely-used approach is what we'll call *receiver-side enforcement* of causal delivery. This means that the recipient of messages must ensure that delivery happens in an order that is consistent with their happens-before order.

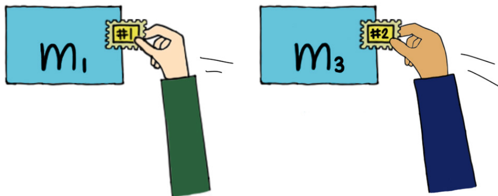


With receiver-side enforcement of causal delivery, Carol can still receive m_3 before receiving m_1, but then she would *delay* the delivery of m_3 until after m_1 is received and delivered at her process. Nothing about the timing of the sends and receives of these messages would change, but Carol is able to ensure that causal order isn't violated.



How would Carol know what order the messages were sent in, though? She doesn't have knowledge of every event and message that Alice and Bob are involved in!

For such a receiver-side protocol to work, this knowledge needs to be passed to the recipient of messages *through the messages themselves*. We can do this by attaching some form of extra *metadata* to each message that provides a clear mechanism for the recipient to determine the happens-before order of messages.

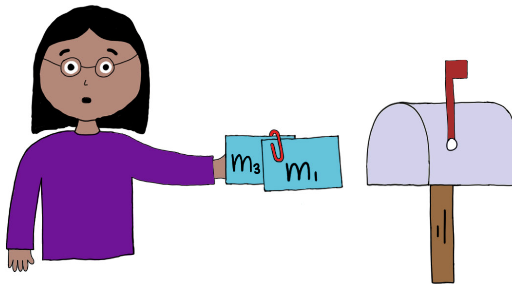


What should this metadata look like, though? Many answers to this question have been proposed over the years! Let's consider a couple of different approaches.

An early solution to this problem, described by Birman and Joseph [1987], is to have each message carry with it a copy of all *other* messages that the sender has seen so far!

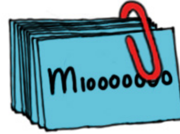
Birman and Joseph's "message piggybacking" mechanism is specific to a broadcast setting, in which all messages go to all participants. That's different from our scenario, but we can still imagine how something like it might work for us.

In our example, Alice's message m_2 to Bob would include a copy of her message m_1 to Carol, even though Bob isn't the intended destination of the message, and Bob's message m_3 to Carol would, in turn, include copies of both m_1 and m_2. Then, when Carol initially received m_3, she would first look through the attached copies and find m_1, which was intended for her and had not yet arrived. She could then make sure m_1 is delivered before delivering m_3. When the original m_1 showed up eventually, she could just discard it.



This would solve Carol's problem — but in a very inefficient way! All those messages take up a lot of space. Each process would have to keep around a copy of every message it had ever received or sent, since those messages precede any message the process might send in the future.

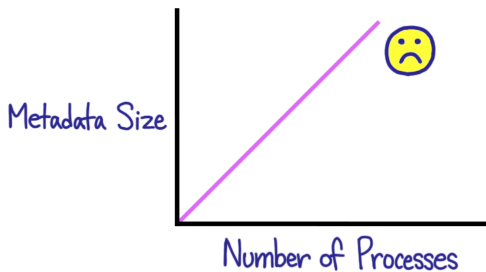
And, as Alice, Bob, and Carol continue communicating, the amount of causal information attached to each of their messages would accumulate over time and would periodically have to be cleaned up using a *garbage collection* algorithm.



Fortunately, there is an alternative to just attaching all preceding messages to every message! Instead, the sender could just attach some *much smaller* metadata to each message that summarizes *how many* messages causally precede it. Then, recipients can check the metadata and see if there were any preceding messages for them that they haven't gotten yet. If so, then the recipient knows those messages are still in transit, so they put the current message into a queue for later delivery and wait for the in-transit messages to show up.

Several algorithms for this "count the preceding messages" approach were proposed in the late 1980s and early 1990s [Schiper et al. 1989, Raynal et al. 1991, Birman et al. 1991]. These receiver-side algorithms differ in the details of their implementations and in the assumptions they make, but all of them use some kind of *logical clock* mechanism to keep track of how many preceding messages have been sent.

Further discussion of logical clocks is out of scope for this zine, but logical-clock-based protocols are a huge efficiency improvement over the piggybacking approach! However, even with logical clocks, the size of the metadata can still get prohibitively large. The size of the metadata would grow in proportion with the number of processes participating in the system. It wouldn't be so bad if it were just Alice, Bob, and Carol, but it still wouldn't scale well to a system with, say, hundreds of thousands of participants.

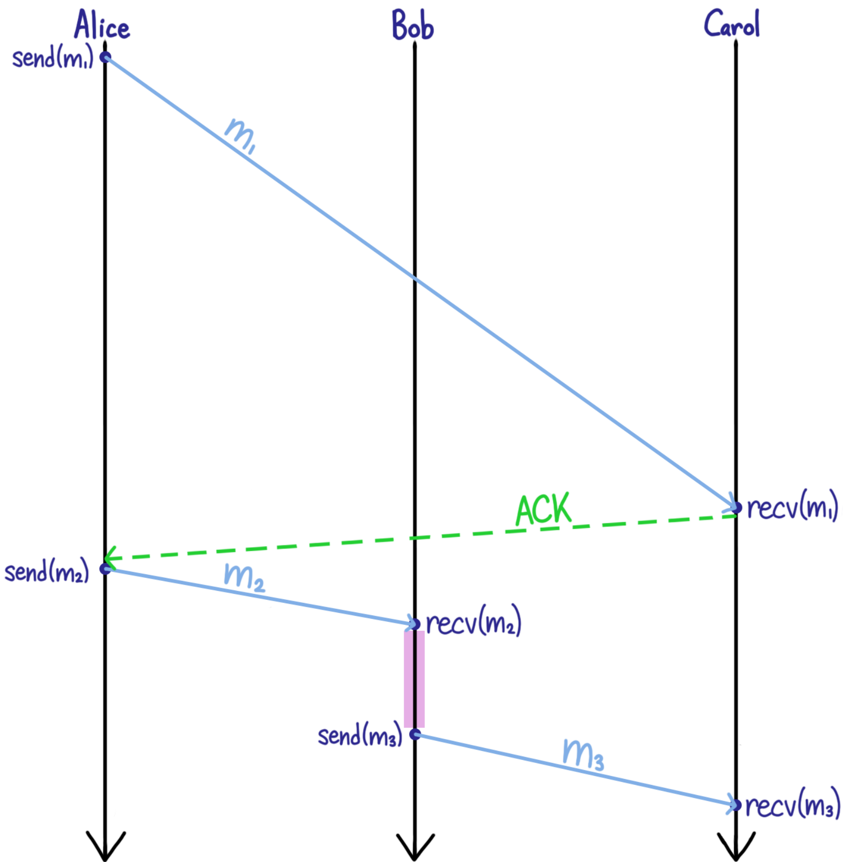


Another solution: sender-side enforcement

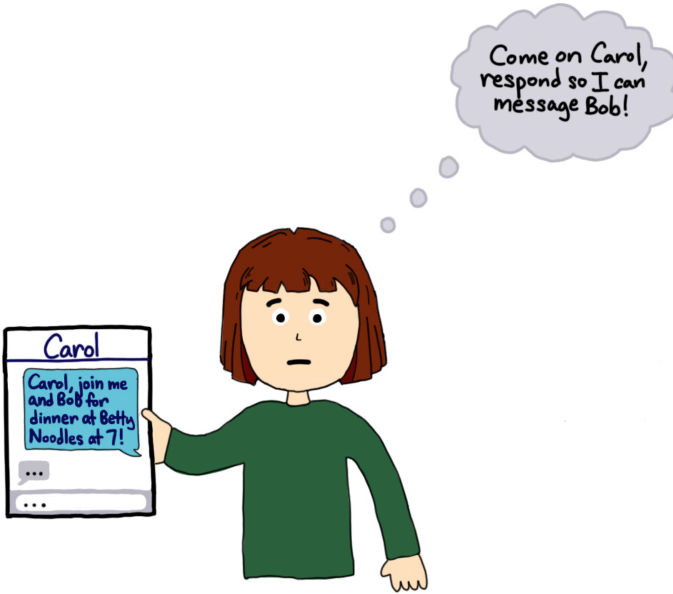
Instead of requiring all of this metadata to help Carol sort through the messages, an alternative protocol, proposed by Mattern and Fünfrocken [1995], would alter the timing of when Alice and Bob's messages are sent in such a way that guarantees Carol will read them in the correct order. So, Carol's mail clerk is no longer responsible for sorting through the messages and ordering them — that is taken care of for her by Alice and Bob, the senders of the messages!

This can be done by preventing Alice from messaging Bob (m_2) at all until she receives an acknowledgement message (ACK) from Carol that she is aware of their dinner plans.

So, because Carol is guaranteed to have read Alice's message m_1 before Alice sends m_2 to Bob and, thereafter, Bob sends m_3 to Carol, Carol is guaranteed to have read Alice's message m_1 before Bob's message m_3.



This protocol doesn't require *any* metadata in order to provide guarantees of causal delivery. But now there's a new wrinkle. Particularly, Alice is now delayed in sending m_2 to Bob, thereby delaying his call to Betty Noodles to make a reservation!



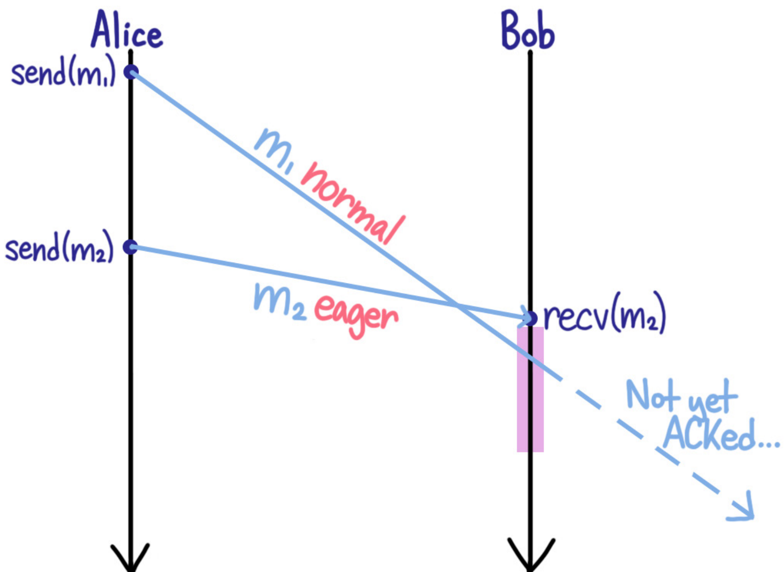
By imposing these delays, the communication between processes now more closely models that of a system with *synchronous* communication. Synchronous communication is often simpler to reason about than asynchronous communication, but that simplicity comes at a cost. In the Mattern and Fünfrocken sender-side protocol, Alice's send of m_2 to *Bob* is delayed, even though m_2 never posed any risk of violating causal delivery! It's really only m_3 getting delivered too early that we should worry about.

Is there a way to avoid the metadata overhead of the receiver-side approach, but also avoid the needless delays of the sender-side approach?

Can you keep a secret?

The recently proposed "Can You Keep a Secret", or Cykas, protocol [Tong et al. 2026], is an update to the traditional Mattern-Fünfrocken sender-side protocol.

Instead of restricting Alice from sending a message to Bob until she receives acknowledgement back from Carol, the Cykas protocol lets her send that message to Bob. We will call this message, which was sent while still awaiting acknowledgement from Carol, an _eagerly sent_ message.



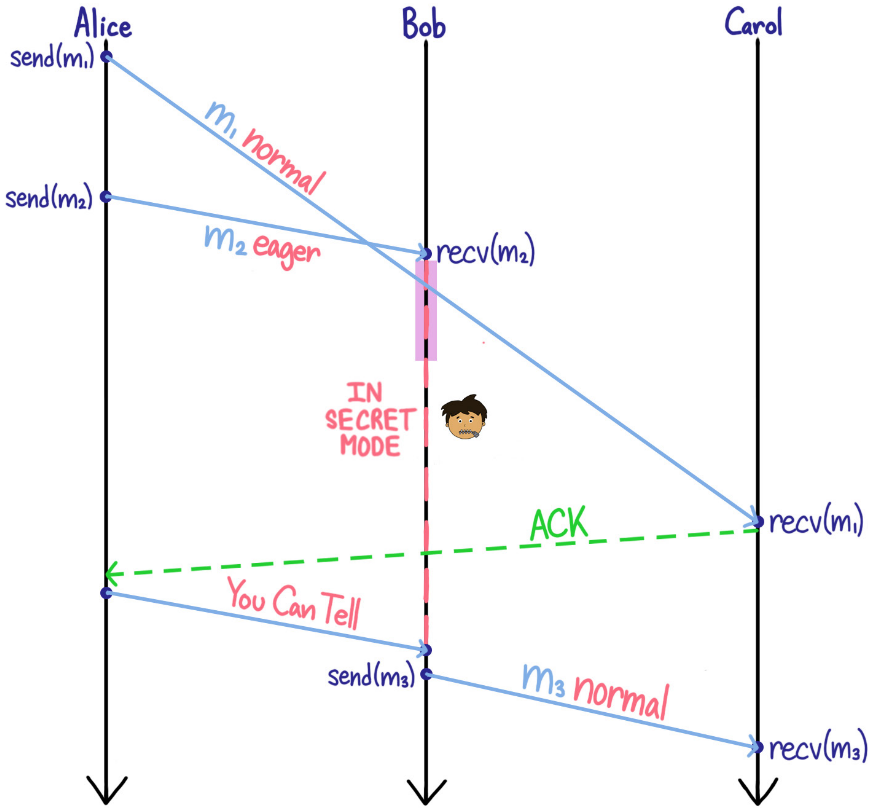
This lets Bob get started with making the reservation right after receiving Alice's message, without being unnecessarily impacted by Alice's slow message to Carol. This helps the whole execution run more quickly, especially if making the reservation takes a while!

We still have yet to prevent the violation of causal delivery that could happen once Bob finishes booking the table and messages Carol asking her to carpool, though.

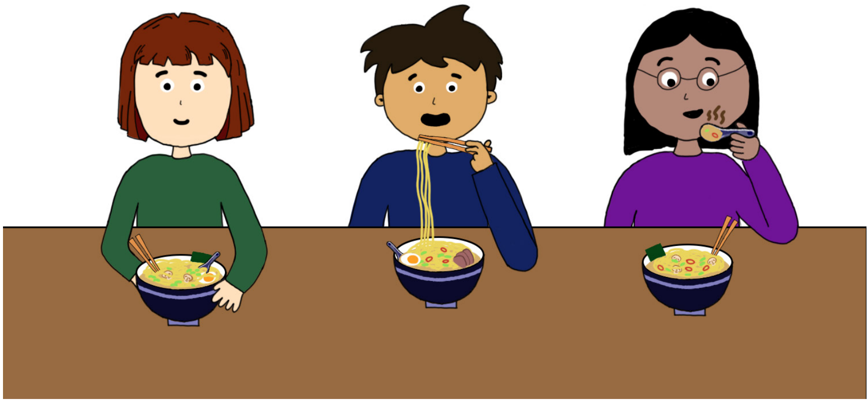
To do this, the Cykas protocol treats the eagerly sent message as a *secret* Alice is sharing with Bob! When he receives the eagerly sent m_2, he goes into *secret mode* and cannot take any externally observable actions, such as sending a message to Carol.

As with the Mattern-Fünfrocken sender-side protocol, Carol must acknowledge Alice's message. Once Alice receives this acknowledgement, she will immediately send Bob a *You Can Tell* message, which brings him out of secret mode back to *normal mode*.

Bob can now send his message m_3 asking Carol to carpool as a *normal* message, with a guarantee that Carol has already seen Alice's dinner invite in message m_1.



This way, Alice isn't delayed from sending other messages while waiting for Carol to read her message, so Bob can get started on setting up the reservation more quickly. That's good news, because the faster Bob can get done, the faster everyone can eat noodles!



Takeaways

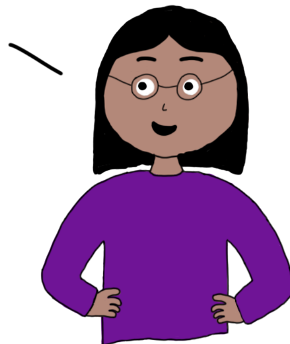
In this zine we've discussed causally ordered message delivery, a widely used building block of distributed systems, and touched on a few ways to enforce it, both on the message sender's side and on the recipient's side.

The approaches we've discussed all have pros and cons. The **sender-side approach** doesn't require attaching any metadata to messages, but it delays some messages needlessly. The **receiver-side approach** doesn't suffer from those needless delays, but it does require causal metadata on each message, and even "lightweight" approaches use metadata proportional to the number of processes in the system. Finally, the **Cykas protocol** is a new sender-side protocol that attempts to avoid the needless delay issue, but it requires more messages than the traditional sender-side approach.

Applications that rely on causal delivery include op-based CRDTs [Shapiro et al. 2011] and certain distributed snapshot protocols [Acharya and Badrinath 1993], among others!

We haven't gone into detail about how any of the mentioned enforcement approaches work or about the assumptions they make, but we recommend checking out the papers we've cited for all of the details.

Thanks for reading!



References

[Acharya and Badrinath 1993] Arup Acharya and B. R. Badrinath. 1993. Recording distributed snapshots based on casual order of message delivery. *Inf. Process. Lett.* 44, 6 (Dec. 28, 1992), 317–321. https://doi.org/10.1016/0020-0190(92)90107-7

[Birman and Joseph 1987] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.* 5, 1 (Feb. 1987), 47–76. https://doi.org/10.1145/7351.7478

[Birman et al. 1991] Kenneth Birman, André Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* 9, 3 (Aug. 1991), 272–314. https://doi.org/10.1145/128738.128742

[Lamport 1978] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

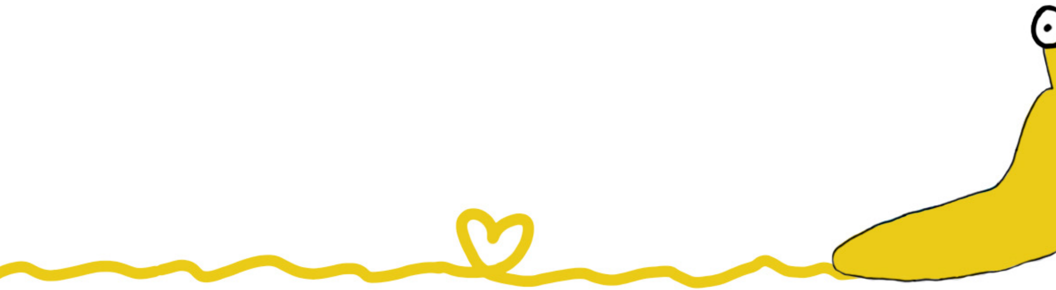
[Mattern and Fünfrocken 1995] Friedemann Mattern and Stefan Fünfrocken. 1995. A non-blocking lightweight implementation of causal order message delivery. In: Birman, K.P., Mattern, F., Schiper, A. (eds) *Theory and Practice in Distributed Systems. Lecture Notes in Computer Science*, vol 938. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-60042-6_14

[Raynal et al. 1991] Michel Raynal, André Schiper, and Sam Toueg. 1991. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, Volume 39, Issue 6, 1991, pages 343-350, ISSN 0020-0190. https://doi.org/10.1016/0020-0190(91)90008-6

[Schiper et al. 1989] André Schiper, Jorge Eggli, and Alain Sandoz. 1989. A new algorithm to implement causal ordering. In: Bermond, JC., Raynal, M. (eds) Distributed Algorithms. WDAG 1989. Lecture Notes in Computer Science, vol 392. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-51687-5_45

[Shapiro et al. 2011] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011 Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds) Stabilization, Safety, and Security of Distributed Systems. SSS 2011. Lecture Notes in Computer Science, vol 6976. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-24550-3_29

[Tong et al. 2026] Yan Tong, Nathan Liittschwager, and Lindsey Kuper. 2026. Can you keep a secret? A new protocol for sender-side enforcement of causal message delivery. arXiv:2603.14690 [cs.DC] https://doi.org/10.48550/arXiv.2603.14690



Download your own printable copy of this zine and others!

https://decomposition.al/zines/

A special thanks to Julia Evans and Marie Claire LeBlanc Flanagan for their feedback on a draft of this zine!