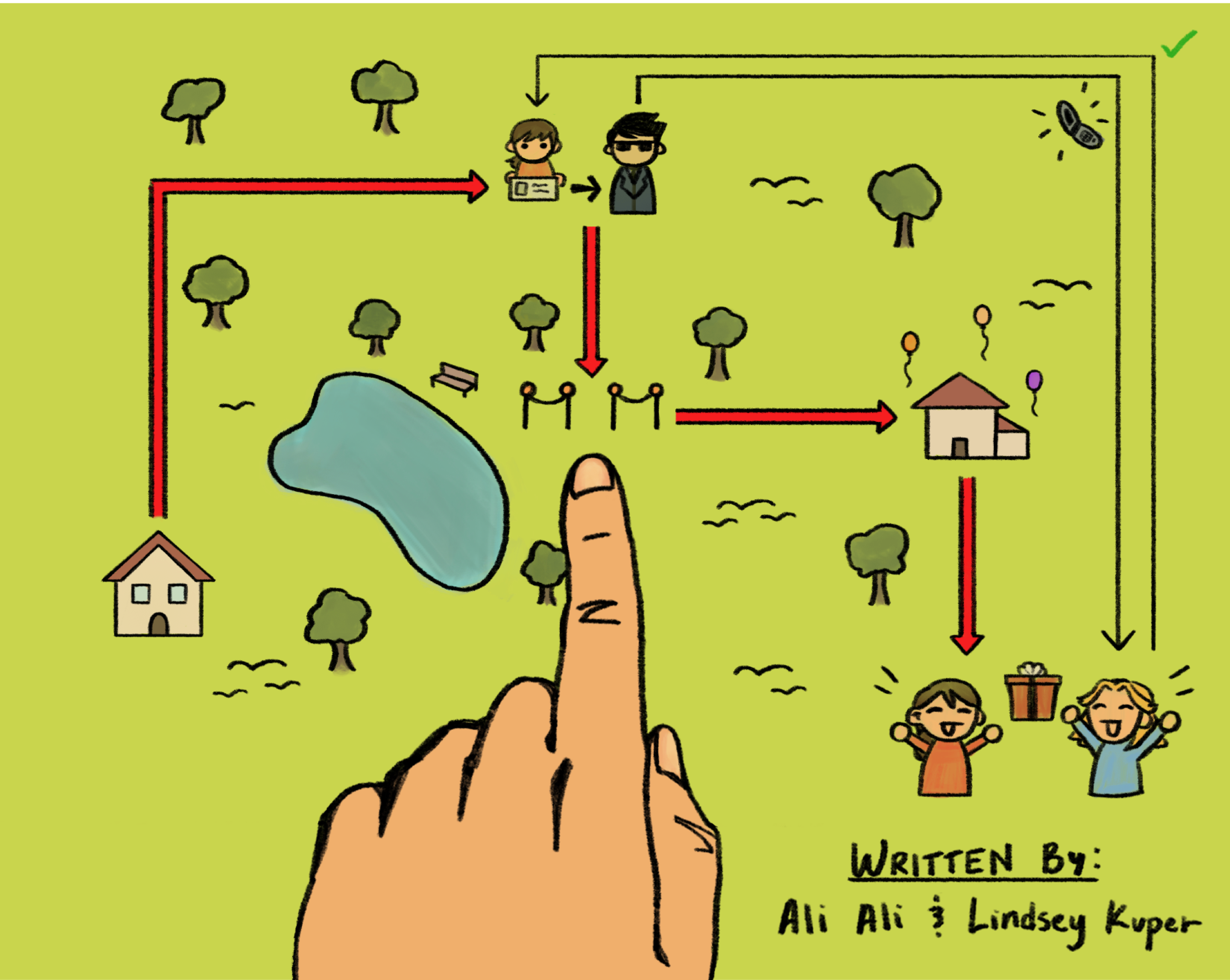
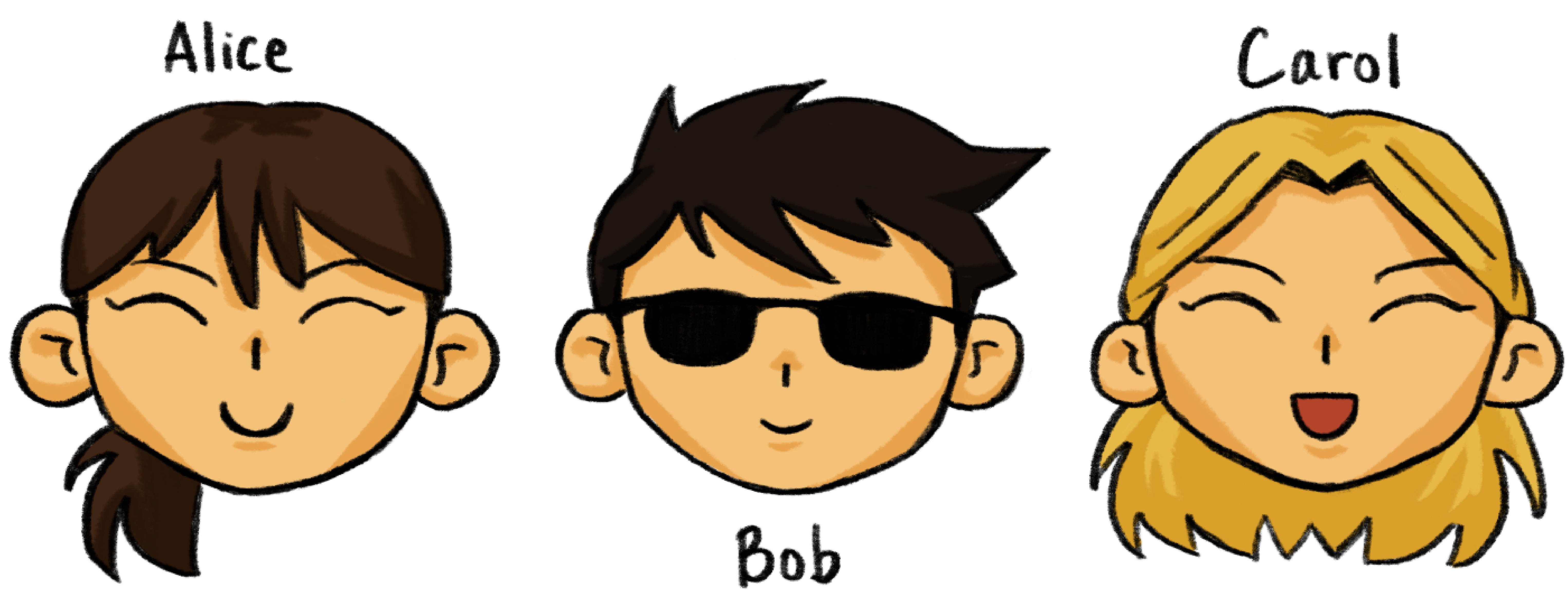
Communicating Chorrectly with a Choreography



WRITTEN BY:
Ali Ali & Lindsey Kuper

In the coming pages, you'll see these little people:



The names came from a 1978 paper by Rivest et al. on encryption, and are often used to describe communication protocols.

They can easily be shortened to A, B, and C, respectively, and will represent nodes in this zine. Nodes are entities that send and receive messages in a message-passing system.

We thought it would be fun to give them life, so you'll be seeing them interact with each other throughout this zine.

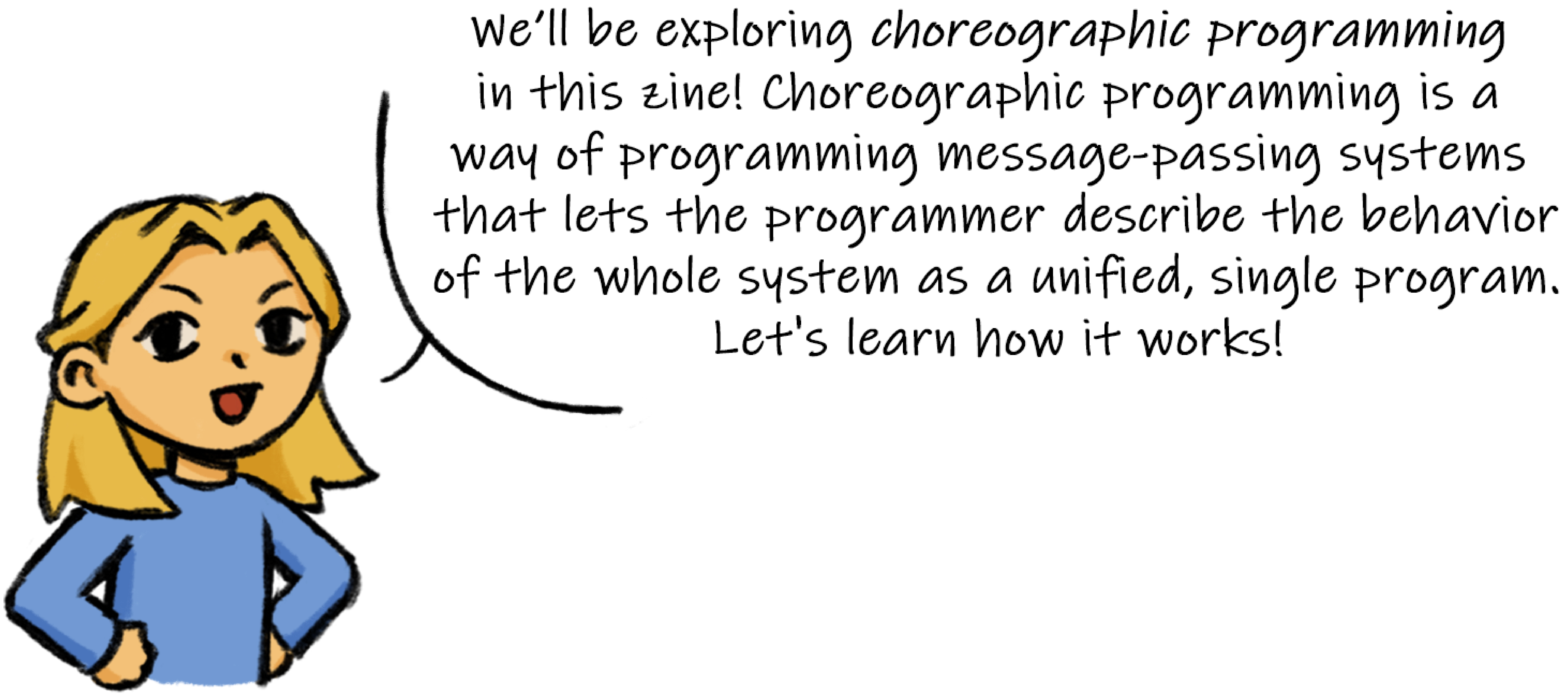


Table of Contents

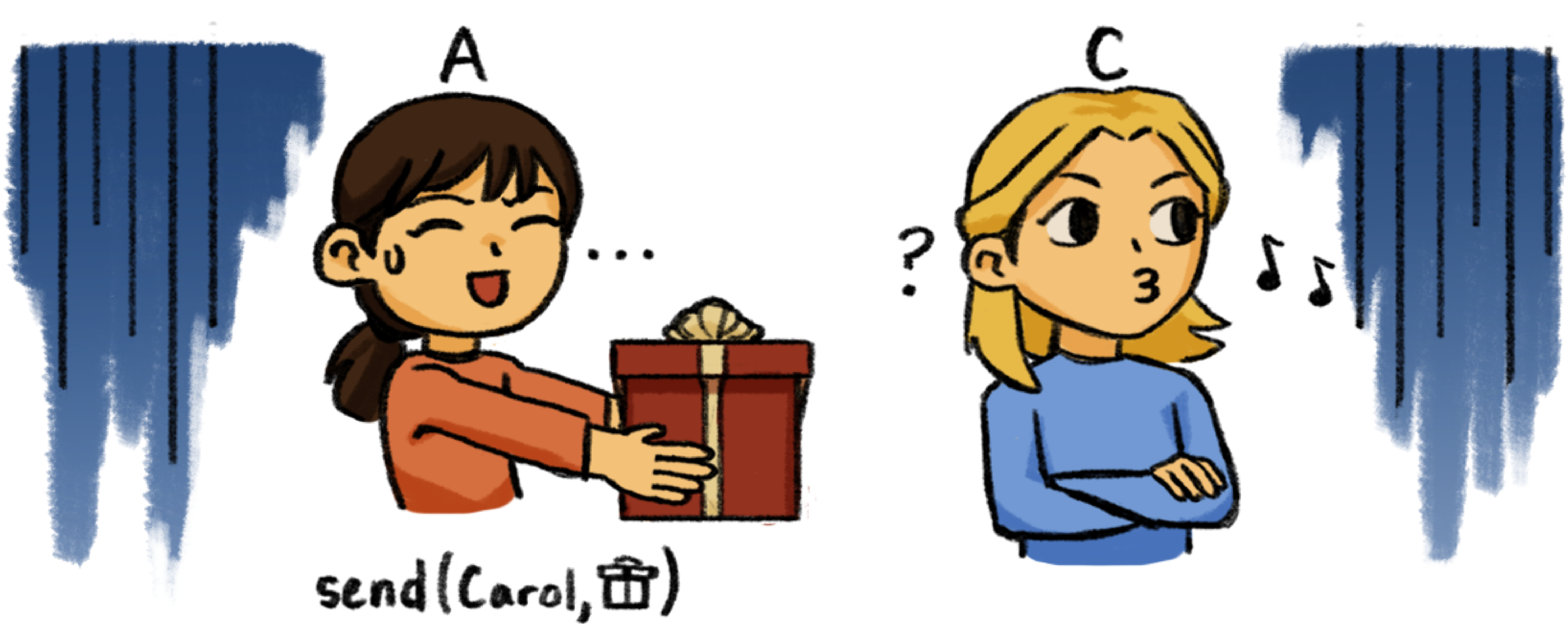
An Intro to our Systems

Before we dive into choreographic programming, we should first understand the context it exists in.

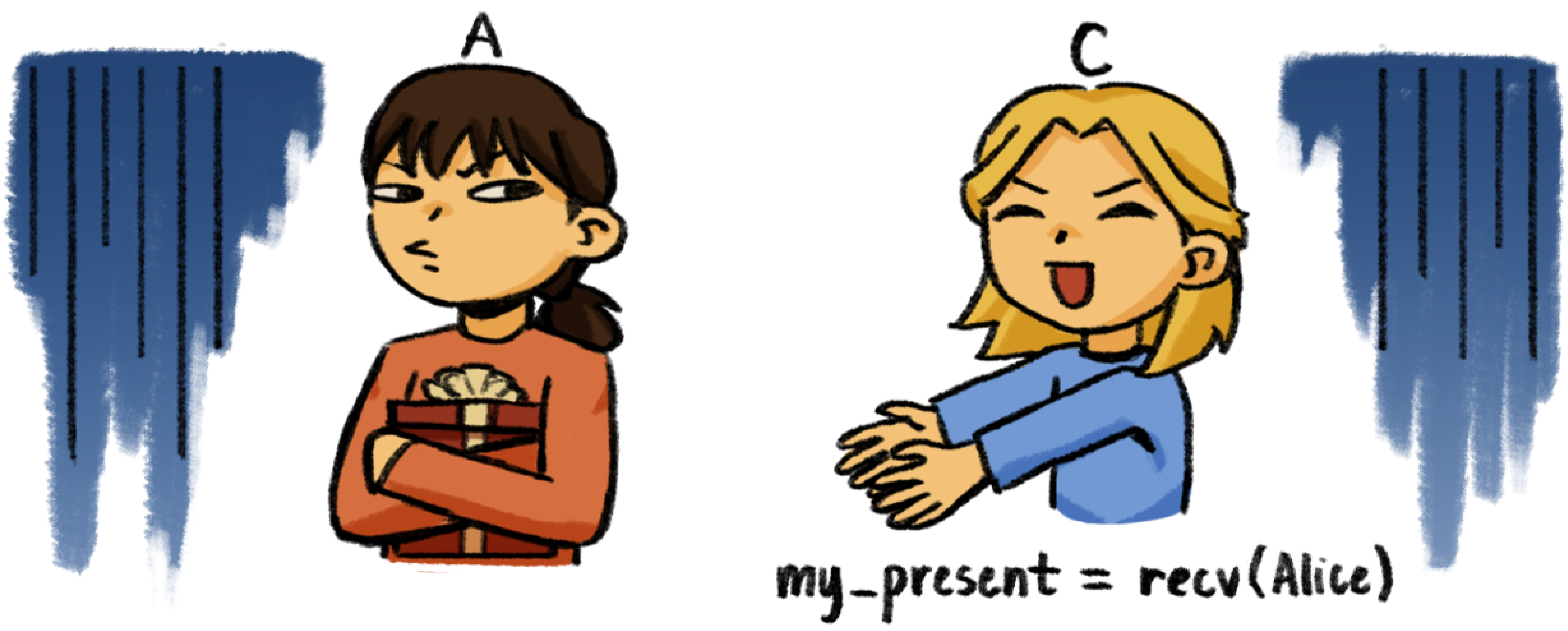
In a message-passing system, independent nodes communicate by sending and receiving messages over some kind of network. Message passing is everywhere in computing, and is especially important to the study of concurrent and distributed systems!

Every message has a sending node and a receiving node. Each node performs a sequence of sends and receives. Think of each node as running a program that makes **send()** and **recv()** calls, along with its own internal actions.

However, if the intended recipient of a sent message does not call **recv()**, they won't actually receive it.



Similarly, if a node calls **recv()** for a message that was never sent to it, then there will be nothing for the receiving node to receive!



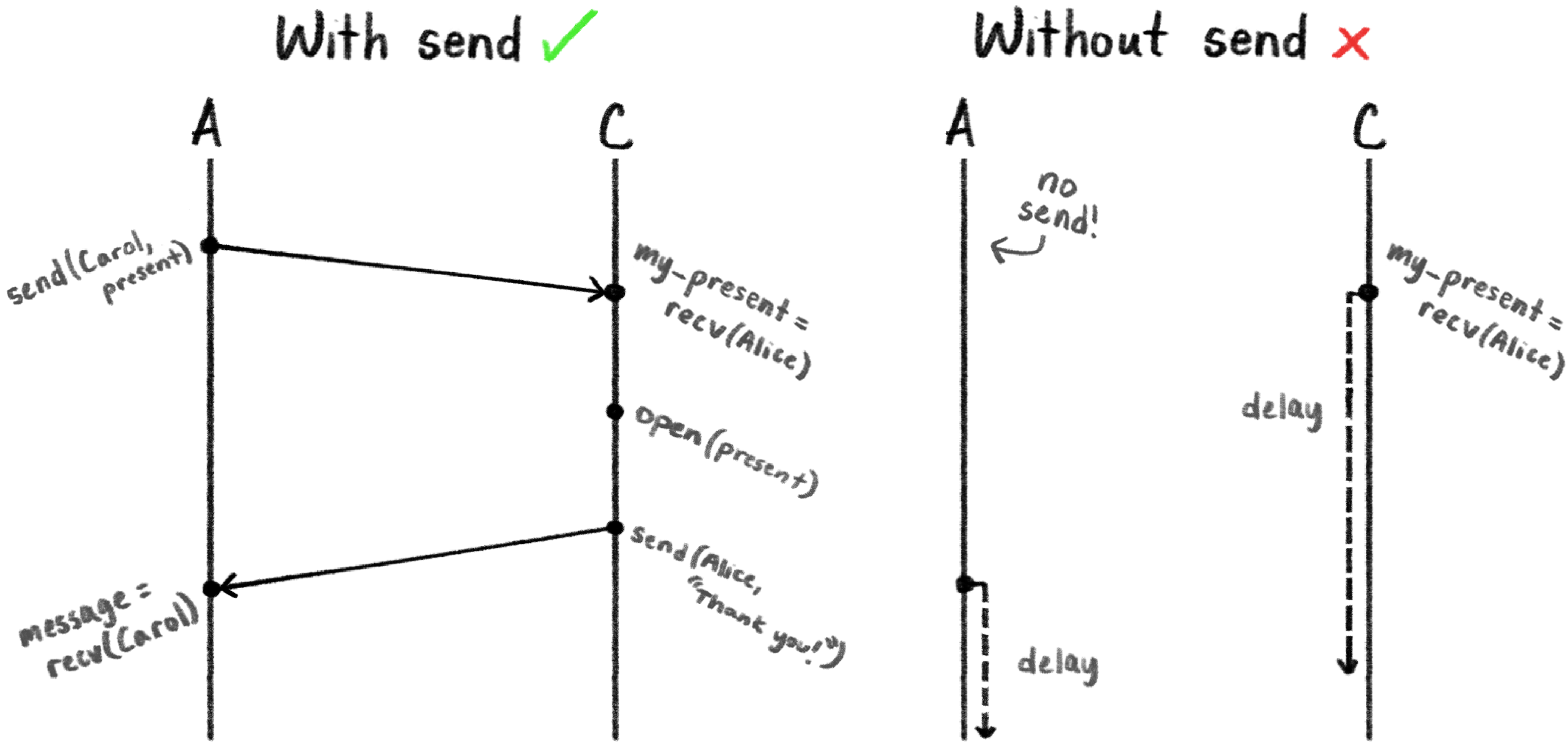
In both of these scenarios, the message passing between our nodes is unsuccessful. When this happens, it may cause a delay or a failure in the system.

Consider the following code for Alice and Carol:

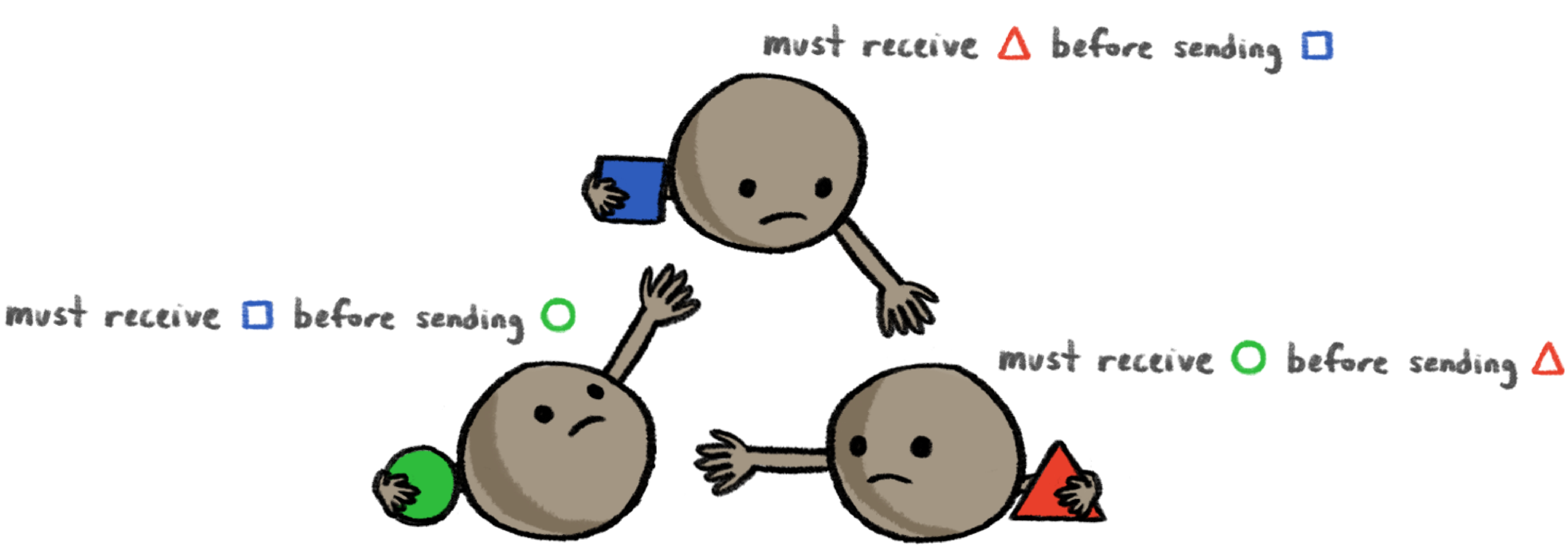
Alice	Carol
send(Carol, present) message = recv(Carol)	my_present = recv(Alice) open(my_present) send(Alice,“Thank you!”)

Now imagine if the programmer made a mistake and forgot the line **send(Carol, present)** in Alice's program. What would happen to the rest of the program's execution?

Let's assume that each event on a node is executed sequentially, meaning that an event must complete before the following one starts.



Here, things go wrong for Carol, even though it was only Alice's program that had a bug. If **send()** and **recv()** calls aren't correctly paired up with each other, nodes could crash, and the system could deadlock.



(deadlock communication error)

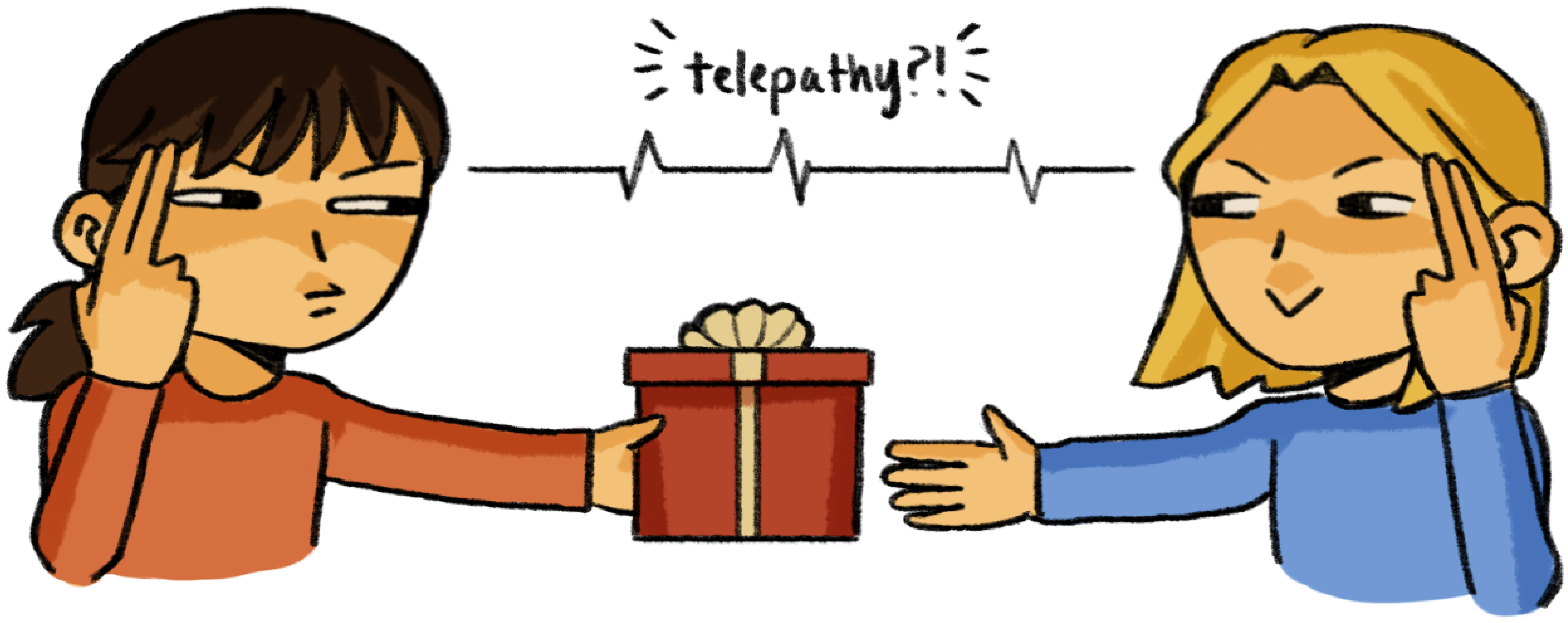
So just match each **send()** with a **recv()**, right? While it might seem pretty straightforward in the case of Alice and Carol's short exchange of messages, it quickly becomes a challenge in large systems with many interacting nodes and lots of messages.

Thankfully, this is where choreographic programming can come in!

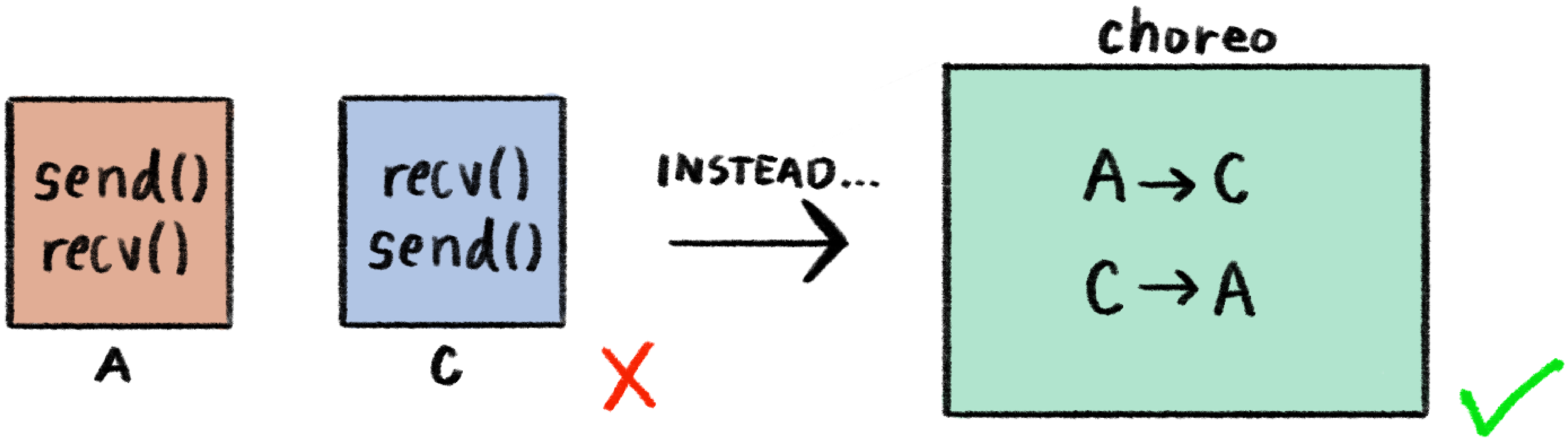
What is Choreographic Programming?

The idea of choreographic programming is to raise the level of abstraction for writing communication protocols.

Instead of writing separate programs for each node in our system, we can write a <u>choreography</u> (coordination plan), which, as the name might suggest, describes the behavior of *all* the nodes in the system.



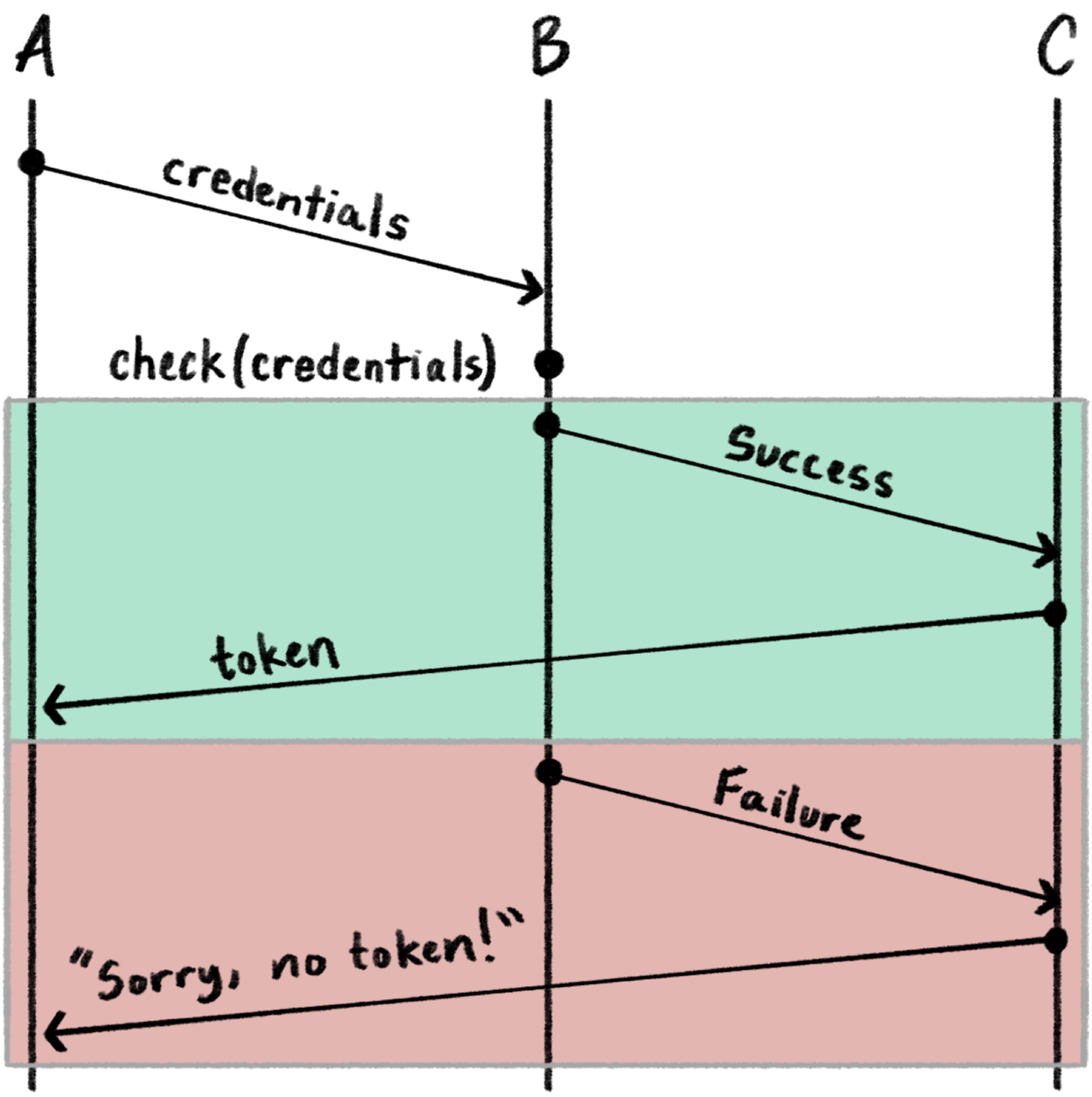
With a choreography, we can ensure that a distributed system is deadlock-free by construction. Every send will have a corresponding receive, with no mismatches that might lead to deadlock.



Choreographies can also make our program's behavior easier to understand! Instead of having to follow the flow of control as it jumps from node to node, we can read a choreography line by line.

How does it work?

Suppose Alice wants to go to Carol's birthday party, but bouncer Bob stops her to check her ID first:



In this protocol, Alice's credentials must be verified through Bob before she can access Carol's party.

Bob runs a **check()** function on Alice's communicated data, and depending on the outcome, Alice either receives an access token from Carol or does not.

This example resembles a simple sign-on protocol through an authentication service.

In a choreography, a single syntactic construct represents both the sending and the reception of a message.

`A.x -> B.y`

The `->` means 'communicate', so we can read `A.x -> B.y` as A's local variable x is communicated to B's local variable, y.

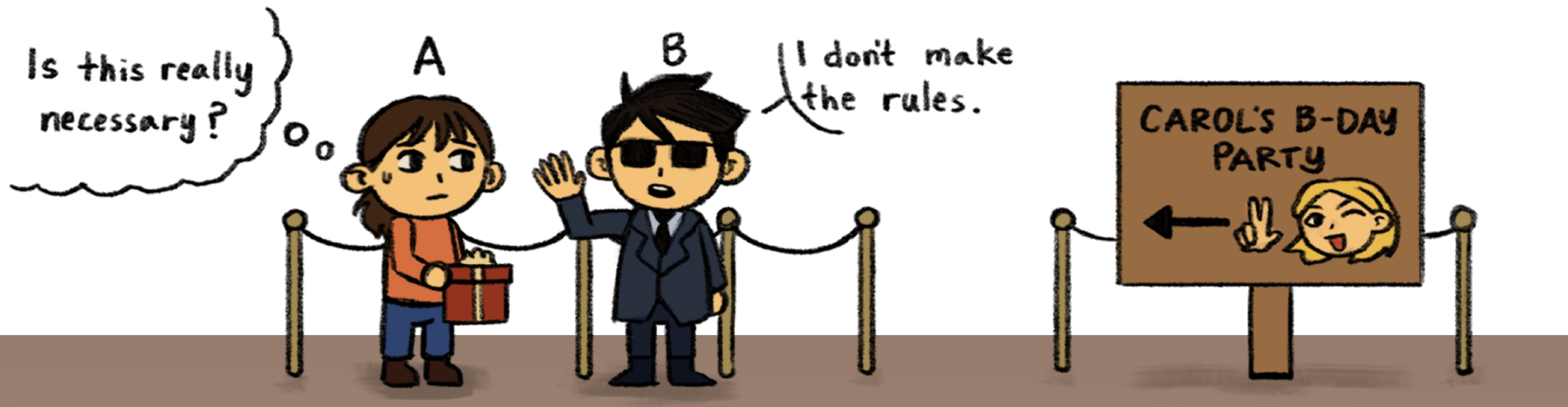
Now let's try our example, this time written as a choreography!

```
Alice.credentials -> Bob.authRequest
if Bob.check(authRequest) then
     Bob.Success -> Carol.decision
     Carol.newToken() -> Alice.result
else
     Bob.Failure -> Carol.decision
     Carol.noTokenMessage -> Alice.result
```

We're just using some made-up syntax here as a stand-in for a real language syntax.

In this choreography, there are three <u>roles</u> (or participants) defined:

- Alice (our client)
- Bob (the authentication service)
- Carol's Party (the service being accessed)



We can read the choreography as follows:
1. First, Alice's credentials are sent to Bob, to be received and stored in Bob's authRequest variable.
2. Bob checks authRequest, and again, depending on the outcome, either:

   A. a success message is sent from Bob to Carol, and a new token is sent from Carol to Alice.

   B. a failure message is sent from Bob to Carol, and an error message is sent from Carol to Alice.

If we were to rewrite this choreography as a separate program for each role, it might look something like this:

```
Alice
send credentials to Bob
recv result from Carol
```

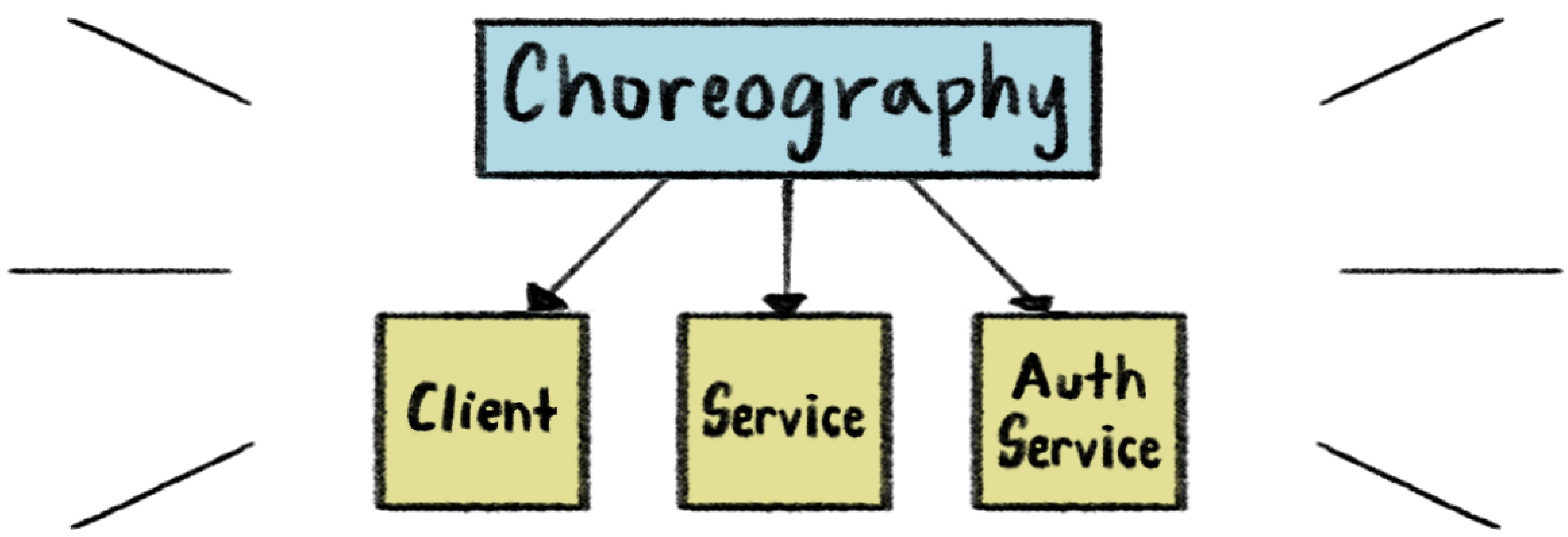
```
Carol
recv decision from Bob
switch(decision)
  case Success:
     send newToken() to Alice
  case Failure:
     send "Sorry, no token!" to Alice
```

```
Bob
recv authRequest from Alice
if check(authRequest) then
     send Success to Carol
else
     send Failure to Carol
```

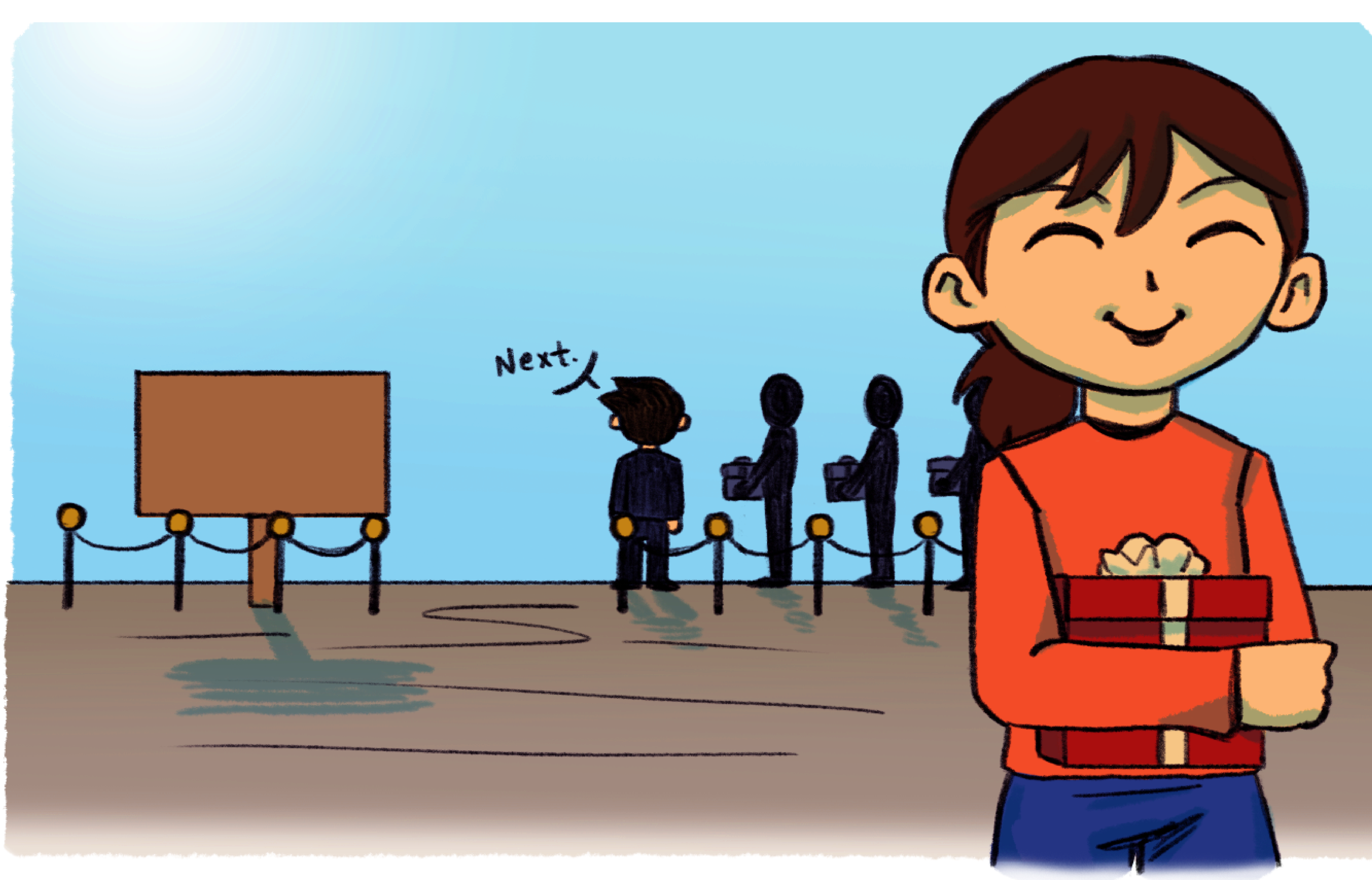
Not as easy to follow, right?
And this is just a very simple example!

It would be nice if we could write our communication protocols as choreographies from the get-go.

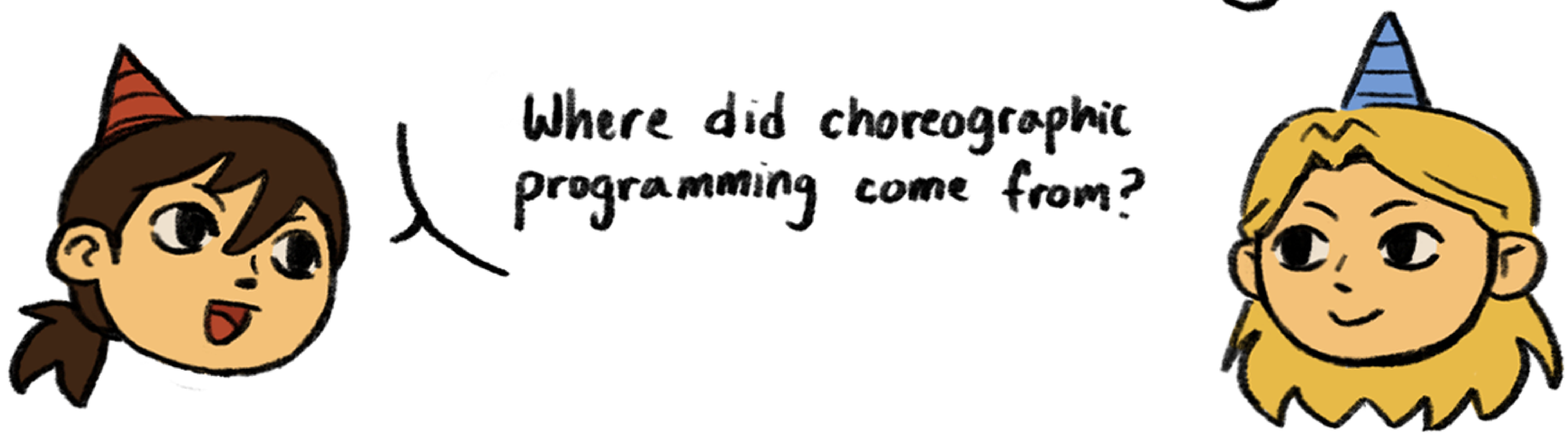
However, in order to run choreographies, we would still need to turn them into some sort of executable code. Thankfully, choreographic programming has something for this! It's called endpoint projection (EPP).



EPP takes a choreography and compiles it to a collection of several executable programs—a different one for each participating role. Then each role can run its own program. Every choreographic programming language or library provides a mechanism for EPP -- more about that later!



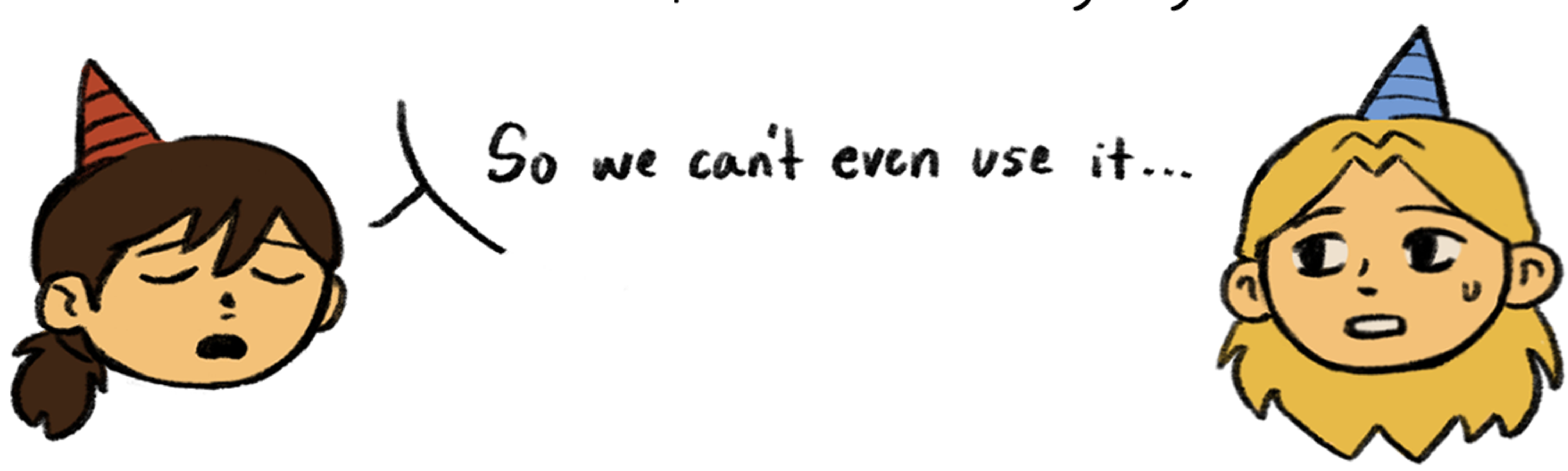
A Brief History



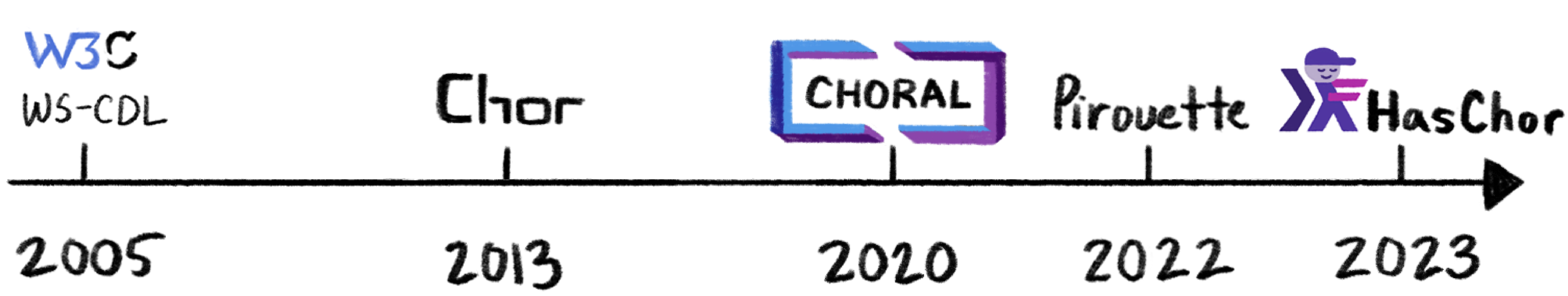
Choreographies have a long history. The **A -> B** notation for expressing communication was first used in a formal publication by Needham and Schroeder [1978] for describing a security protocol.

In the early 2000s, the W3C Web Services Choreography Working Group, alongside a group of invited experts from academia, developed a draft specification for a choreographic language.

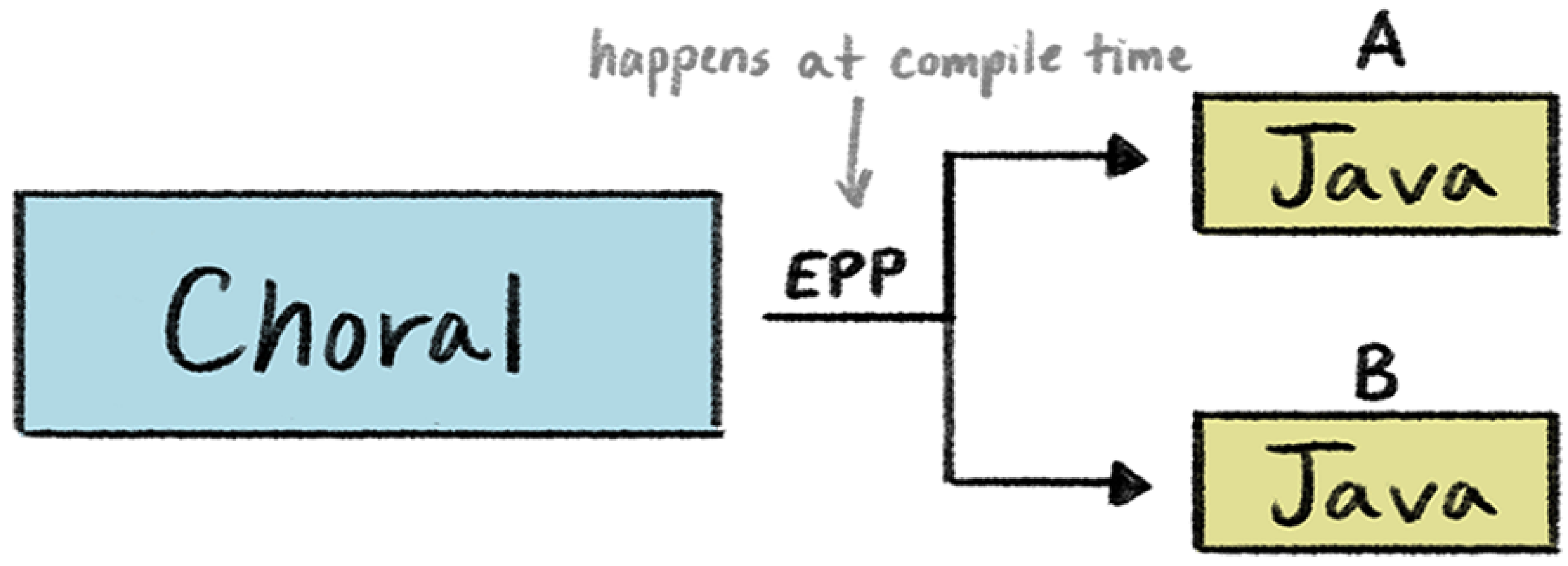
It was called Web Services Choreography Description Language, or WS-CDL for short. However, the WS-CDL language was not intended to be executable, and only formalized choreographic interactions and roles in a specification language.



Not so fast! In 2013, Chor was created by Marco Carbone and Fabrizio Montesi [Carbone and Montesi 2013]. Chor pioneered the idea of an executable choreographic programming language. It used endpoint projection to compile choreographies to runnable node-local programs.

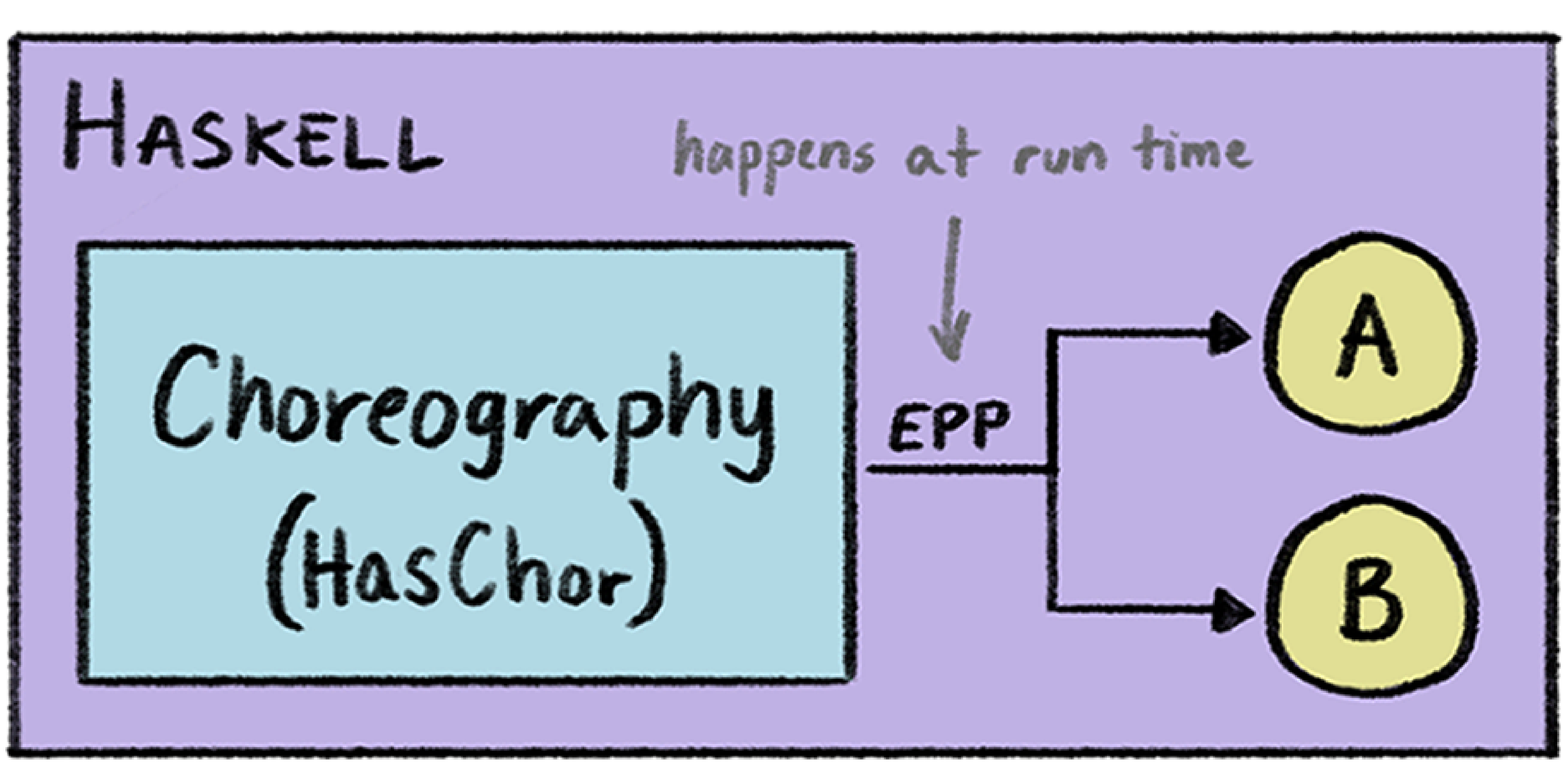


In 2020, Choral was created by Saverio Giallorenzo, Marco Peressotti, and Montesi [Giallorenzo et al. 2024]. Choral is the first choreographic programming language that can be used to program realistic software. It integrates choreographies with modern programming abstractions like functions, objects, genericity, and higher-order programming. The Choral compiler projects choreographies to executable Java code.



In 2022, Andrew Hirsch and Deepak Garg introduced Pirouette [Hirsch and Garg 2022], which combined choreographies with higher-order functional programming, leading to increased interest in choreographic programming in the functional programming research community. Pirouette is also notable for its machine-checked proof of deadlock freedom.

Then, in 2023, Gan Shen, Shun Kashiwa, and Lindsey Kuper published HasChor [Shen et al. 2023], a library for choreographic programming in Haskell. In HasChor, a choreography is a Haskell program, and EPP is carried out at run time. Because HasChor is a library rather than a standalone language, programmers have easy access to the whole Haskell ecosystem. While HasChor compromises some features compared to standalone languages, it demonstrates that we can get some of the advantages of choreographic programming without having to implement an entire new language and related support tools.



Next Steps

We hope that this zine has gotten you interested in learning more about choreographic programming. What's next?

If you're ready to try a real choreographic programming language or library, check out Choral or HasChor!

CHORAL https://www.choral-lang.org/

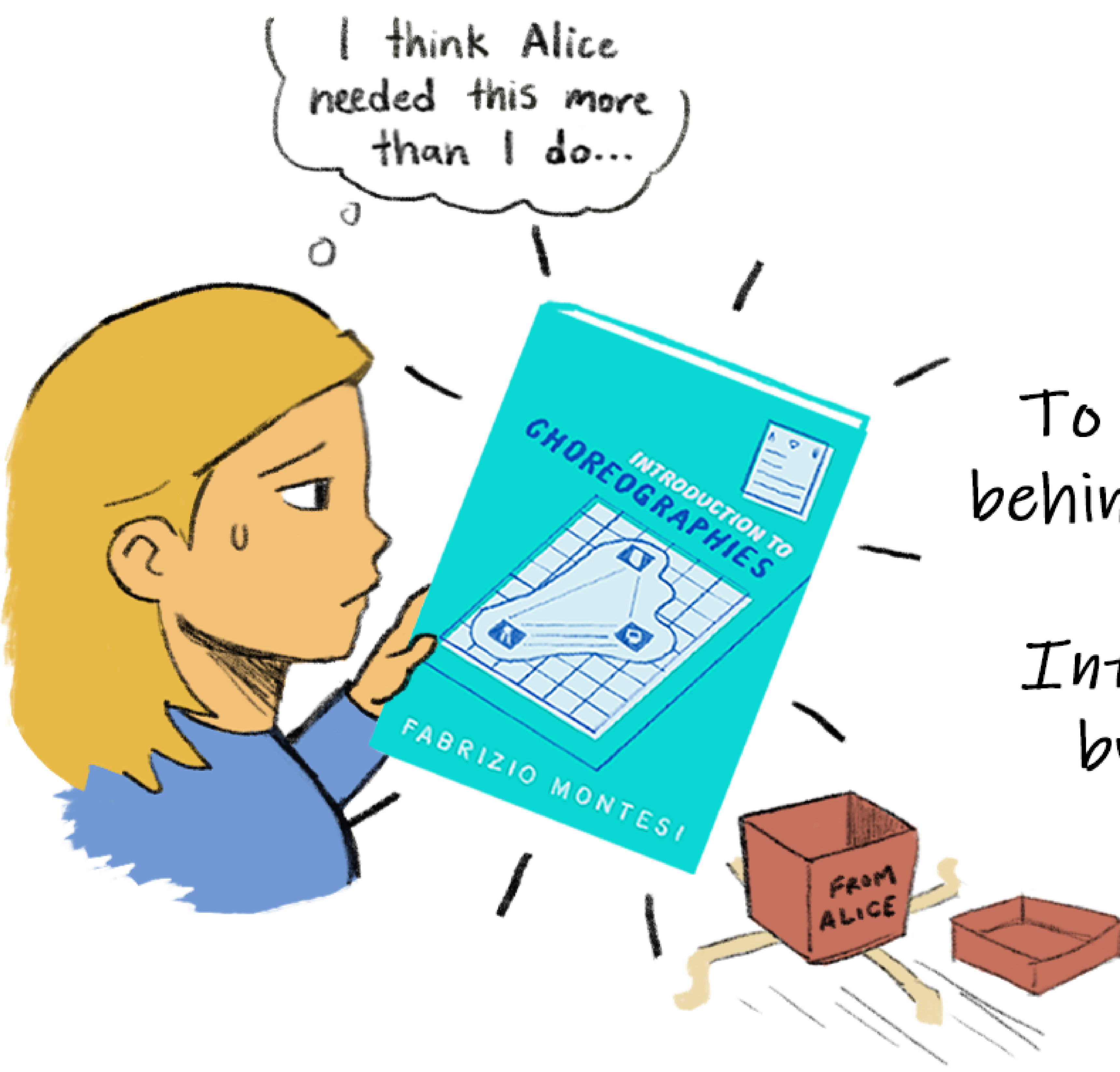
HasChor https://github.com/gshen42/HasChor

Knowledge of Choice

One of the key concepts of choreographic programming that we didn't discuss in this zine is knowledge of choice. The knowledge of choice problem comes into play when a choreography involves a conditional, like the "if ... then ...else" expression in the choreography on p. 6.

All parties affected by the conditional need to be told what branch to take! The choreography on p. 6 is handling knowledge-of-choice propagation properly, because Bob is communicating the outcome of check(authRequest) to Carol in both branches. In general, choreographic languages need a strategy for dealing with knowledge of choice (like giving you an error if you write a choreography that doesn't correctly propagate knowledge of choice, or by inserting the necessary communication for you).

Bibliography



To learn more about the theory behind choreographic programming, we recommend the book *Introduction to Choreographies* by Fabrizio Montesi [2023]!

[Carbone and Montesi 2013] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. (POPL 2013) https://doi.org/10.1145/2429069.2429101

[Giallorenzo et al. 2024] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. TOPLAS (January 2024) https://doi.org/10.1145/3632398

[Hirsch and Garg 2022] Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. (POPL 2022) https://doi.org/10.1145/3498684

[Montesi 2023] Fabrizio Montesi. Introduction to Choreographies. 2023. Cambridge University Press https://doi.org/10.1017/9781108981491

[Needham and Schroeder 1978] Roger M. Needham and Michael D. Schroeder. 1978. Using encryption for authentication in large networks of computers. CACM (December 1978) https://doi.org/10.1145/359657.359659

[Shen et al. 2023] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). (ICFP 2023) https://doi.org/10.1145/3607849

This material is based upon work supported by the National Science Foundation under Grant No. CCF-2145367.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Thanks to our friends Julia Evans and Fabrizio Montesi for feedback on a draft of this zine!