Portable, Efficient, and Practical Library-Level Choreographic Programming

SHUN KASHIWA, University of California, Santa Cruz, USA
GAN SHEN, University of California, Santa Cruz, USA
SOROUSH ZARE, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

Choreographic programming (CP) is an emerging paradigm for programming distributed applications that run on multiple nodes. In CP, instead of implementing individual programs for each node, the programmer writes one, unified program, called a *choreography*, that is then transformed to individual programs for each node via a compilation step called endpoint projection (EPP). While CP languages have existed for over a decade, *library-level* CP — in which choreographies are expressed as programs in an existing host language, and choreographic language constructs and endpoint projection are provided entirely by a host-language library — is in its infancy. Library-level CP has great potential, but the only existing implementation approaches have portability, efficiency, and practicality drawbacks that hinder its adoption.

In this paper, we aim to advance the state of the art of library-level CP with two novel techniques for choreographic library design and implementation: *endpoint projection as dependency injection* (EPP-as-DI), and *choreographic enclaves*. EPP-as-DI is a language-agnostic technique for implementing EPP at the library level. Unlike existing library-level approaches, EPP-as-DI asks little from the host language — support for higher-order functions is all that is required — making it usable in a wide variety of host languages. Choreographic enclaves are a language feature that lets the programmer define *sub-choreographies* within a larger choreography. Within an enclave, "knowledge of choice" is propagated only among the enclave participants, enabling the seamless use of the host language's conditional constructs while addressing the efficiency limitations of existing library-level implementations of choreographic conditionals. We implement EPP-as-DI and choreographic enclaves in ChoRus, the first CP library for the Rust programming language. Our case studies and benchmarks demonstrate that the usability and performance of ChoRus compares favorably to traditional, non-choreographic distributed programming in Rust.

1 INTRODUCTION

In a distributed system, a collection of independent nodes communicate with each other by sending and receiving messages. Programmers must ensure that nodes' local behaviors — sending and receiving messages, and taking internal actions — together amount to the desired global behavior of the entire system. As a very simple example, consider a distributed protocol involving nodes Alice and Bob, in which Alice sends a greeting to Bob and Bob responds. In traditional distributed programming (assuming the existence of `send` and `receive` functions that implement message transport), Alice might run a node-local program `send("Hello!", Bob); receive(Bob)`. Meanwhile, Bob would run his own node-local program `receive(Alice); send("Hi!", Alice)`. Alice and Bob depend on each other to faithfully follow the protocol: if either of them forgets to call `send`, for instance, then their counterpart will wait forever to receive a message (or time out and report an error). This approach is prone to bugs, including deadlocks.

The emerging paradigm of *choreographic programming* [Carbone and Montesi 2013; Montesi 2013; Cruz-Filipe and Montesi 2020; Giallorenzo et al. 2020; Hirsch and Garg 2022; Shen et al. 2023; Montesi 2023] offers a way to rule out this class of bugs. Instead of programming individual nodes, the choreographic programmer writes a single program, called a *choreography*, that

Authors' addresses: Shun Kashiwa, University of California, Santa Cruz, USA, Santa Cruz, shkashiw@ucsc.edu; Gan Shen, University of California, Santa Cruz, USA, Santa Cruz, gshen42@ucsc.edu; Soroush Zare, University of California, Santa Cruz, USA, Santa Cruz, sozare@ucsc.edu; Lindsey Kuper, University of California, Santa Cruz, USA, Santa Cruz, lkuper@ucsc.edu.

expresses the behavior of the entire system from an objective, third-party point of view. For example, the above protocol might be written as the choreography `Alice("Hello!") ~> Bob; Bob("Hi!") ~> Alice`. The `~>` operator denotes communication between a sender and a receiver. Choreographies are transformed into collections of node-local programs via a compilation step called *endpoint projection* (EPP) [Qiu et al. 2007; Carbone et al. 2007, 2012]. If EPP is correct, every `send` in one of the resulting node-local programs is guaranteed to have a corresponding `receive` in another node-local program, ensuring deadlock freedom [Carbone and Montesi 2013].

In the last ten years, several choreographic programming (CP) languages have been proposed [Carbone and Montesi 2013; Montesi 2013; Dalla Preda et al. 2014, 2017; Giallorenzo et al. 2020; Hirsch and Garg 2022]. However, *library-level* CP — in which choreographies are expressed as programs in an existing host language, and choreographic operators and EPP are provided entirely by a host-language library — is just beginning to emerge.

Library-level CP has the potential to immensely improve the accessibility and practicality of CP by meeting programmers where they are — in their programming language of choice, with access to that language's ecosystem. A library-level implementation of choreographic programming in a given host language would enjoy the usual advantages of embedded DSLs [Hudak 1996]: it would be installable just like any host-language library, compilable just like any host-language program, and could use any host-language-specific tools for development, debugging, and deployment. Library-level CP would also aid the integration of choreographies into larger systems, without any need for the programmer to change languages just to implement certain components choreographically. Library-level CP would fit especially nicely into a workflow in which a programmer wishes to port a *non*-distributed program — say, a turn-based game in which players sit next to each other at the same machine — to a distributed implementation in which the players interact over a network. With CP, we start and end the process with *one* program, and library-level CP further means that we need never switch languages along the way.

Given these advantages, how can we accomplish library-level CP? So far, the only existing library-level CP implementation is the recently proposed HasChor framework [Shen et al. 2023], which implements support for CP by means of a domain-specific language embedded in Haskell. In HasChor, choreographies are monadic computations in which choreographic operators such as `~>` may be used. Under the hood, the HasChor library uses a clever implementation technique based on dynamic interpretation of *freer monads* [Kiselyov and Ishii 2015] to carry out EPP.

The HasChor framework represents the current state of the art of library-level CP. However, HasChor's implementation approach has several limitations that hinder its portability, efficiency, and practicality. First, HasChor's implementation relies on Haskell-specific language features and is not easily portable to other languages that lack Haskell's particular arsenal of programming abstractions. In particular, HasChor's use of freer monads to implement EPP makes it challenging to port to other languages. Second, HasChor's implementation of EPP — which is the only existing library-level implementation of EPP for choreographies — results in inefficient runtime behavior in node-local programs compared to what standalone choreographic languages can offer. Specifically, HasChor's treatment of conditionals in choreographies results in unnecessary network traffic, making it unsuitable for use in typical distributed deployments in which network bandwidth is a scarce resource. The HasChor approach also requires programmers to use a HasChor-specific language construct for conditionals, rather than using the control flow constructs of the host language. Finally, HasChor lacks features that would make it easier to integrate choreographic components into larger, non-choreographic software systems. This is unfortunate, since the ability to seamlessly integrate choreographic and non-choreographic code should be a selling point of the library-level CP approach, as opposed to standalone CP languages.

In this paper, we aim to advance the state of the art of library-level CP by addressing the above limitations. We make the following specific contributions:

- We propose *endpoint projection as dependency injection* (EPP-as-DI), a novel and language-agnostic implementation technique for library-level CP (Section 3). Unlike the HasChor implementation approach, EPP-as-DI asks little from the host language: support for *higher-order functions* is all that is required. As such, the EPP-as-DI approach is straightforward to use in a wide variety of host languages.
- We propose a novel design and implementation technique for implementing efficient conditionals in library-level CP: *choreographic enclaves* (Section 4). Using enclaves, a programmer can sidestep the bandwidth inefficiency of a naive implementation of choreographic conditionals, while still making seamless use of the host language's conditional constructs.
- We present ChoRus, a choreographic programming library for Rust, implemented using our proposed techniques (Section 5). Along with EPP-as-DI and choreographic enclaves, ChoRus is the first CP library to support *located* arguments and return values (Section 5.2.2), a feature that aids the integration of choreographic components into larger, non-choreographic software projects. We empirically evaluate the usability and performance of ChoRus compared to traditional distributed programming in Rust (Section 6).

The ChoRus implementation, case studies and benchmarking code, and documentation are available at https://github.com/lsd-ucsc/ChoRus.

2 BACKGROUND ON CHOREOGRAPHIC PROGRAMMING

We begin in Section 2.1 with a brief overview of CP using an example implemented in the standalone CP language Choral [Giallorenzo et al. 2020]. Then, in Section 2.2, we give an overview of library-level CP using the HasChor framework [Shen et al. 2023], and we discuss the strengths and limitations of library-level CP as it stands today. For a comprehensive introduction to CP, we refer the reader to Montesi [2023, 2013].

2.1 The Elements of Choreographic Programming

To illustrate the key concepts of CP, let us consider a well-known example from the literature: the "bookseller" protocol [Carbone et al. 2007, 2012; Honda et al. 2008; World Wide Web Consortium 2006]. This protocol describes the interactions between a bookseller and a potential book buyer. First, the buyer sends the name of a book they wish to purchase to the seller. In response, the seller looks up the catalog and sends back the price of the book to the buyer, who then checks whether the price is within their budget. If the buyer has the means to purchase the book, they notify the seller and obtain an estimated delivery date. Alternatively, if the book's cost exceeds their budget, they communicate to the seller their decision not to proceed with the purchase. Figure 1 shows how this protocol might be implemented in a traditional (non-choreographic) fashion as two individual programs, running on the buyer and seller's distinct nodes. We use Python in Figure 1 as a representative mainstream programming language. We assume that the `send` and `receive` functions are provided by some library that implements network communication between nodes.

Even for simple protocols like the bookseller protocol, it is easy to introduce bugs. For example, the programmer might forget to send the decision to the seller, in which case the seller will wait indefinitely for the buyer's response, causing a deadlock. The programmer might also use different encodings for the delivery date in the buyer and seller programs, in which case the buyer will not be able to parse the delivery date sent by the seller, causing a type error.

Choreographic programming addresses these problems by letting the programmer implement a protocol as a single, unified program, called a choreography. Figure 2 shows an implementation

```python
def buyer():
    title = input()
    send(title, "seller")
    price = receive("seller")
    decision = price <= budget
    if decision:
        send(True, "seller")
        delivery = receive("seller")
    else:
        send(False, "seller")
```

```python
def seller():
    title = receive("buyer")
    price = catalog.get_price(title)
    send(price, "buyer")
    decision = receive("buyer")
    if decision:
        delivery = catalog.get_delivery(title)
        send(delivery_date, "buyer")
```

Fig. 1. The bookseller protocol implemented as individual node-local programs

```
1  String@Buyer title_buyer = UI@Buyer.input();
2  String@Seller title_seller = c.<String>com(title_buyer);
3  Integer@Buyer price = c.<Integer>com(catalog.quote(title_seller));
4  boolean@Buyer decision = price <= budget;
5  if(decision) {
6      c.<EnumBoolean>select(EnumBoolean@Buyer.True);
7      String@Seller delivery = catalog.get_delivery(title_seller);
8      String@Buyer delivery2 = c.<String>com(delivery);
9  } else {
10     c.<EnumBoolean>select(EnumBoolean@Buyer.False);
11 }
```

Fig. 2. The bookseller protocol implemented in the Choral choreographic language

of the bookseller protocol as a choreography in Choral [Giallorenzo et al. 2020], a standalone CP language. Taking Figure 2's implementation of the bookseller protocol as an example, let us consider four key elements of CP:

- **Located data and computation.** The bookseller protocol involves two locations, `Buyer` and `Seller`, and data and computation reside at one of these locations. On line 1 of Figure 2, the call to the function `input` happens at the buyer, as indicated by the `@Buyer` annotation on the function call. Likewise, the value returned by `input` is located at the buyer, which we see in its type, `String@Buyer`. Choral's type system ensures that data at one location cannot be accessed at a different location without an explicit communication.[1]
- **A unified language construct for communication.** Choreographies replace explicit calls to `send` and `receive` with a single language construct representing communication between a sender and a receiver. In the bookseller protocol, the buyer sends the title of the book to the seller (and the seller receives it) on line 2 of Figure 2, using the `com` method. Here, `com` takes a string located at the buyer and returns a string located at the seller. Additional calls to `com` on lines 3 and 8 express communications from the seller to the buyer.
- **Propagation of "knowledge of choice".** On line 4 of Figure 2, the buyer checks whether the price of the book is within their budget, and depending on the decision, the choreography takes different branches. Conditionals in choreographic programming are challenging because of the problem known as "knowledge of choice" [Castagna et al. 2011]. When the branches of a conditional expression encode different communication patterns, all affected locations must be notified of the outcome of evaluating the conditional. In Choral, the `select` method is used to express *selections*, which indicate that the choreography has

[1]While choreographic languages often represent locations at the type level, the notion of located data and computation is always present in choreographic programming, whether or not locations are made explicit in a type system like Choral's.

```
1  bookseller :: Choreo IO (Maybe Day @ "buyer")
2  bookseller = do
3    title'   <- (buyer, title) ~> seller
4    price    <- seller `locally` \un -> return (priceOf (un title'))
5    price'   <- (seller, price) ~> buyer
6    decision <- buyer `locally` \un -> return (un price' <= budget)
7
8    cond (buyer, decision) \case
9      True -> do
10       date  <- seller `locally` \un -> return (deliveryDate (un title'))
11       date' <- (seller, date) ~> buyer
12       buyer `locally` \un -> return $ Just (un date')
13     False -> do
14       buyer `locally` \_ -> return Nothing
```

Fig. 3. The bookseller protocol implemented in Haskell using HasChor [Shen et al. 2023]

taken a particular branch and propagate the information to relevant locations. On line 6, the buyer uses `select` to send `True` to the seller if the price is within the budget; otherwise, the buyer sends `False` to the seller on line 10. The seller will receive `True` or `False` and take the appropriate branch.

- **Endpoint projection.** By itself, a choreography is useful as a global specification of the behavior of a protocol. If we wish to have a runnable implementation, however, we need a way to perform *endpoint projection* (EPP).[2] EPP transforms a choreography into an individual program for each target node. For the bookseller protocol, the Choral compiler carries out EPP and generates Java programs similar to those in Figure 1.

So far, we have been using Choral to illustrate the key concepts of CP. Choral exemplifies *language-level* CP, where choreographies are programs in a standalone language with its own syntax, type system, compiler, and so on. Nearly all existing choreographic programming languages are implemented as standalone languages. Section 7 discusses Choral and other standalone choreographic languages in more detail. We now turn to *library-level* CP, the focus of this paper.

2.2 CP Implemented as a Library

As discussed in Section 1, the only existing library-level CP framework is HasChor [Shen et al. 2023], which implements CP as a Haskell library. Figure 3 shows Shen et al.'s implementation of the bookseller protocol as a Haskell program using HasChor.

With HasChor, choreographies are written as computations that run in the `Choreo` monad provided by the library. The `bookseller` choreography's type signature on line 1 of Figure 3 shows that it returns a value of type `Maybe Day` at the `"buyer"` location. In general, HasChor supports *located values* of type `a @ l`, implemented using GHC Haskell's support for type-level symbols. The `(~>)` operator, seen on lines 3, 5, and 11 of Figure 3, implements communication between a sender and a receiver and is HasChor's counterpart of the `com` method in Choral. The `locally` operator, on lines 4, 6, 10, 12, and 14 of Figure 3, implements local computation at a particular node — for instance, looking up the book's price on the seller's node (line 4), and computing whether the book is in budget on the buyer's node (line 6). A located value of type `a @ l` may be "unwrapped" and used at the specified location `l` using the special `un` function passed to `locally`. Finally, the

[2]Unlike in the literature on multiparty session types [Honda et al. 2008], in which endpoint projection refers to projecting a *global type* to a collection of *local types*, in choreographic programming we are concerned with projecting a *global program* (that is, a choreography) to a collection of *local programs*.

`cond` operator on line 8 of Figure 3 implements a choreographic conditional expression. Unlike with the Choral bookseller implementation in Figure 2, the HasChor programmer does not need to use anything like `select` to solve the knowledge-of-choice problem. Instead, in HasChor, `cond` *automatically* inserts the necessary communication to propagate knowledge of choice. While this design choice saves the programmer the tedium of writing calls to `select`, it has unfortunate consequences for efficiency, as we will discuss in Section 4.

To run **`Choreo`** computations, the HasChor framework provides a `runChoreography` function that performs endpoint projection. Given a choreography of **`Choreo`** type (such as `bookseller`) and a location name (such as `"buyer"`), `runChoreography` acts something like a just-in-time compiler: it dynamically generates (and runs) a node-local program at the specified location, by dynamically interpreting the choreography. This approach to library-level EPP is possible because HasChor implements **`Choreo`** as a *freer monad* [Kiselyov and Ishii 2015], whose operations can be given different semantics depending on the location at which they are run. For instance, in HasChor the `~>` operator is interpreted as `send` for the sender, `receive` for the receiver, and as a no-op for other participants in a choreography.

Library-level CP has the usual advantages of embedding a DSL in an existing host language, including ability to piggyback on the host language's ecosystem and tooling, a gentle learning curve for host-language users, and seamless integration with existing host-language code. HasChor enjoys all of these advantages. It is therefore tempting to directly port the HasChor library to lots of languages in which programmers might benefit from CP. A world with PyChor, JSChor, JavaChor and RustChor libraries would surely make CP more practical and accessible than it is today. Unfortunately, this "port HasChor to your favorite language" plan has some flaws:

- *Tight coupling with Haskell and monads.* HasChor's monadic implementation approach relies on Haskell-specific language features. While these implementation choices are appropriate (and elegant) in the context of Haskell, they are not necessarily easily portable to other languages. To make choreographic programming more widely accessible, a more general approach to implementing library-level CP is called for.
- *Inefficient conditionals.* HasChor's implementation of conditionals in choreographies involves broadcasting the value of the condition expression to *all* nodes participating in the choreography, even those nodes that are not involved in the execution of the conditional. Implementing conditionals efficiently is a particular challenge for library-level CP: while standalone choreographic languages can statically analyze choreographies to insert only the minimum ammount of inter-node communication needed, such an analysis would be difficult (if not impossible) to accomplish in HasChor, given its implementation approach that relies on dynamic interpretation of free monads. Therefore, HasChor's implementation of conditionals is unlikely to scale well to systems with large numbers of nodes, or those where network bandwidth is a bottleneck.
- *Lack of support for located arguments and return values.* One of the biggest advantages of library-level CP is that it can easily be integrated with existing host-language code. However, HasChor does not support providing located arguments to choreographies or returning located values from choreographies. This limitation makes it difficult to use HasChor as part of a larger application.

In summary, HasChor aims to make CP easy to *use*, and it succeeds at that goal — provided that the user is a Haskell programmer. But the HasChor design does not make CP easy to *implement* in one's language of choice, and it suffers from efficiency and practicality drawbacks. Our aim in the rest of this paper is to democratize the *implementation* of library-level choreographic programming while improving its efficiency and practicality.

$$
\begin{aligned}
a &: \text{Type} \\
l &: \text{Location} \\
a \mathbin{@} l &= \text{Local } a + \text{Remote} && \text{(Located Values)} \\
\text{Unwrap } l &= a \mathbin{@} l \to a && \text{(Unwrap)} \\
\text{Choreo } a &= \text{Ops} \to a && \text{(Choreography)} \\
\text{Ops} &= \text{Locally} \times \text{Comm} \times \text{Bcast} && \text{(Choreographic Operators)} \\
\text{Locally} &= \forall a.\ (l : \text{Location}) \to (\text{Unwrap } l \to a) \to a \mathbin{@} l && \text{(Local Computation)} \\
\text{Comm} &= \forall a.\ (s\ r : \text{Location}) \to a \mathbin{@} s \to a \mathbin{@} r && \text{(Communication)} \\
\text{Bcast} &= \forall a.\ (l : \text{Location}) \to a \mathbin{@} l \to a && \text{(Broadcast)}
\end{aligned}
$$

Fig. 4. The interface provided by the host-language library for expressing choreographies.

3 ENDPOINT PROJECTION AS DEPENDENCY INJECTION

A central concept of CP is that a single choreography exhibits different behaviors depending on the location to which it is projected. Each local computation may or may not be executed, and each communication becomes a send, a receive, or a no-op. Allowing a caller (in this case, endpoint projection) to modify the behavior of the callee (in this case, a choreography) is a common pattern in software engineering to improve code reusability and testability. One technique to achieve this is through dependency injection (DI) [Fowler 2004]. In DI, the callee receives its dependencies from the caller, who can alter the callee's behavior by providing different dependencies.

In this section, we present endpoint projection as dependency injection (EPP-as-DI), a new technique for implementing library-level choreographic programming. The key idea of EPP-as-DI is that we can implement CP by representing a choreography as a host-language function that takes *choreographic operators* as arguments. Then, endpoint projection can change the behavior of the choreography by injecting specialized implementations of the choreographic operators, depending on the projection target. This technique can be used in any host language that supports higher-order functions, enabling the straightforward implementation of choreographic programming libraries in a wide variety of languages. We introduce a simple host language as a stand-in for an arbitrary host language in Section 3.1, then show how EPP-as-DI is implemented in Section 3.2.

3.1 Choreographies as Host-Language Programs

To introduce EPP-as-DI, we assume a simple ML-like host language that supports higher-order functions. For ease of exposition in this section, our host language is typed; however, types are not essential to implement EPP-as-DI. Choreographies are expressed as host-language functions using the interface presented in Figure 4, which we now describe.

3.1.1 Located Values. We assume a set of Locations with decidable equality and write them as l. A *located value*, written $a \mathbin{@} l$, is a value of type a at location l. A located value can either be a Local a, meaning the value is at the current location, or a Remote, meaning the value is at some remote location. We maintain the invariant that, when doing endpoint projection for l, $a \mathbin{@} l$ is always a Local. To use a located value at l, it needs to be *unwrapped* first. Since it does not make sense to unwrap a remote value, we provide an Unwrap l function that can only unwrap values at l. Given a value of type $a \mathbin{@} l$, Unwrap l produces a value of type a.

```
bookseller : Choreo (Option Date @ buyer)
bookseller(locally, comm, bcast) =
  let title_buyer = locally(buyer, λ(un) → input()) in
  let title_seller = comm(buyer, seller, title_buyer) in
  let price_seller = locally(seller, λ(un) → catalog.get_price(un(title_seller))) in
  let price_buyer = comm(seller, buyer, price_seller) in
  let decision_buyer = locally(buyer, λ(un) → un(price_buyer) ≤ budget) in
  let decision = bcast(buyer, decision_buyer) in
  if decision then
    let delivery_seller = locally(seller, λ(un) → catalog.get_delivery(un(title_seller))) in
    let delivery_buyer = comm(seller, buyer, delivery_seller) in
    locally(buyer, λ(un) → Some(un(delivery_buyer)))
  else
    locally(buyer, λ(un) → None)
```

Fig. 5. Bookseller Choreography

3.1.2 Choreographies. A choreography Choreo *a* is a function that takes a set of choreographic operators Ops as dependencies and returns some result of type *a*. The host-language library interface provides three choreographic operators that are sufficient to realize the key elements of CP described in Section 2.1. We will use lower-case locally, comm, and bcast as the names of operators that have types Locally, Comm, and Bcast, respectively:

- locally performs a local computation: it takes a location and a function and runs the function locally at the location.
- comm communicates a value between two locations: it takes a sender and a receiver location, a value at the sender, and returns the same value at the receiver.
- bcast broadcasts a value to the group of locations involved in the interaction: it takes a sender location, a value at the sender, and returns a value at all locations.

We can write choreographies as functions of type Choreo *a* by using the provided choreographic operators in the body of the function. To illustrate, Figure 5 shows the bookseller protocol implemented in our notional host language using the API of Figure 4. We assume that the host language supports standard language constructs such as **let ... in** and **if ... then ... else**. The bookseller choreography uses bcast to propagate knowledge of choice and implement conditionals. When the buyer makes a decision ($\text{decision}_{\text{buyer}}$), it is broadcasted to all locations (decision). Since all locations have the same data, it is safe to use the control-flow constructs of the host language, such as **if**, to implement conditionals in choreographies.

3.2 Endpoint Projection as Injecting Dependencies

Since a choreography is a function that takes choreographic operators as dependencies, we can determine the meaning of these operators by injecting specialized implementations of them, leading to the definition of endpoint projection as a host-language function epp, shown in Figure 6. We assume the existence of send and recv functions in the host language that implement message transport, for instance, by calling into a host-language networking library. epp takes a choreography *c*, a list of locations participating in the choreography *ls*, and a target location *l*, then projects

```
epp : Choreo a → [Location] → Location → a
epp(c, ls, l) =
  let unwrap(v) = if let Local(a) = v then a else error("impossible") in
  let locally(l′, f) = if l == l′ then Local(f(unwrap)) else Remote in
  let comm(s, r, a) =
    if l == s then send(unwrap(a), r); Remote else if l == r then Local(recv(s)) else Remote in
  let bcast(s, a) = if l == s then ∀r ∈ ls. send(unwrap(a), r); unwrap(a) else recv(s) in
  c(locally, comm, bcast)
```

Fig. 6. Endpoint Projection as Injecting Dependencies

the choreography to a node-local program for the target location. Inside epp, we construct the three choreographic operators from the viewpoint of l and supply them to c:

- For operator $\mathsf{locally}(l', f)$, if l is the same as l', we perform the local computation f; otherwise, no action is taken.
- For operator $\mathsf{comm}(s, r, a)$, if l is the same as the sender location s, we perform a send of a to the receiver; or if l is the same as the receiver location r, we perform a recv from the sender; otherwise, no action is taken.
- For operator $\mathsf{bcast}(s, a)$, if l is the same as the sender location s, we perform a series of sends of a to all the locations participating in the interaction; otherwise, no action is taken.

We used the EPP-as-DI technique to implement ChoRus, a choreographic programming library for Rust. We describe the design and implementation of ChoRus in Section 5.

4 EFFICIENT CONDITIONALS WITH CHOREOGRAPHIC ENCLAVES

As discussed in Section 2.1, implementing conditionals in choreographic programming is challenging because of the "knowledge of choice" problem [Castagna et al. 2011]. A CP language must ensure — either statically or dynamically — that choreographies propagate knowledge of the outcome of evaluating a conditional expression to all locations that are affected by the choice. If CP is implemented as a standalone language, then the compiler can perform static analysis to check this property, and a choreography that fails to propagate knowledge of choice is deemed *unprojectable*. Standalone CP languages can even support choreography *amendment* [Cruz-Filipe and Montesi 2020; Lanese et al. 2013; Basu and Bultan 2016; Cruz-Filipe and Montesi 2023], a procedure that determines if a choreography is unprojectable as-is and then automatically inserts the minimum necessary communication to make it projectable.

Without access to the full AST of programs in the CP language, however, static analysis becomes infeasible. In particular, with both the EPP-as-DI approach of Section 3 and in HasChor's freer-monad-based approach, we cannot perform static analysis on choreographies to determine how knowledge of choice needs to be propagated. With static analysis off the table as an option, then propagation of knowledge of choice needs to be handled some other way. In Section 3, we solved the problem in a naive way by implementing conditionals with broadcast, which ensures that *all* locations receive the knowledge of choice, whether they are affected by the choice or not. HasChor's cond operator internally uses broadcast as well. Not only does this naive approach introduce unnecessary communication, it may cause an undesired leak of information to locations who should not have it.

Alternatively, we could do without static analysis another way: by requiring the programmer to provide annotations to convey their intent. In fact, in the absence of choreography amendment, this is the typical approach even in standalone CP languages: the programmer must annotate the branches of a conditional with selection annotations that indicate to the compiler that knowledge of choice must be propagated, as we see in the Choral code in Figure 2 that uses the `select` method. Yet the approach of adding selection annotations is somewhat unsatisfying, because we must add annotations to make our code *correct* (that is, projectable). If we must annotate our code for the benefit of the compiler, it would be preferable if we could begin with a choreography that is *correct, but inefficient*, and then add annotations to make it *efficient*.

In this section, we address this design challenge with *choreographic enclaves*, a novel CP language feature. Enclaves are sub-choreographies that execute at a specified subset of the locations involved in a larger choreography. One may broadcast within an enclave, just like in any other choreography, but the broadcast will only go to those locations that are in the specified subset. Enclaves allow finer control over the propagation of knowledge of choice, enabling an efficient implementation of conditionals in library-level CP without static analysis.

In Section 4.1, we present a variant of the bookseller protocol to motivate the need for fine-grained control over propagation of knowledge of choice. Then, in Section 4.2 we introduce choreographic enclaves. We define an enclave operator and present its type signature and implementation, and we show how to implement the two-buyer protocol with enclave and compare it with the naive approach and the selection-annotation approach.

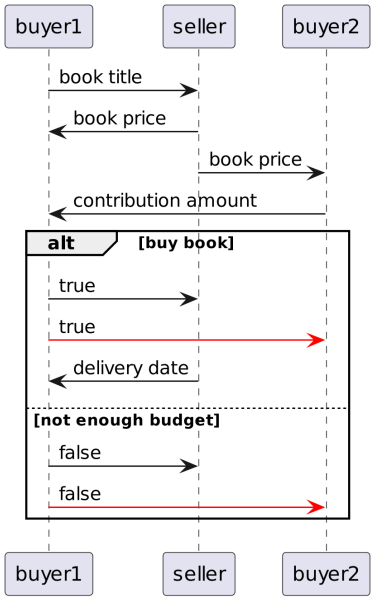
4.1 The Two-Buyer Protocol

To illustrate the problem of inefficient conditionals, let us consider a variant of the bookseller protocol: the *two-buyer* protocol [Honda et al. 2008; Hirsch and Garg 2022]. In this protocol, there are two buyers who wish to collectively buy a book from the seller. First, buyer1 sends the title to the seller, and the seller sends the price to both buyers. Then, buyer2 tells buyer1 how much they can contribute, and buyer1 decides whether to buy the book by comparing the price with the buyers' combined budget. If buyer1 decides to buy the book, they send their intent to buy to the seller, and the seller sends the delivery date to buyer1. Otherwise, buyer1 tells the seller that they will not buy the book.

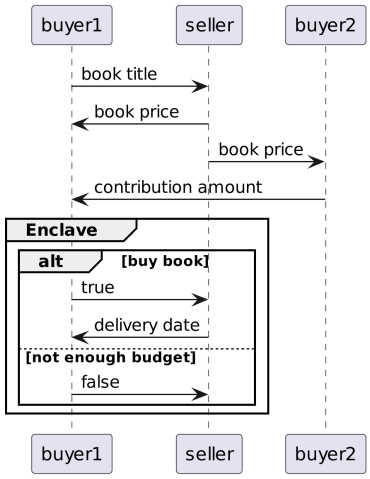
Using the bcast choreographic operator that we introduced in Section 3.1, we can implement the two-buyer protocol in our notional host language, as shown in Figure 7. Figure 8a shows a sequence diagram of the execution of the protocol. After buyer1 makes a decision, to perform the conditional, it broadcasts the decision to all locations, i.e., the seller and buyer2. It is important that the seller receives the decision because the seller needs to know whether to send a delivery date to buyer1, but buyer2 does not need to receive the decision, as its subsequent behavior does not depend on it. Nonetheless, because of broadcast, buyer2 receives the decision, causing unnecessary communication between buyer1 and buyer2, shown in red in Figure 8a. While this communication does not affect the correctness of the choreography, it is inefficient and can be problematic in more complex choreographies with many participants. Moreover, it leaks information about choice, which can be a security concern. For example, buyer2 might infer the budget of buyer1 by observing the decision, and this type of information leakage might be undesirable in some applications.

```
two_buyer : Choreo (Option Date @ buyer1)
two_buyer(locally, comm, bcast) =
  let title_buyer1 = locally(buyer1, λ(un) → input()) in
  let title_seller = comm(buyer1, seller, title_buyer1) in
  let price_seller = locally(seller, λ(un) → catalog.get_price(un(title_seller))) in
  let price_buyer1 = comm(seller, buyer1, price_seller) in
  let price_buyer2 = comm(seller, buyer2, price_seller) in
  let contribution = comm(buyer2, buyer1, buyer2_budget) in
  let decision_buyer1 = locally(buyer1, λ(un) → un(price_buyer1) ≤ buyer1_budget + contribution) in
  let decision = bcast(buyer1, decision_buyer1) in
  if decision then
    let delivery_seller = locally(seller, λ(un) → catalog.get_delivery(un(title_seller))) in
    let delivery_buyer1 = comm(seller, buyer1, delivery_seller) in
    locally(buyer1, λ(un) → Some(un(delivery_buyer1)))
  else
    locally(buyer1, λ(un) → None)
```

Fig. 7. A naive version of the two-buyer protocol with bcast



(a) Using broadcast naively



(b) With enclave

Fig. 8. Sequence diagrams of the two-buyer protocol with and without enclave

```
two_buyer : Choreo (Option Date @ buyer1)
two_buyer(locally, comm, bcast, enclave) =
  ...
  let decision_buyer1 = locally(buyer1, λ(un) → un(price_buyer1) ≤ buyer1_budget + contribution) in
  let c(locally, comm, bcast, enclave) =
    let decision = bcast(buyer1, decision_buyer1) in
    if decision then
      let delivery_seller = locally(seller, λ(un) → catalog.get_delivery(un(title_seller))) in
      let delivery_buyer1 = comm(seller, buyer1, delivery_seller) in
      locally(buyer1, λ(un) → Some(un(delivery_buyer1)))
    else
      locally(buyer1, λ(un) → None)
  in
  enclave([buyer1, seller], c)
```

Fig. 9. A more efficient version of the two-buyer protocol using an enclave

4.2 The Enclave Operator

To prevent unnecessary communication, we introduce the enclave choreographic operator. The enclave operator executes a sub-choreography at a specified set of locations. Inside the sub-choreography, the broadcast operator sends data only to locations in the specified set. This allows us to perform conditionals without sending data to unaffected locations.

We extend the interface of our host-language library from Figure 4 to add support for an enclave operator with the type Enclave, specified below:

$$\text{Ops} = \text{Locally} \times \text{Comm} \times \text{Bcast} \times \text{Enclave} \qquad \text{(Choreographic Operators)}$$

$$\text{Enclave} = \forall a, l.\ [\text{Location}] \to \text{Choreo}\ a\ @\ l \to a\ @\ l \qquad \text{(Enclave)}$$

The first argument to enclave is a list of locations where the sub-choreography is to be executed, and the second argument is the sub-choreography. It returns the result of running the sub-choreography. To implement endpoint projection for enclave, we update the definition of epp from Figure 6 as follows:

```
epp : Choreo a → [Location] → Location → a
epp(c, ls, l) =
  ...
  let enclave(ls′, c′) = if l ∈ ls′ then epp(c′, ls′, l) else Remote in
  c(locally, comm, bcast, enclave)
```

The enclave operator recursively calls the sub-choreography by calling epp with the sub-choreography and the list of locations where the sub-choreography is executed if the projection target is one of the specified locations. The behavior of bcast inside the sub-choreography depends on the ls' argument to the recursive call to epp, so bcast inside the sub-choreography will only send data to the specified locations. Using enclave, we can rewrite the last part of the two-buyer protocol, as shown in Figure 9. After buyer1 makes a decision, we define a sub-choreography c that uses bcast to perform conditionals. Then, we call the sub-choreography at buyer1 and seller

using enclave. Because the sub-choreography is not executed at buyer2, bcast does not send the decision to buyer2, as shown in Figure 8b.

While we have shown how to implement endpoint projection for enclave using the EPP-as-DI technique, the use of choreographic enclaves is orthogonal to the use of EPP-as-DI. For instance, one could extend HasChor with an enclave operator without departing from HasChor's freer-monad-based implementation of EPP.

5 CHORUS: LIBRARY-LEVEL CHOREOGRAPHIC PROGRAMMING FOR RUST

In this section, we present ChoRus, the first choreographic programming library for the Rust programming language. ChoRus is implemented using EPP-as-DI, supports choreographic enclaves, and has other features that make it a practical choice for distributed programming in Rust. We describe how we encode EPP-as-DI in Rust Section 5.1, and give a brief tour of ChoRus features Section 5.2. The code shown in this section is simplified for presentational purposes. ChoRus is open source, and its implementation, case studies and benchmarking code, and documentation are available at https://github.com/lsd-ucsc/ChoRus.

5.1 EPP-as-DI in ChoRus

5.1.1 Locations. ChoRus represents each location at which node-local code runs as a distinct type. In Rust, we can create a new type by defining a struct. Locations must be comparable for equality to perform endpoint projection. To that end, ChoRus defines the `ChoreographyLocation` trait, which all location types must implement:

```
trait ChoreographyLocation: Copy {
    fn name() -> &'static str;
}
```

The `name` method returns the string representation of the location, which is used to compare locations for equality. Thanks to Rust's macro system, `ChoreographyLocation` can be derived automatically. For example, the following code defines a location named `Alice`:

```
#[derive(ChoreographyLocation)]
struct Alice;
```

5.1.2 Located Values. Located values are values that reside at a specific location. ChoRus defines the `Located<V, L1>` struct to represent a located value of type `V` at location `L1`:

```
struct Located<V, L1: ChoreographyLocation> {
    value: Option<V>,
    phantom: PhantomData<L1>,
}
```

The `value` field holds a value of type `Option<V>`; it is `Some` if the current projection target is `L1` and `None` otherwise. We use `std::marker::PhantomData` to indicate to the compiler that the `L1` parameter is not used at run time.

5.1.3 Choreography Trait. In Section 3, we represented choreographies as functions. To provide a more ergonomic API, ChoRus represents choreographies as structs that implement the `Choreography` trait. The `Choreography` trait is defined as follows:

```
trait Choreography<R = ()> {
    fn run(self, op: &impl ChoreoOp) -> R;
}
```

The `R` type parameter represents the return type of the choreography. The `run` method takes a reference to an object that implements the `ChoreoOp` trait, which provides the choreographic operators.

```rust
trait ChoreoOp {
    fn locally<V, L1: ChoreographyLocation>(
        &self,
        location: L1,
        computation: impl Fn(Unwrapper<L1>) -> V,
    ) -> Located<V, L1>;
    // ...
}
```

Fig. 10. The `ChoreoOp` trait (excerpt).

5.1.4 ChoreoOp Trait. ChoRus supports the four choreographic operators `locally`, `comm`, `broadcast`, and `enclave`, as described in Section 3 and Section 4.

Figure 10 shows an excerpt of the `ChoreoOp` trait that implements the `locally` operator. The `locally` method takes a location `location` and a function `computation` and returns a `Located` value. The `computation` function takes an argument of type `Unwrapper<L1>`, which it can use to unwrap located values at location `L1`. Other choreographic operators are defined similarly as methods of the `ChoreoOp` trait.

5.1.5 Transport. The `Transport` trait represents the message transport layer. Users can implement the `Transport` trait by providing the `send` and `receive` methods. ChoRus has two built-in transport implementations: `LocalTransport` and `HttpTransport`. The `LocalTransport` implementation models each location as a thread and uses an inter-thread channel to send messages. The `HttpTransport` implementation uses HTTP to send messages.

5.1.6 Endpoint Projection. ChoRus provides the `Projector` struct to perform endpoint projection and execute choreographies. First, users construct a `Projector` by passing the projection target and the transport. Then, they can call the `epp_and_run` method to perform endpoint projection and execute the choreography. The `epp_and_run` method takes a choreography, defines `EppOp` — an object that implements `ChoreoOp` for the projection target — and calls the `run` method of the choreography with it. Figure 11 shows an excerpt of the `epp_and_run` method and the implementation of `locally`.

5.2 Advanced Features

ChoRus supports all the features supported by HasChor [Shen et al. 2023], such as swappable transport backends, higher-order choreographies, and location polymorphism. In this section, we present two new features of ChoRus: *location sets* and *located input/output*.

5.2.1 Location Sets. In ChoRus, the set of locations at which a choreography runs is represented at the type level. We call this type the *location set* of the choreography. Each choreography has an associated type `L` that represents its location set. `ChoreoOp` is parametrized by the location set of the choreography and prevents users from using locations that are not in the location set. For example, the following code defines a choreography `AliceBobChoreography` that runs on locations `Alice` and `Bob`. `LocationSet!` is a macro that constructs a special location set type. Inside the `run` method, we can only use locations `Alice` and `Bob`. If we try to use location `Carol`, the Rust compiler will report an error. Location sets are especially useful when defining choreographic enclaves, as they prevent us from accidentally using locations outside the enclave.

5.2.2 Located Input/Output. When a choreography is used as part of a larger program, it is often useful to be able to pass *located* values to and from the choreography. For example, consider a simple password authentication protocol between a client and a server. The client reads a password

```
impl<...> Projector<...> {
    pub fn epp_and_run<...>(&'a self, choreo: C) -> V {
        struct EppOp<...> {...}
        impl<...> ChoreoOp for EppOp<...>
        {
            fn locally<V, L1: ChoreographyLocation>(
                &self,
                location: L1,
                computation: impl Fn(Unwrapper<L1>) -> V,
            ) -> Located<V, L1> {
                if L1::name() == Target::name() {
                    let value = computation(Unwrapper::new());
                    Located::local(value)
                } else {
                    Located::remote()
                }
            }
            // ...
        }
        choreo.run(&EppOp {...})
    }
}
```

Fig. 11. The `epp_and_run` method of the `Projector` struct.

```
struct AliceBobChoreography;
impl Choreography for AliceBobChoreography {
    type L = LocationSet!(Alice, Bob);
    fn run(self, op: &impl ChoreoOp<Self::L>) {
        op.locally(Carol, |_| println!("Hello from Carol!"));
    }
}
```

Fig. 12. Invalid use of location `Carol` in `AliceBobChoreography`.

from the user and sends it to the server. The server checks the password and sends the result back to the client. The client then prints the result. The inputs to this choreography are (1) the typed password on the client, and (2) the correct password on the server, and the output is the result of the authentication on the client. Morally, these are all *located* values; for example, when running the choreography on the client, we do not have access to the correct password on the server. However, from outside the choreography, we do not have a way to talk about their locations. To solve this problem, ChoRus provides a *located input/output* feature that provides a convenient and type-safe way to handle located values.

`Projector` plays an important role in the located input/output feature. `Projector` is parameterized by the projection target and can construct (1) local located values at the projection target, and (2) remote located values at other locations. It can also unwrap located values at the projection target, but not at other locations.

Figure 13 shows the password authentication choreography and code to execute the choreography as the client and as the server. When running the choreography as the client, we use an instance of `Projector` that is parameterized by the client location. We provide the password attempt as a local located value and the correct password as a remote located value on the server. Conversely, when running the choreography as the server, we provide the password attempt as a

```rust
struct PasswordAuthChoreography {
    attempt_password: Located<String, Client>,
    correct_password: Located<String, Server>,
}
impl Choreography<Located<bool, Client>> for PasswordAuthChoreography {
    type L = LocationSet!(Client, Server);
    fn run(self, op: &impl ChoreoOp<Self::L>) -> Located<bool, Client> {
        let password = op.comm(Client, Server, &self.attempt_password);
        let result = op.locally(Server, |un| {
            un.unwrap(&password) == un.unwrap(&self.correct_password)
        });
        op.comm(Server, Client, &result)
    }
}
```

(a) Password authentication choreography

```rust
let result = client_projector.epp_and_run(PasswordAuthChoreography {
    attempt_password: client_projector.local("1234".to_string()),
    correct_password: client_projector.remote(Server),
});
println!("Result: {}", client_projector.unwrap(result));
```

(b) Client code

```rust
server_projector.epp_and_run(PasswordAuthChoreography {
    attempt_password: server_projector.remote(Client),
    correct_password: server_projector.local("password".to_string()),
});
```

(c) Server code

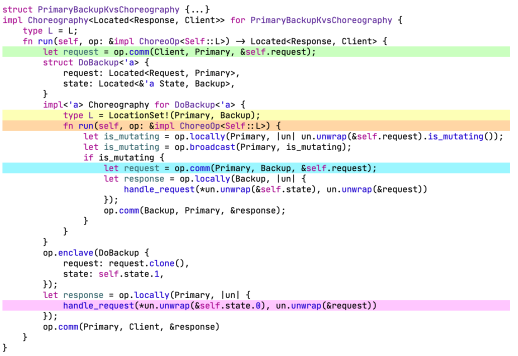
Fig. 13. The password authentication choreography (a), along with node-local code to invoke it on the client (b) and server (c).

remote value and the correct password as a local value. The result can only be unwrapped at the client location using the `unwrap` method of `Projector`.

6 EVALUATION

In this section, we assess the utility and practicality of library-level CP with ChoRus. First, to demonstrate that ChoRus indeed brings the advantages of CP to Rust, we present a case study involving a key-value store (Section 6.1). In this case study, we implement a simple replicated key-value store as a choreography and as a traditional Rust program. We compare these two implementations and highlight how choreography helps to track the flow of data and control. Next, to illustrate that library-level CP enables code reuse, we conduct a second case study: a multiplayer tic-tac-toe game (Section 6.2). We begin by showing the code for a tic-tac-toe game that runs locally, then we use ChoRus to modify the program to run across multiple computers over a network with minimal changes. We observe that library-level CP allows a substantial portion of the local code to be reused for the distributed implementation. Finally, we measure the performance overhead incurred by using ChoRus (Section 6.3). Through benchmarking, we show that ChoRus introduces very minimal overhead, making it sufficiently practical for use.

(a) Handwritten Rust KVS



(b) Choreographic ChoRus KVS

Fig. 14. Comparison of the flow of control between the handwritten and choreographic KVS

6.1 Case Study 1: Replicated Key-Value Store

To demonstrate how ChoRus helps developers to implement distributed systems, we consider a simple replicated key-value store. Our key-value store supports two operations: `get` and `put`. The `get` operation takes a key and returns the value associated with the key. The `put` operation takes a key and a value, and associates the key with the value. Our system consists of three nodes: `Client`, `Primary`, and `Backup`. The `Client` node takes a request from the user and sends the request to the `Primary` node. The `Primary` node checks the type of the request. If the request is a `get` request, it looks up the requested key in its local state and returns the response to the client. If the request is a `put` request, it forwards the request to the backup node. The backup node updates its local state and returns the response to the `Primary` node. Once `Primary` receives the response from `Backup`, it applies the update to its local state and returns the response to the client.

While the protocol is simple, implementing it is error-prone. Figure 14a shows the implementation of the protocol *without* using choreographic programming. The code defines three functions for each node. The highlight and arrows show the flow of data between the nodes. Because sends and receives are interleaved, it is difficult to track the flow of data and control.

Figure 14b shows the implementation of the same protocol as a choreography in ChoRus. The choreography communicates the request from `Client` to `Primary` using `comm`. Then, it uses the `enclave` operator to call the `DoBackup` sub-choreography at `Primary` and `Backup`. The sub-choreography branches on the type of the request, and if the request is `put`, it forwards the request to the backup node. After the sub-choreography returns, the primary node processes the request and returns the response to the client. The choreographic version is easier to understand because both data and control naturally flow from top to bottom.

While we could implement the KVS protocol as a choreography in HasChor, the naive implementation of conditionals in HasChor would present a problem. When we branch on the type of the request on the primary node, it broadcasts the type, and in HasChor, this broadcast would also go to the client, leaking an implementation detail. By using enclaves, we can implement the protocol in a more efficient (and secure) manner.

6.2 Case Study 2: Multiplayer Tic-Tac-Toe

An advantage of library-level choreographic programming is that it allows developers to reuse existing code. This is especially useful for implementing a distributed version of an existing local

```
let mut board = Board::new();
loop {
    board = brain_for_x.think(&board);

    if !board.check().is_in_progress() {
        break;
    }
    board = brain_for_o.think(&board);

    if !board.check().is_in_progress() {
        break;
    }
}
```

```
let mut board = Board::new();
loop {
    board = op.broadcast(
        PlayerX,
        op.locally(PlayerX, |un| un.unwrap(&self.brain_for_x).think(&board)),
    );
    if !board.check().is_in_progress() {
        break;
    }
    board = op.broadcast(
        PlayerO,
        op.locally(PlayerO, |un| un.unwrap(&self.brain_for_o).think(&board)),
    );
    if !board.check().is_in_progress() {
        break;
    }
}
```

Fig. 15. Diff between the local Rust and distributed ChoRus implementations of the tic-tac-toe game

program. In this case study, we implement a distributed version of a tic-tac-toe game using ChoRus. We start with a local implementation of the game where two players play on the same computer. Then, we use ChoRus to port the local implementation to a distributed version, which lets the players play on different computers over the network, with minimal changes to the code. Finally, we compare the ChoRus implementation with a handwritten distributed version of the game.

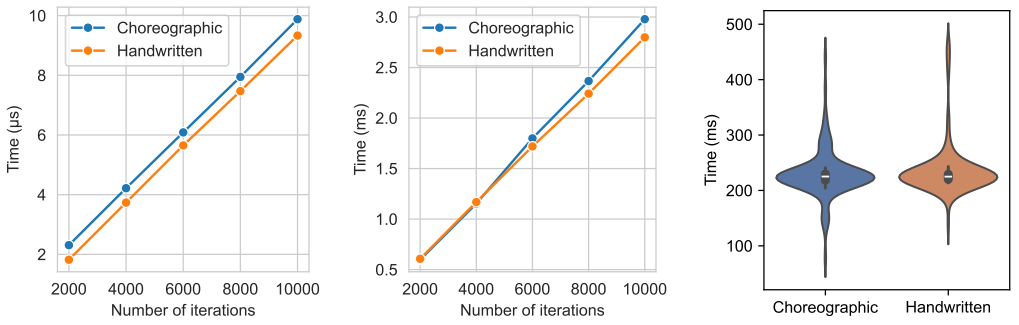
Let us start with the local implementation of the game. The left side of Figure 15 shows the structure of the main game loop written in Rust. We omit the definitions of the structs and traits that capture the core logic of the game, such as `board`, `brain_for_x`, and `brain_for_o`. The game starts with an empty board. Then, the game enters a loop in which the two players take turns to make a move. After each move, we check the status of the board, and if the game is over, we break out of the loop. Finally, we print the result of the game.

Because ChoRus is a library, we can reuse the existing local Rust code to implement the distributed version of the game. The right side of Figure 15 shows the distributed implementation of the game written as a choreography in ChoRus. For brevity, we only show the `run` method of the choreography. Just like its local counterpart, the choreography starts with an empty board. Then, the choreography enters a loop in which the two players make a local move and broadcast the new board. After each move, we check the status of the board, and if the game is over, we break out of the loop. Finally, we print the result of the game from the perspective of each player.

As highlighted in Figure 15, changing the local implementation to the distributed implementation requires minimal changes to the code. All game logic and control flow are reused, and the only changes are the addition of the `locally` and `broadcast` operators to specify the location of data and computation. This is a significant advantage of library-level CP as opposed to a standalone CP language, because it allows developers to reuse existing code for local computation and focus on the distributed aspects of the program.

6.3 Performance

To employ CP in production, the performance overhead of using CP must be acceptable. Performance is a particular concern for library-level CP, which involves carrying out EPP at runtime. In this section, we measure the performance overhead of using ChoRus compared to traditional distributed programming in Rust. We focus on the overhead of running a choreography with EPP-as-DI. We conducted two experiments. First, we performed microbenchmarking to measure the overhead of EPP-as-DI in isolation. Second, we measured and compared the performance of the two versions of the key-value store from Section 6.1. All experiments in this section were performed on a MacBook Pro 2020 with an Apple M1 chip, 16 GB of RAM, and macOS Sonoma 14.0.



(a) locally (b) comm (c) KVS Benchmark Results

Fig. 16. Benchmark Results

6.3.1 Microbenchmarks. With microbenchmarking, we measured the performance overhead of using two of the choreographic operators in ChoRus: `locally` and `comm`.

To measure the overhead of the `locally` operator, we implemented a simple counter program as a handwritten Rust program and as a ChoRus choreography. The program initializes a counter and repeatedly increments it a given number of times. The ChoRus version is written as a choreography that runs only at one location and uses the `locally` operator to perform initialization and increments. We use endpoint projection to execute the choreography. We measured the runtime of the two versions of the program with different numbers of iterations. Figure 16a shows the result of the microbenchmark. There is a small, constant overhead of using ChoRus. Because the overhead does not grow with the number of iterations, the overhead is likely due to the cost of endpoint projection, and there is no observable overhead of using the `locally` operator.

We also measured the performance of the `comm` operator. We implemented a simple protocol that moves data from one location to another as a handwritten Rust program and as a ChoRus choreography. In both the handwritten Rust and the ChoRus versions, to isolate the performance overhead of using EPP, we used ChoRus' `LocalTransport` message transport layer to send data between the two locations. Figure 16b shows the result. The message passing is dominating the running time in both versions, and the overhead of endpoint projection is not observable. The ChoRus version performed slightly worse for larger iterations, with a difference of <0.5 ms.

6.3.2 Key-Value Store Benchmark. We also benchmarked the two versions of the key-value store from Section 6.1 to measure the system-level performance overhead of using ChoRus . We generated 100 random requests of 50% `get` and 50% `put` requests. We measured the runtime of the two versions of the program. We used `HttpTransport` for communication between nodes in both versions. Figure 16c shows a violin plot of 100 runs of the benchmark. The median runtime of the choreographic version was 225.09 ms, while the median runtime of the handwritten version was 224.93 ms. Even though the nodes are running on the same computer, the running time is dominated by the network latency, and we did not observe significant overhead of using ChoRus.

7 RELATED WORK

Choreographies were originally a specification mechanism for distributed systems [World Wide Web Consortium 2006]. Researchers soon began to explore the notion of endpoint projection for choreographies [Mendling and Hafner 2005; Qiu et al. 2007; Carbone et al. 2007, 2012; Lanese et al. 2008; McCarthy and Krishnamurthi 2008]. The Chor language [Carbone and Montesi 2013; Montesi

2013] pioneered the use of choreographies as executable programs by means of endpoint projection. Much of the subsequent literature on CP places emphasis on its formal foundations [Cruz-Filipe et al. 2021; Hirsch and Garg 2022; Pohjola et al. 2022; Cruz-Filipe et al. 2022; Graversen et al. 2023] rather than on practically usable implementations. An exception to this rule is Choral [Giallorenzo et al. 2020], arguably the most practical option among existing CP languages. Choral has been used to implement widely used real-world protocols as choreographies [Lugović and Montesi 2023]. ChoRus and Choral share practicality and accessibility as goals, but make different design decisions in service of those goals. As a Rust library, ChoRus enjoys the advantages of library-level CP discussed in Section 1; as a standalone language with its own compiler, Choral has limited tooling and IDE support, although it prioritizes interoperability with Java, the target language for its EPP. On the other hand, as a standalone CP language, Choral can statically generate local code for each endpoint to run, whereas ChoRus (and HasChor [Shen et al. 2023], the only other library-level CP implementation, which we discussed in Section 2.2) must dynamically generate node-local programs at run time.

ChoRus and Choral differ in their treatment of message transport. Choral provides a hierarchy of *channel types*, a very flexible abstraction that enables use of many types of user-defined channels within a single choreography. The closest counterpart in ChoRus is the `Transport` trait. While users may implement their own `Transport` types, they must choose a single implementation of `Transport` for each `Projector`. This design decision is orthogonal to ChoRus's implementation of CP as a library, and a Choral-like channel abstraction would be feasible at the library level. For our ChoRus case studies so far, though, the existing `Transport` mechanism has sufficed.

ChoRus and Choral also handle propagation of knowledge of choice differently. As we have seen, ChoRus supports choreographic enclaves (Section 4), which improve on the naive broadcast-based approach used in previous library-level CP implementations [Shen et al. 2023]. Choral, like other choreographic languages before it [Carbone and Montesi 2013; Montesi 2013], supports selection annotations via the `select` method. Although enclaves are more fine-grained than naive broadcast, selection annotations give the programmer even more fine-grained control over how knowledge of choice is propagated. Not all standalone CP languages use selection annotations; for instance, the language AIOC [Dalla Preda et al. 2014, 2017] uses an endpoint projection approach that automatically broadcasts to the locations involved in both branches of a conditional expression — a granularity similar to what enclaves offer, made possible by the ability to carry out static analysis on the branches.

ScalaLoci [Weisenburger et al. 2018] is a language for *multitier* programming [Weisenburger et al. 2020], implemented as an embedded DSL in Scala. Multitier programming can be seen as a close relative of CP [Giallorenzo et al. 2021], and ChoRus and ScalaLoci are both "library-level": a ScalaLoci program is a Scala program, just as a ChoRus program is a Rust program. We observe that ScalaLoci — and, perhaps, multitier programming generally — seems to be a natural fit for asynchronous, reactive distributed applications, while ChoRus — and perhaps CP generally — seems to be a natural fit for synchronous, turn-based distributed applications, such as our tic-tac-toe case study or password authentication example. We posit that the library level is a fruitful place to continue to explore the evident relationship between multitier and choreographic programming.

HasChor [Shen et al. 2023] uses freer monads to embed CP in Haskell. Another popular technique to implement eDSLs in functional languages is *tagless final* [Carette et al. 2007], where programmers use typeclasses to express abstract operations whose implementations are provided implicitly as typeclass instances. Our EPP-as-DI technique is similar in spirit, but uses higher-order functions to explicitly pass choreographic operators as function arguments.

8 CONCLUSION

Library-level choreographic programming holds great promise. In this paper, we aim to democratize library-level CP with techniques that make CP possible, efficient, and practical to implement in a wide variety of host languages. In particular, we presented *endpoint projection as dependency injection (EPP-as-DI)*, a technique for library-level implementation of EPP in any host language that supports higher-order functions, and *choreographic enclaves*, a language feature that improves on the efficiency of existing library-level CP. We have implemented EPP-as-DI and choreographic enclaves in ChoRus, a library for CP in Rust. Our case studies and benchmarks demonstrate that programming wtih ChoRus compares favorably with traditional distributed programming in Rust. We hope that in the future, our proposed techniques will bring CP libraries to many host languages, and a subsequent expansion and diversification of CP in many communities.

REFERENCES

Samik Basu and Tevfik Bultan. 2016. Automated Choreography Repair. In *Fundamental Approaches to Software Engineering*, Perdita Stevens and Andrzej Wąsowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 13–30.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2007. Structured Communication-Centred Programming for Web Services. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–17.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured Communication-Centered Programming for Web Services. *ACM Trans. Program. Lang. Syst.* 34, 2, Article 8 (June 2012), 78 pages. https://doi.org/10.1145/2220365.2220367

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/2429069.2429101

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems*, Roberto Bruni and Juergen Dingel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28.

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional choreographic programming. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 212–237.

Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theoretical Computer Science* 802 (2020), 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

Luís Cruz-Filipe and Fabrizio Montesi. 2023. Now It Compiles! Certified Automatic Repair of Uncompilable Protocols. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:19. https://doi.org/10.4230/LIPIcs.ITP.2023.11

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021. Formalising a Turing-Complete Choreographic Language in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:18. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2017. Dynamic Choreographies: Theory And Implementation. *Logical Methods in Computer Science* Volume 13, Issue 2 (April 2017). https://doi.org/10.23638/LMCS-13(2:1)2017

Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. 2014. AIOCJ: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings 7*. Springer, 161–170.

Martin Fowler. 2004. Inversion of control containers and the dependency injection pattern. https://martinfowler.com/articles/injection.html

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Object-Oriented Choreographic Programming. https://doi.org/10.48550/ARXIV.2005.09520

Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty languages: The choreographic and multitier cases. In *ECOOP 2021-European Conference on Object-Oriented Programming*.

Eva Graversen, Andrew K Hirsch, and Fabrizio Montesi. 2023. Alice or Bob?: Process Polymorphism in Choreographies. *arXiv preprint arXiv:2303.04678* (2023).

Andrew K Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–27.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (dec 1996), 196–es. https://doi.org/10.1145/242224.242477

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. Bridging the Gap between Interaction- and Process-Oriented Choreographies. *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods* (2008), 323–332. https://api.semanticscholar.org/CorpusID:11388743

Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2013. Amending Choreographies. In *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013 (EPTCS, Vol. 123)*, António Ravara and Josep Silva (Eds.). 34–48. https://doi.org/10.4204/EPTCS.123.5

Lovro Lugović and Fabrizio Montesi. 2023. Real-World Choreographic Programming: An Experience Report. *arXiv preprint arXiv:2303.03983* (2023).

Jay A. McCarthy and Shriram Krishnamurthi. 2008. Cryptographic Protocol Explication and End-Point Projection. In *European Symposium on Research in Computer Security*. https://api.semanticscholar.org/CorpusID:15429446

Jan Mendling and Michael Hafner. 2005. From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL. In *Proceedings of the 2005 OTM Confederated International Conference on On the Move to Meaningful Internet Systems* (Agia Napa, Cyprus) *(OTM'05)*. Springer-Verlag, Berlin, Heidelberg, 506–515. https://doi.org/10.1007/11575863_70

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

Fabrizio Montesi. 2023. *Introduction to Choreographies*. Cambridge University Press.

Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:18. https://doi.org/10.4230/LIPIcs.ITP.2022.27

Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. 2007. Towards the Theoretical Foundation of Choreography. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. Association for Computing Machinery, New York, NY, USA, 973–982. https://doi.org/10.1145/1242572.1242704

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All. *Proc. ACM Program. Lang.* 7, ICFP (Aug. 2023). https://doi.org/10.1145/3607849

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A survey of multitier programming. *ACM Computing Surveys (CSUR)* 53, 4 (2020), 1–35.

The World Wide Web Consortium. 2006. Web Services Choreography Description Language: Primer. https://www.w3.org/TR/ws-cdl-10-primer/