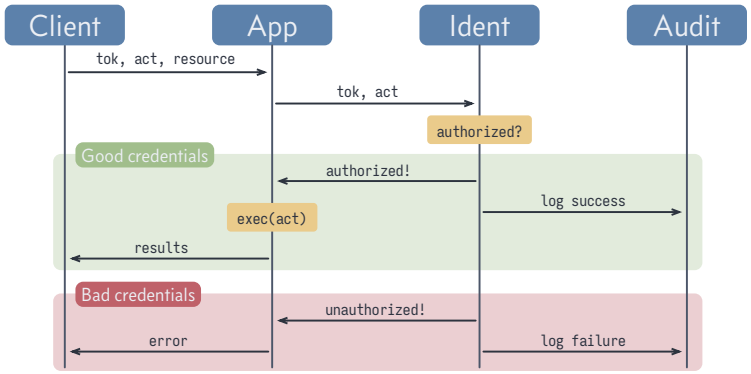
Event-Driven Choreographies

Ashton Wiersdorf

Contents

1. The travails of distributed systems

Distributed systems allow us to scale our software at the cost of increased complexity. Modern software systems are often built out of distributed components that must coordinate to accomplish high-level tasks. For example, an application might need to work with an *identity service* that manages user authentication and authorization and a *logging service* to maintain an audit trail of state changes. In order to perform any action on behalf of a client, the application would need to ensure that the client has the requisite permissions. The following design diagram describes how these services might interact in a typical system:




```
Client.{tok, act, r} ~> App.{tok, act, r}
App.{tok, act} ~> Ident.{tok, act}

# Ident uses internal `authorized?` function
# to decide if client is authorized
if Ident.authorized?(tok, act) do
  # Client is authorized; perform action
  App.exec(act, r) ~> Client.results
  Client.report_results(results)
  Audit.log_success(tok, act)
else
  # Client is not authorized
  App.(tok, act) ~> Audit.(tok, act)
  Audit.log_failure(tok, act)
end
```

Figure 1: A high-level design document for client performing an action requiring authorization. On the left: a sequence diagram for the 4 parties involved. On the right: a corresponding choreography.

With a design document, it is easy to get a high-level view of how the system is supposed to work. Such a document might be found buried in the internal documentation for the platform or on a whiteboard in a programmer's office. It is then up to the programmer to convert the high-level description into code for each of the components. In this case, the four components `Client`, `App`, `Ident`, and `Audit` each need an implementation that reflects their own view—or *projection*—of the overall design. The programmer must be careful to ensure that every `send` in one component has a corresponding `receive` in the destination component: a mismatch can lead to deadlock.

A better situation would be to have a way to mechanically transform the design document as seen above into components for each of the distributed parts. This idea is called *choreographic programming* (CP) [1], and programs that carry out the mechanical transformation are called *choreography compilers*. These mechanized implementations are *correct by construction*: a choreography compiler will ensure that for every `send` there is a matching `receive` in the proper actors. This eliminates the danger of writing a deadlock bug whilst enabling the user to use a high-level description (such as the right half of Figure 1) as code. Specifically, choreographic programming ensures that distributed systems are free from deadlocks and mismatched messages (i.e. mistaking one message for another).

Choreographic programming is a nascent area of research; existing realizations of choreographic programming are limited in expressiveness. While some real-world applications have been built (see Section 6) to demonstrate CP's potential for practical impact, some applications are currently beyond the scope of any CP system. In particular, existing choreographic systems do not account for complex process lifecycles and delegate event handlers and coordination between choreographic sessions to external code where it becomes more difficult to reason about. Giving choreographies the ability to express dynamic, event-driven choreographies can eliminate bugs and make large systems more comprehensible and maintainable for programmers.

My dissertation will expand the expressiveness of choreographic programming in the following ways:

Event-driven choreographies Many distributed systems are naturally described in terms of events and event handlers. Choreographies ought to be able register event handlers rather than relying on external glue code to catch and process all events.

Census primitives Systems will often need to spin up ephemeral processes to handle tasks from a job queue or perform some work asynchronously. These processes may need to coordinate with other processes, but existing choreographic systems relegate process creation and lifecycle management to the interstitial code around the choreography.

First-class process references Common patterns in distributed systems sometimes require delegation and substitution—e.g. selecting a worker for a client from a pool. Allowing choreographies to select and manipulate references to processes as first-class values will make expressing such patterns straightforward. First-class process references also enable event-driven choreographies to work with

To demonstrate the practicality and desirability of these extensions, I will be augmenting my existing choreographic programming library for the Elixir language—Chorex—so that it can run programs that require constructs beyond what existing theories and implementations can support.

1.1. My thesis

Event handlers, census primitives, and first-class process references are essential features for choreographies to take control of their lifecycle. Moreover, these features can be implemented through metaprogramming, simultaneously reducing the burden of implementation and eliminating friction for end-users.

Choreographies convert deadlocks from a distributed logic problem into a language problem where it can be mechanically remedied. This is the thesis of language-oriented programming: internalize system-level concerns (i.e. communication errors in a distributed system) as language-level constructs (i.e. ~> notation) to solve problems. [2] In this sense, internalizing actor lifecycles is a natural step in expanding the expressiveness and usefulness of choreographic programming: processes are *the primary unit* in distributed systems and choreographies ought to internalize the mechanisms for creating, stopping, and coordinating processes instead of leaving that to external code.

2. The state of Chorex

Chorex is a library that enables choreographic programming in Elixir. Chorex transforms descriptions nearly identical to the code in Figure 1 into *real, runnable Elixir code*. Chorex is built using Elixir's macro system and requires no separate build step to perform projection (unlike some other choreographic programming systems, e.g. Choral [3], [4]). This also allows Chorex to integrate tightly with Elixir tooling such as its language server [5]. Other library-based systems, such as HasChor [6], typically perform endpoint projection at runtime and can perform very little analysis leading to inefficiencies in the projected code. Chorex uses macro expansion time to analyze the choreography and avoid such inefficiencies.

Chorex works by expanding a choreography into a separate module for each role in the choreography. These roles are realized as stateful processes—called *GenServers* in Elixir's parlance—which can send messages to each other to perform computation. Chorex in its existing state is a performant and capable system, and it will serve as a solid foundation for the next version I plan to build as part of my dissertation.

Elixir is the target language of choice because of its metaprogramming facilities and its concurrency model. Elixir macros are syntactic, hygienic, and procedural, which makes Elixir one of the best languages for embedded language development [7]. The underlying Erlang VM (the BEAM) provides primitives for creating and monitoring message-passing processes.

Chorex's chief contributions to choreographic programming are that it works entirely via metaprogramming and that it can handle crashing actors via a `checkpoint/rescue` syntax reminiscent of `try/rescue`. This section will explore Chorex as it currently stands.

2.1. The language of Chorex

To write a choreography using Chorex, create a module, `import Chorex`, specify roles in a list after the `defchor` macro, and write the choreography in functions in the body. Listing 1 shows a small example choreography involving two actors: `Alice` and `Bob`.

```
defmodule MyChor do
  import Chorex

  defchor [Alice, Bob] do
    def run(Alice.(title)) do
      Alice.(title) ~> Bob.(title)
      Bob.get_price(title) ~> Alice.(price)

      # Alice has a 50% off coupon!
      if Alice.in_budget?(price / 2) do
        Alice.get_addr() ~> Bob.(addr)
        Bob.ship(title, addr)
      else
        Alice.("Too expensive!")
      end
    end
  end
end
```

Listing 1: A small choreography written using the current version of Chorex.

The body of the `defchor` macro consists of a list of function definitions—nothing more. There must be at least one function named `run` present; this is the main entry point of the choreography. Multiple functions of the same name are allowed, per Elixir's dispatching conventions, though caveats apply.

Functions can have one or more parameters. Parameters are *located at an actor*; `Alice.(title)` means that actor `Alice` will have access to a parameter `title` at runtime, whereas `Bob` will not be able to access that parameter. Expressions must also be located; in the above choreography, `Bob` computes `get_price(title)` and `ship(title, attr)` on his node. The expressions are *plain Elixir code* and get lowered directly into the modules that Chorex projects.

Lines like `Alice.(expr) ~> Bob.(var)` are called *delivery notation*, and this example indicates `Alice` computing `expr` locally and sending the result to `Bob` where it is stored locally in the variable `var`. This pairing of sends and receives in the syntax ensures that deadlocks do not occur.

Conditionals are notoriously tricky in choreographies [8]. Once an actor has computed which branch to take, how are the other actors to be notified of this? This is called the "knowledge of choice" (KoC) problem. By default, Chorex broadcasts the result of the conditional in an `if` branch to all other actors in the system. This set can be slimmed down by specifying the actors to notify directly (`if Alice.in_budget?(price/2), notify: [Bob]`). Chorex checks at compile time to ensure that all actors who need this information get it.

2.1.1. Chorex-specific features

One of the key challenges of distributed programming is that actors might not be reliable. No choreographic system prior to Chorex accounts for recovering from actor crashes. Chorex allows programmers to wrap

critical regions in a `checkpoint` block with an associated `rescue` block to run if an actor crashes during the duration of the `checkpoint`.

```
checkpoint do
  Alice.dangerous_op() ~> Bob.result
  Bob.report(result)
rescue
  Alice.safe_fallback() ~> Bob.safe_result
  Bob.report(safe_result)
end
```

Listing 2: A choreography where `Alice` might crash running `dangerous_op`.

In Listing 2, the actor filling the `Alice` role might crash when running `dangerous_op` locally. If this happens, `Alice` will be restarted and `Alice` and `Bob` will resume execution inside the `rescue` block.

2.1.2. Projecting step-by-step

Projection is intricate; what follows is a high-level overview meant to provide intuition about what Chorex does during macro expansion to project a choreography into runnable Elixir code. For additional details, see [5]. We will reference Listing 1 throughout.

Projecting a choreography for a given role is the same as answering, "what does this role need to do here?" Chorex works by iterating through the list of actors provided (`[Alice, Bob]`) and creating a module for each one that contains that actor's view of the choreography.

For `Alice`, Chorex creates a function `run(title)`, the body of which sends `title` to `Bob`, receives a message from `Bob` and stores it in `price`. Next is the conditional: since `Alice` is the one computing the conditional, the projection includes the call to `in_budget(price / 2)`. Projection proceeds similarly through each branch for `Alice`, and then through the whole program again for `Bob`. A rough projection for both actors of Listing 1 is shown in Listing 3.

`Alice`'s projection

```
defmodule Alice do
 def run(title) do
   send(Bob, title)
   price = receive, do: m -> m

   if in_budget(price / 2) do
     # communicate choice: yes buy
     send(Bob, true)
     send(Bob, get_addr())
   else
     # communicate choice: no buy
     send(Bob, false)
     "Too expensive!"
   end
 end
end
```

`Bob`'s projection

```
defmodule Bob do
 def run(_) do
   title = receive, do: m -> m
   send(Alice, get_price(title))

   # Get choice from Alice
   which_branch = receive, do: m -> m

   if which_branch do
     addr = receive, do: m -> m
     ship(title, addr)
   else
     nil
   end
 end
end
```

Listing 3: Rough projection for each actor of Listing 1.

This is very close to how early iterations of Chorex worked. Note how an `if` statement induces a `send` on the side of the decider (`Alice`) and a `receive` on the side of the other party. This is Chorex handling knowledge-of-choice. If there is a third party to the choreography, Chorex would project both branches and then attempt to unify them. If an actor's behavior is identical between branches, then there is no need for that actor to

receive the choice from `Alice`. If there is a difference, as there is in this example with `Bob`, then that actor must get choice information.

2.1.3. The subtlety of receiving from a GenServer

Chorex projects actors into GenServers[1] as opposed to the straight-line processes seen in Listing 3. This gives us two specific abilities that straight-line processes lack:

- **GenServers can handle out-of-band messages.** A high-priority message informing an actor of another actor's crash could come at any time, and this message needs to be handled before any other messages. A straight-line process would have to search its entire process mailbox for high-priority messages at *every* `receive`. In contrast, GenSevers simply register a handler for special messages.
- **GenServers can be supervised.** Any process on the BEAM can be monitored, but adding a process to a supervision tree requires that it be a GenSever. Chorex adds actors to a supervision tree to ensure clean process termination when the choreography exits.

GenServers have their own drawbacks, e.g., complicated variable scope, but Chorex works around these problems during macro expansion to ensure that GenServers track the right state and have the right callbacks. One particularly thorny issue that GenServers must deal with is that every receive must be anticipated with a callback function: any time an actor receives inside the body of a function, it must split that function into two pieces. Figure 2 shows a choreography where the actor `Bob` receives a message from `Alice`; `Bob`'s projection produces two functions: one for the first half of the choreography leading up to the receive, and second for receiving the message and subsequent calculations. To tie the control flow together, Chorex generates a unique token (`:tok1` in the example; in practice a UUID); `Bob`'s process pushes this token onto a stack when it is ready to receive. When the message arrives, this token gets popped off the stack to ensure control flows to the correct handler.

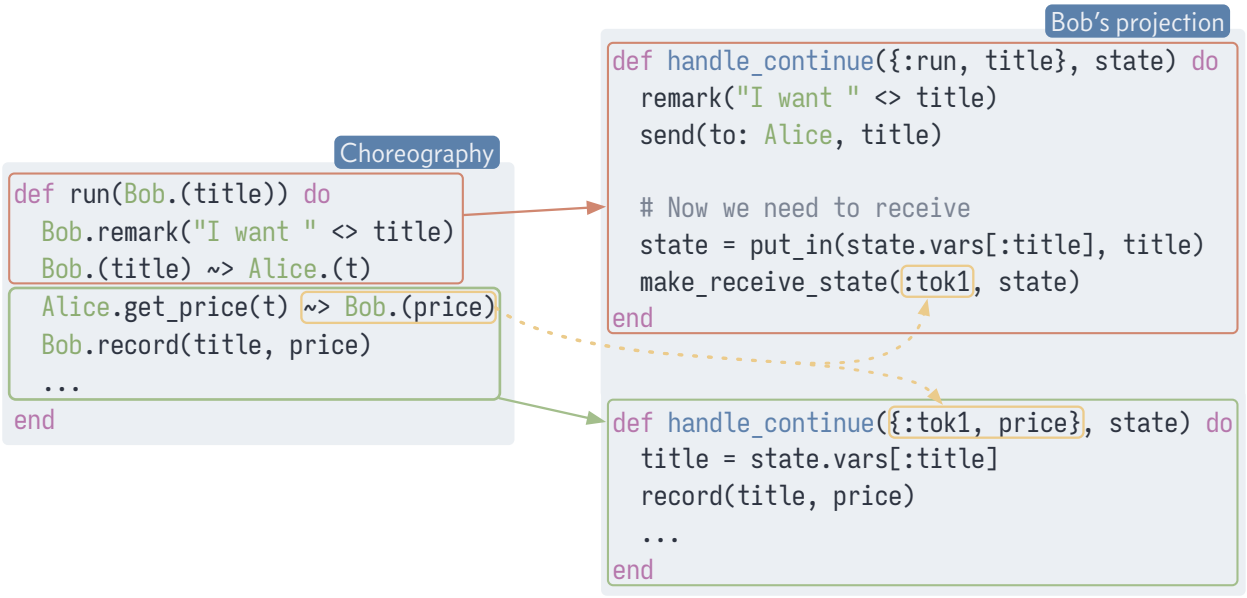


Figure 2: Receives in a choreography must be split across two functions inside the GenServer.

Actors keep the following inside their GenServer states:

- actor-local variables
- custom mailbox
- choreography call stack

[1] GenServers are a part of the Elixir standard library. See https://hexdocs.pm/elixir/GenServer.html

During projection, Chorex tracks the set of variables introduced in the choreography. Whenever an actor must anticipate a receive, these variables get saved to the GenServer's state in the code immediately before the receive. In the handler for the message, the variables get re-introduced so that local expressions can reference them.

Chorex uses the call stack stored in the GenServer's state to support function calls in the choreography. Since receives must be split across two callbacks, the native Elixir call stack cannot be used not support the call/return semantics in the choreography. Instead, an identifier like the token `:tok1` in Figure 2 gets pushed onto the call stack, and a new GenServer callback where the function should return to gets generated with a matching token. This continuation-style control flow management also allows Chorex to support `if` in non-tail position, something that Pirouette [9] is not able to do.

2.1.4. Process monitoring

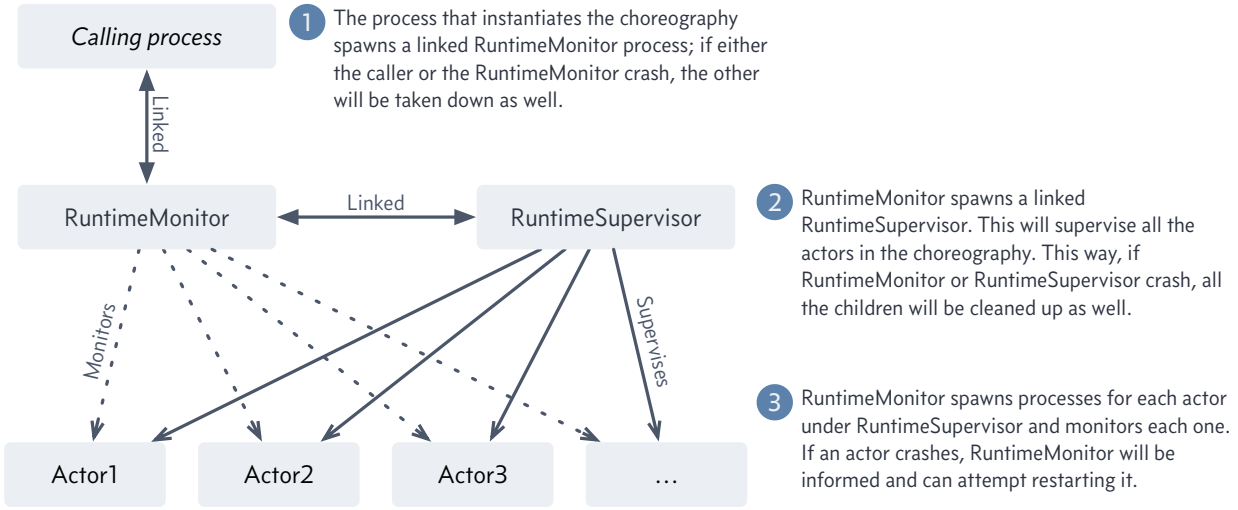


Figure 3: Process linking, monitoring, and supervising relationships in a running choreography.

Figure 3 shows what processes are involved when a choreography gets instantiated. The *calling process* is whatever process instantiates the choreography. Chorex starts up two auxiliary processes: a choreography monitor and a dynamic supervisor. The purpose of the monitor is to watch for actor crashes. When an actor goes down, it gets a message and reacts by restarting the actor, restoring its state, and notifying the other actors about the crash. The dynamic supervisor binds all the actors together with the monitor: if the monitor goes down, the dynamic supervisor ensures that the other processes get cleaned up as well.

2.2. Programming with Chorex

Chorex takes a description like on the right side of Figure 1 and produces code for each role in the system: Chorex creates modules for the `Client`, `App`, `Ident`, and `Audit` roles. These generated modules handle just the communication and control-flow aspects of the distributed system; details such as the `authorized?` or `report_results` functions on the `Ident` and `Client` roles respectively are handled by *implementing modules* for each of these roles.

An implementing module handles the specific local computation for each role. When it is time to run the choreography, each role gets associated with an implementation module to dispatch local computation to. This makes choreographies modular: for example, during testing, engineers might use an implementation for `Ident` that has a few mock credentials will well-defined roles. In production, the implementation for `Ident` would use whatever identity management system they use for their real clients. Figure 4 illustrates

the relationship between implementation modules (user-written), projected modules (macro-generated), and the choreography (user-written).

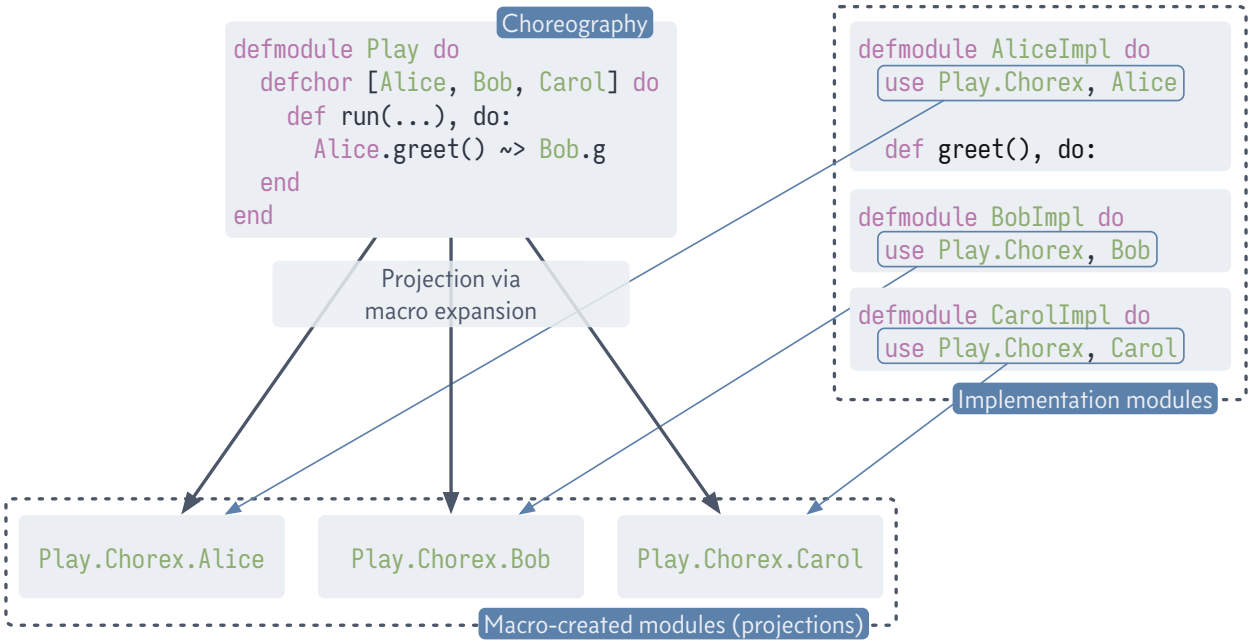


Figure 4: Projection of a choreography produces a module for each role; corresponding implementation modules handle the local computation.

3. The gap in our expressive power

To illustrate where choreographies are lacking in expressivity, consider the following setup as you might find in a typical application.

A client (`Client`) connects to the application server and gets assigned a worker process (`Worker`) to handle its requests. One of the actions the client may take is to request a report of some kind rendered as a PDF. This kind of job typically takes several seconds and requires multiple queries to the database (`Database`). This can easily be captured with a choreography.

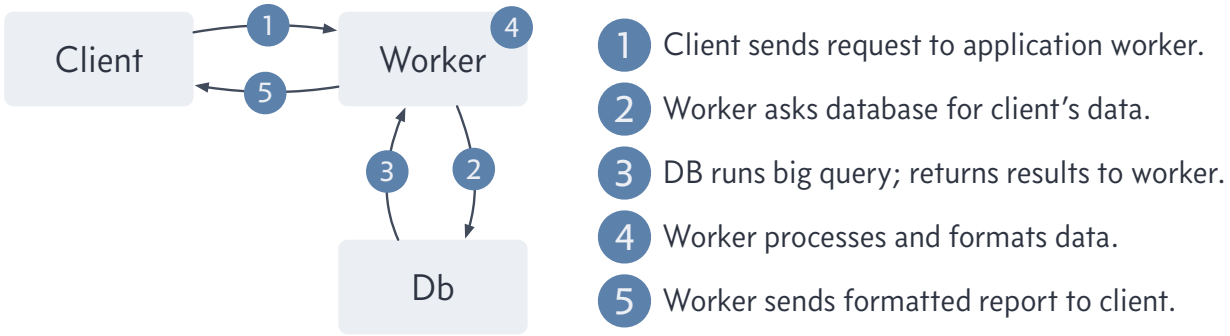


Figure 5: The happy path: a client sends a request and gets a response.

```
def make_report([c: Client, w: Worker, db: Database], c.id, c.report_type) do
  c.{id, report_type} ~> w.{client_id, report_type}
  w.client_id ~> db.client_id
  db.gather_data(client_id) ~> w.client_data       # Expensive!
```

```
  if w.(report_type == :pdf) do
    w.render_report(client_data) ~> c.full_report  # Expensive!
    c.print(full_report)
  else
    w.format_raw(client_data) ~> c.raw_report
    c.print(raw_report)
  end
end
```

Here is the twist: if the client disconnects while this job is running, terminating the job as soon as possible —rather than finishing and discarding the report—is preferable to save resources. This does not fit neatly into any existing choreographic language.

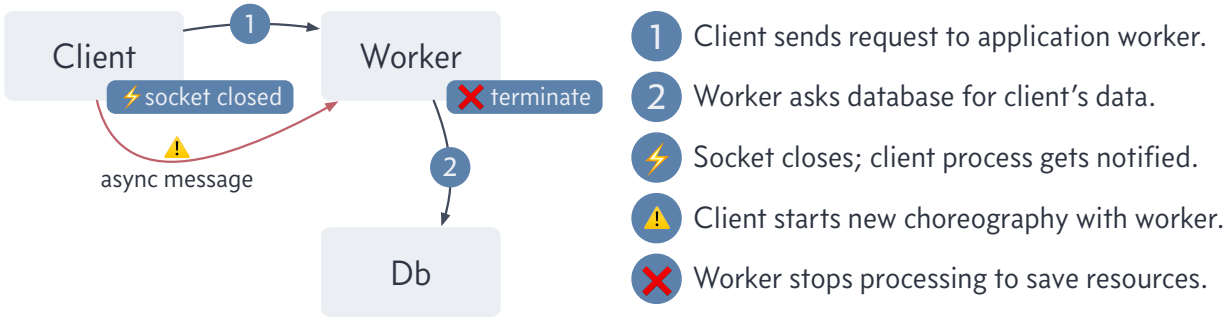


Figure 6: Connection to client closes; worker shuts down to conserve resources.

How would we like to model this? Presumably, the process responsible for managing the client connection could get notified if the socket is closed. We can think of this as an event occurring at the actor implementing the `Client` role. We could imagine how the handler choreography might look:

```
def suspend([c: Client, w: Worker], c.id) do
  c.id ~> w.client_id
  w.suspend_work(client_id)
  w.@terminate
end
```

The problems are twofold: first, we need to set up an event handler to run `suspend` when a process implementing the `Client` role receives `{:error, :closed}`. The second problem is that, in order to run this `suspend` choreography, we need a reference to both the `Client` and the corresponding `Worker` roles, but the event only occurs on the `Client` role. Moreover, there will likely be *many* processes at any given time filling the `Client` and the `Worker` roles. How to we ensure that we only terminate *this* client's corresponding worker?

To solve this, every role needs to track which other role instances it is aware of. A `Client` role should be aware of its corresponding `Worker` worker instance, `Worker` instances should know about their `Client`, as well as have a handle to the `Database` role. The `Database` role might not need to track long-term what the other roles are, so long as we are not concerned with events originating at the database. This could change depending on the situation, of course. We can declare this *role network* as we would declare a type with named fields:

```
role Client,
     worker: Worker

role Worker,
     client: Client,
```

```
    db: Database

role Database
```

When we get an event at an instance of a `Client` role, we can instantiate the `suspend` choreography with the right `Worker` instance:

```
event Client, {:error, :closed} do
  suspend([@self, @self.worker], @self.@state[:client_id])
end
```

Now we can express an *event-driven choreography* which previously was not possible. We also have another new thing: we have *first-class actor references*. Event-driven choreographies motivate first-class actor references because an actor, in order to respond to an event, needs to have a way to communicate with the other actors it is associated with. Explicit networks allow us to recover what other actors are associated with an actor that needs to handle an event.

3.1. Beyond events: creating new actors on-the-fly

Suppose that a `Client` does *not* first connect directly to a `Worker` role, but instead connects to an `Application` service, which creates an instance of `Worker` for each client. This is desirable because each client's requests can be isolated from others'. Creating new actors dynamically has until now been strictly *outside* the scope of choreographies; with first-class actor references and role networks, this becomes tractable and natural.

```
# … same role network as above …

def connect([c: Client, a: Application]) do
  c.@self ~> a.client_ref
  with a.fresh_worker <- a.@spawn(Worker, client: client_ref) do
    a.fresh_worker ~> c.my_worker
    c.@set!(:worker, my_worker) # @set! updates network
    make_report([c, a.fresh_worker, a.database], c.get_id(), c.get_report_type())
  end
end
```

Event-driven choreographies are a natural way to specify common patterns in distributed systems. To make event-driven choreographies feasible, roles need to have a network of other known roles, so that non-trivial choreographies can be instantiated in response to events which are located on just a single node. Once role networks enter the picture, the next natural step is to introduce first-class actor references and dynamic actor creation.

3.2. Bonus: mixed-lifetime choreographies

This new version of Chorex would be built to handle interfacing with long-running processes: to instantiate a choreography, the caller must provide PIDs for each actor. In the case of e.g. a PubSub system, the publisher role might be a freshly-created process and the broadcaster might be a long-running process found via the Elixir `Registry` system.

3.3. Research and implementation challenges

Choreographic systems have historically been oriented around the notion of a *session*. Sessions are self-contained bundles of actors that can all communicate with each other. Sessions can exist concurrently,

but the choreography does not mediate any inter-session communication. This is the model that the first version of Chorex is built around. To send a message, the role names are used directly in delivery notation. This makes projection simple; all the compiler has to do is see if the role under projection is on either side of the `~>` operator.

Event-driven choreographies must rely on actors keeping track of other related actors in order to function. But the first-class actor references that enable this take away the explicit role names that we have previously relied upon for performing projection—they are just variables now.

Another complication arises when an actor receiving an event attempts to fire off a choreography in response: how do the related actors know that it is time to engage in a choreography? The complications compound when considering creating new processes on-the-fly: how do actors know what other actors they are participating with in a choreography? Finally, what kinds of census primitives do we need?

This section goes into detail on the difficulties presented by these aspects.

3.3.1. Function calls and coordinating processes

When a choreography gets instantiated, the actors need to know how to reach the other actors in the system. In Chorex, this means that every actor needs the process ID (PID) of the other actors. The first version of Chorex managed this for the user once during setup, and a specialized runtime monitor alerted actors of the change in PID for any crashed actors.

In this new version, choreographies can be instantiated at any function in the choreography's body. Moreover, choreographies will be instantiated with *already running processes* that will not have references to other actors. This problem crops up a second time during the run of the choreography: suppose we have processes `a` and `b`, and `b` has a reference to another process in slot `c`.

```
some_func([a, b.c], …)
```

When `some_func` gets run, processes `a` and `b.c` will need to find each other so that in the function body they can talk to each other directly. We may need some coordinating process to act as an initial address book. For initial choreographic instantiation that could be the calling process; later junctions might need a named and registered process to find each other. It would be preferable to not have any centralized coordinating process; instead, process `b` might be responsible for forwarding process `c`'s reference to `a` and vice-versa.

3.3.2. Projection with first-class references: the need for a type checker

Consider this choreography:

```
def go([c: Client, s: Server]) do
  c.req() ~> s.request
  s.compute(request) ~> c.resp
  c.report(resp)
end
```

The variables `c` and `s` refer to `Client` and `Server` roles respectively. How are we to project this? In the current version of Chorex, the delivery notation would be `Client.req() ~> Server.request`; notice how the actor names are present *in the syntax*. During projection, Chorex simply examines if the actor under projection is involved in the message delivery using that local information. With actors as variables, however, this is no longer feasible.

We will need to treat the roles as a kind of type and propagate this information from the function header throughout the body. We will also need this information when using an actor's network—the `role` declarations will be important for this to work. The type checker will need to elaborate the choreography source

at every variable with the role information before projection runs; once the variables have role information attached to them, projection can rely on that to tease apart the respective roles' components.

As a bonus, a type checker can also help us get to a clean solution of the seminal knowledge-of-choice issue: by deriving the "type" of each branch (i.e. computing the set of involved actors) we can ensure that correct knowledge-of-choice broadcasting happens automatically. Functions will need to be annotated with the set of roles that they involve—which may be more than the roles found as part of the function signature. For example, a referenced role like `s.worker` would not appear in the parameters, but using that reference would mean that the `Worker` role would need to do some projection for this function. This information needs to be propagated up so that *callers* of the function know what roles are involved so that they can project `if` branches correctly. Role involvement is simple to compute: it is just the union of the set roles.

3.3.2.1. Thorny issues around types

There is another subtle issue noted by [10]: there is nothing to prevent a user from writing down a choreography with two separate but identical roles:

```
def tricky([c1: Client, c2: Client]), do: …
```

Two possible solutions for something like this:

1. Such choreographies are prohibited. This would not be a material disadvantage relative to exiting choreographic systems, and so such a restriction could be justified. This would likely be the easiest to implement and the simplest to understand.
2. A role can have two or more distinct identities. In the above example, an actor fulfilling the `Client` role would need to be able to support running `tricky` in *either* position, but there would be some distinction informing the actor which role it is functioning in.

Option 2 is the most appealing, as restricting a choreography to only permit one instance of a role in a choreography seems to be confining. There will be plenty of subtleties that arise with this, such as a case where the same actor gets called upon to play multiple role instances simultaneously, e.g. `tricky([actor1, actor1])`.

Another issue is that we might permit local expressions to return process references, e.g.:

```
with pool.worker <- pool.select_worker_process(), do: …
```

Type checking local Elixir expressions is out-of-scope for Chorex. We could solve this problem with an explicit annotation (syntax TBD) that says what kind of role reference we expect from a local expression, and then convert that into a runtime check to ensure that the reference is the role we wanted.

3.3.3. Census primitives

It is not yet clear exactly what census primitives will be needed. Bates et al. [11] have a few primitives such as `fanOut`, `fanIn`, etc. Section 4 lists nine new language primitives that the choreographies in Section 5 require; we will see if more or fewer are needed.

The key challenges will be ensuring that the new census primitives do not violate the cardinal property of choreographies: introducing a new primitive should not compromise deadlock freedom. Balancing this requirement with ergonomic and powerful primitives will require investigation.

3.3.4. Reusable pieces and new parts

The first version of Chorex will support and inform the construction of the new event-driven Chorex, even though many assumptions made in the first version will no longer apply in the second. The projection techniques will transfer in large part: e.g. Chorex's solution to projecting `if` forms in non-tail position will be the same. Chorex will continue to project to GenServers, and the runtime model used in those GenServers will serve as the basis for what comes next.

4. New elements for choreographic programming

It is a fact universally acknowledged that a paper on the subject of choreographies must be in want of a bookseller example. This section sketches what a revamped Chorex might look like to get to the expressive power we want. The syntax described here is *tentative* and subject to change as we discover better ways of expressing our intent.

The `defchor` macro will be our starting point to describing a choreography; everything within the purview of Chorex will go in this block.

```
defchor do

  # --- Role network description

  role Buyer                   # Roles declared explicitly

  role Seller,
    shipper: ShippingService    # Roles declare other roles they know about

  role ShippingService


  # --- Role initializers

  init Buyer(title) do          # Optional: initialization functions for roles
    {book_title, []}
  end

  init Seller(shipper) do
    {nil, [shipper: shipper]}
  end


  # --- Functions (i.e. choreographies)

  def buy([b: Buyer, s: Seller], b.max_price) do
    b.@get(@state) ~> s.book_title
    s.get_price(book_title) ~> b.price
    if b.(price <= max_price) do
      with b.arrival <- make_purchase([b, s, s.shipper], b.get_addr()) do
        b.report(arrival)
      end
    else
      b.report("Too expensive")
    end
  end

  def make_purchase([b: Buyer, s: Seller, sp: ShippingService], b.addr) do
```

```
    b.addr ~> s.addr
    s.addr ~> sp.addr
    sp.send_to(addr) ~> b.date
    b.date
  end
end
```

The `role` keyword introduces a new role name in the choreography. It is akin to a type definition. Role descriptions include the name of the role and, optionally, a keyword list enumerating the other roles this role is aware of. In the above example the `Seller` role is aware of an instance of a `ShippingService` role under the name `:shipper`. Role relations can be a single instance (`ShippingService`), a homogeneous list `[Workers]`, or a homogeneous map `%{TopicHandlers}`.[2]

Inside the choreography, when a variable `s` is bound to a `Seller`, you can reference the `ShippingService` with `s.shipper`. Only the instance `s` knows about `shipper`; `s` can send messages to `s.shipper`, but other processes cannot unless `s` first sends them the reference:

```
s.shipper ~> b.shipper_ref
b.get_addr() ~> b.shipper_ref.addr
```

The `init` keyword introduces an initializer for a role. These are *optional* and are used primarily (though not exclusively) when a new instance of that role gets `@spawn`ed inside the choreography. The internals are *not* a choreography, but a standard Elixir function. Inside this block, `@self` may be used to reference the actor being initialized. Initialization blocks must return a 2-tuple consisting of `{initial_state, kw_list_of_actors}`. The second value must correspond to the reference slots defined by `role`. Actors do not need to always be initialized this way; their networks can be updated later in the choreography.

A choreography is a function parameterized over different locations. The first parameter is a keyword list mapping variable names to role types. The remaining parameters are parameters to the choreography.

When you call a function, you must pass references to the actors you wish to use. Calling a function has the effect of informing participants about each other.

Other keywords not shown in this example:

- `for`

  A `for` comprehension used in conjunction with a list of references will run the body in parallel for all roles.

- `@get`, `@set!`, `@update!`

  Three special functions invoked on an actor that allow access to its network.

- `@spawn`, `@spawn_unlinked`

  Create a new process with a given role. The `unlinked` variant ensures that a child crashing will not take down the spawning process.

- `@monitor`

  Monitors another process. This allows actors to listen for other actors' crashes and respond appropriately. Crashes are reported as events.

- `@exit`

  Forces the actor to exit cleanly.

[2] We only specify the value; Chorex does not care what the type of the key is.

5. Evaluation plan

This section outlines some choreographies that Chorex should be able to run if properly equipped with the expressive tools in Section 3. The exact syntax here is subject to change.

Implementing these examples would represent a significant milestone in the expressive capabilities of choreographic programming: these are all examples with immediate real-world application. Some of these examples are well beyond the expressive power of current choreographies. Other choreographic systems have implemented similar systems, but Chorex would be able to describe them more succinctly and clearly.

5.1. Basic PubSub system

This is a simple PubSub system: processes can *subscribe* to topics and then get notified when another process *publishes* a message on that topic. For this, we use a single broadcaster process that keeps a map of topic → subscribers in its memory.

```
defmodule PubSub do
  defchor do
    role Subscriber

    role Publisher,
         backbone: Broadcaster

    role Broadcaster,
         subscribers: %{[Subscriber]}  # Chorex doesn't care about the key type

    init Publisher(broadcaster) do
      {nil, %{backbone: broadcaster}}
    end

    init Broadcaster() do
      {%{}, nil}
    end

    def subscribe([s: Subscriber, b: Broadcaster], s.topic) do
      s.@self ~> b.new_sub
      s.topic ~> b.the_topic
      b.@update!(:subscribers, fn subs ->
        the_list = Map.get(subs, the_topic, [])
        Map.put(subs, the_topic, [new_sub | the_list])
      end)
    end

    def publish([p: Publisher, b: Broadcaster], p.topic, p.msg) do
      p.{topic, msg} ~> b.{topic, msg}
      for b.s <- b.(get_in(@state, [:subscribers, topic])) do
        b.msg ~> b.s.msg
        b.s.report(msg)
      end
    end
  end
end
```

A tricky piece of this example is that during the `publish` choreography, the process reference returned from `b.(get_in(…))` will not be typed by our type checker. We will need to add an explicit cast annotation. See Section 3.3.2.1.

5.1.1. Using the choreography

Once the choreography is defined, we could use it in an application in the following way:

- The programmer would need implementation modules (Section 2.2) for the `Subcriber` and `Broadcaster` roles. The implementation for `Subscriber` in particular needs to implement the `report/1` function that the `publish` choreography relies upon.
- The Elixir application would specify to start the long-running `Broadcaster` process as part of the application supervision tree.
- A process can subscribe to the topic `"elvish sword"` with a call like the following:

```
PubSub.Chorex.subscribe([self(), Registry.lookup(MyApp.Registry, Backbone)],
  "elvish sword")
```

  The calling process would need to implement the `Subscriber` role; this would imply that it is a GenServer and that it has a function `report/1`.

- When another process wishes to broadcast something on a topic, it can do:

```
PubSub.Chorex.publish([self(), Registry.lookup(MyApp.Registry, Backbone)],
 "elvish sword",
 "your sword has begun to shine with a faint blue glow")
```

  At which point, any processes subscribed to this topic would get their `report` process called as if it has been invoked by:

```
report("your sword has begun to shine with a faint blue glow")
```

5.2. A higher-order choreography

Higher-order choreographies, pioneered by Pirouette [9], bring functional-flavored programming to choreographies. However, in Pirouette, functions—i.e. choreographies—were not truly first-class: they could not be stored in variables or data structures. Chorex aims to change that: in the choreography below, the actor `s` with role `Server` selects between `send_go` and `send_stop` choreographies and stores it in a local variable `s.func`.

```
defchor do
  role Client

  role Server,
       clients: [Client]

  def send_go([c: Client, s: Server]) do
    s.go() ~> c.x
    c.handle(x)
  end

  def send_stop([c: Client, s: Server]) do
    s.stop() ~> c.x
    c.handle(x)
  end
```

```
  def broadcast([s: Server], s.go?) do
    with s.func <- (if s.go?, do: &send_go/0, else: &send_stop/0) do
      for s.c <- s.clients do
        s.func.([s.c, s])
      end
    end
  end
end
```

This will be challenging to implement, and will necessitate similar coordination features to those in Section 3.3.1.

5.3. IRC implementation

IRC is a venerable internet standard. It also serves as the focus of one of the first large-scale evaluations of choreographic programming, implemented in Choral [4].

Matching the capabilities of Choral is an important baseline for capability; Chorex can do better by obviating the need for much of the connective code surrounding the choreographic pieces as well as providing a simpler and more general events-based workflow. This is because choreographies in Chorex can be written in an explicit event-driven style, and because choreographies can fully manage process lifecycles within the choreography. In this example, each room on an IRC server gets a dedicated process that handles message broadcasting; the choreography handles users joining rooms—none of this must be delegated to external code.

```
defchor do
  role Client,
       server: Server

  role Server,
       peers: [Server],
       clients: [Client]

  init Server(peer_servers) do
    # 1st element: initial actor state; access with @state
    # 2nd element: initial network
    {
      %{nicks: %{}, client_rooms: %{}},
      %{peers: peer_servers, clients: []}
    }
  end

  init Client do
    { {}, %{server: nil} }      # not connected initially
  end

  def login([c: Client, s: Server], c.username, c.passwd) do
    c.{username, passwd} ~> s.creds
    if s.valid?(creds) do
      # Update networks
      s.@update(:clients, fn cs -> [c | cs] end)
      c.@update(:server, fn _ -> s end)
      set_nick([c, s], c.get_nick())
    else
      c.report_invalid()
```

```
    end
  end

  def set_nick([c: Client, s: Server], c.nick) do
    c.nick ~> s.nick
    s.set_nick(c, nick)
  end

  # Event triggered by client's UI
  event Client c_join(room_name) do
    join([@self, @self.server], room_name)
  end

  def join([c: Client, s: Server], c.room) do
    c.room ~> s.t
    s.@update(@state, fn s -> update_in(s, [:client_rooms, c], fn rs -> [t | rs] end))
  end

  event Client c_send_msg(room, msg) do
    send_msg([@self, @self.server], room, msg)
  end

  def send_msg([c: Client, s: Server], c.room, c.msg) do
    c.{room, msg} ~> s.{room, msg}
    broadcast([s], s.room, s.msg, s.lookup_nick(@state, c))
  end

  def broadcast([s: Server], s.room, s.msg, s.nick) do
    for other <- s.clients do
      if s.in_room?(@state, room, other) do
        s.{nick, msg} ~> other.new_msg
        other.notify(new_msg)
      end
    end

    # FIXME: avoid communication loops
    for p <- s.peers do
      s.{room, msg, client_nick(c)} ~> p.{room, msg, nick}
      broadcast([p], p.room, p.msg, p.nick)
    end
  end
end
```

One advantage Choral will have over Chorex is that it was able to use IRC wire format to exchange messages. Chorex relies on CIV tokens [12] in its messages; it might be possible to recover full compatibility with the IRC wire spec with appropriate messaging proxies.

5.4. Key-value store à la MultiChor

This example is based off of the key-value store presented in the MultiChor paper [11].

```
defchor do
  role Client

  role Primary,
```

```
      replicas: [Replica]

role Replica,
      primary: Primary

init Primary(n_replicas) do
  {
    %{},
    %{
      replicas: for _i <- 1..n_replicas, do: @spawn(Replica, [%{}])
    }
  }
end

init Replica(parent, init_store) do
  # could monitor parent here to take over in case of crash
  {
    init_store
      %{
        primary: parent
      }
  }
end

def request([c: Client, p: Primary], c.req) do
  c.req ~> p.client_req

  case p.client_req do
    {:put, key, value} ->
      handle_put([p], key, value)

    {:get, key} ->
      handle_get([p], key)
  end
  |> ~> c.resp

  c.report(resp)
end

def handle_put([p: Primary], p.key, p.value) do
  for p.r <- p.replicas do
    p.{key, value} ~> p.r.{k, v}
    p.r.update_store!(k, v) ~> p.ack
  end

  p.update_store!(key, value)
after
  resync([p, p.replicas])
end

def handle_get([p: Primary], p.key) do
  p.get_store(key)
end

def resync([p: Primary, rs: [Replica]]) do
  with p.hashes <- (for r <- rs, do: r.hash_state()) do
    if p.(Enum.all_equal?(hashes)) do
```

```
        :ok
      else
        for r <- rs do
          p.get_store() ~> r.new_store
          r.set_store!(new_store)
        end
      end
    end
  end
end
```

5.5. Web server

This example demonstrates how one might build a web server with a choreography. For this, we're using an architecture similar to Elixir's ThousandIsland library [13].

Figure 7 shows how this architecture works. There are two choreographies at play: the first choreography negotiates what happens when a client first connects to the server, and the second choreography handles how the client–server interaction proceeds. In the first choreography, the `Listener` process listens on a port and wait for a client to knock. Once a connection is present, the `Listener` process pulls an `Acceptor` off of its pool of `Acceptors` round-robin style and goes back to listening for connections. The `Acceptor` process creates a new `Client` and `AppHandler` roles to manage the client–server interaction, and it spawns a *new* choreography and puts the two new processes into it.[3] The `@round_robin` function is a convenience accessor to return the first element of a list of references while rotating the list; this could be implemented in terms of `@get` and `update`.

The `Client` and the `AppWorker` processes monitor each other: this way, if the `Client` goes down (e.g. because the socket closed), the `AppWorker` can clean up any state if it wants do; if the `AppServer` goes down, the `Client` gets a chance to send a 500 error across the web socket.

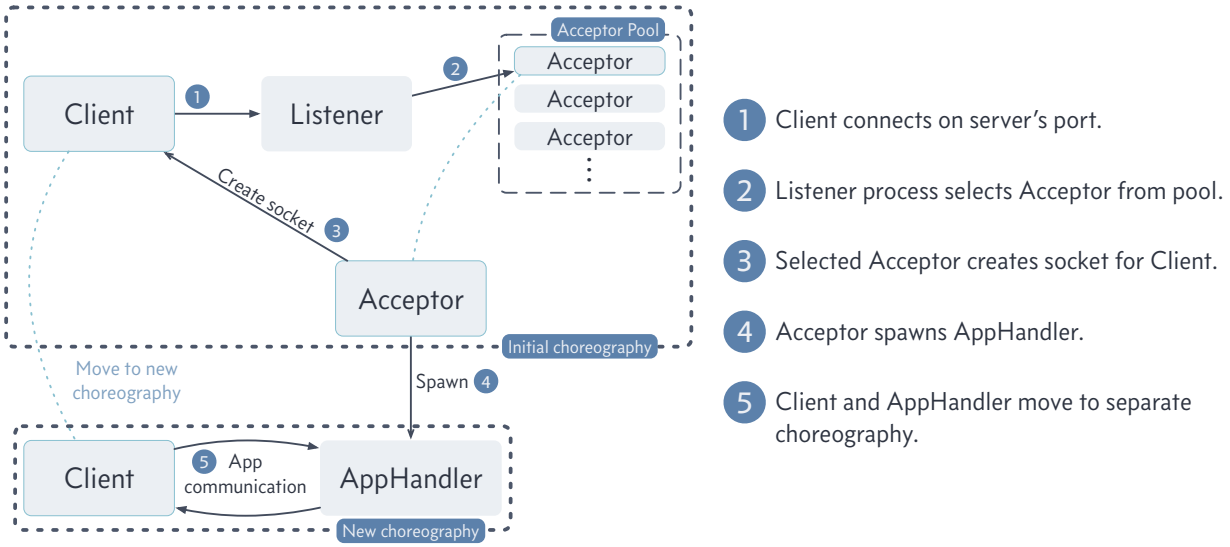


Figure 7: Two-tier choreography for a web server: the first tier handles the client connection and building a socket. The second tier is for the client–server interaction.

[3] The `Listener` process *could* handle all the socket setup itself, but handing this off to another process means that there's lower latency for fresh clients trying to connect.

This example is a good milestone because the first iteration of Chorex also implements a web server. However, it is a lot more clumsy with its handling of multiple choreographies: the *implementation* of the socket acceptors have to manually invoke `Chorex.start` to kick off the application choreography. With the new model, a choreography can kick of another one simply by invoking a function and instantiating it with the right members.

```
defchor do
  role Client,
       worker: AppWorker

  role Listener,
       acceptor_pool: [Acceptor]

  role Acceptor

  role AppWorker,
       client: Client

  init Client(socket) do
    {socket, []}
  end

  init AppWorker(client) do
    @monitor(client)
    {nil, [client: client]}
  end

  init Listener(listen_config, pool_size) do
    pool = for 0..pool_size, do: @spawn(Acceptor, [])
    {listen_config, [acceptor_pool: pool]}
  end

  # Wait for a connection, then let a member of the acceptor pool
  # handle the task of standing up app handler processes
  def listen([l: Listener], l.listen_socket) do
    with l.next_acceptor <- l.@round_robin(:acceptor_pool) do
      l.accept_connection(listen_socket) ~> l.next_acceptor.conn
      setup_application([l.next_acceptor], l.next_acceptor.conn)
      listen([l], l.listen_socket)
    end
  end

  # Acceptor creates client and app server processes
  def setup_application([a: Acceptor], a.conn) do
    with a.client <- @spawn_unlinked(Client, a.conn),
         a.worker <- @spawn_unlinked(AppWorker, a.client) do
      a.worker ~> a.client.worker
      a.client.@set!(:worker, worker)
      a.client.@monitor(worker)
      run_application([a.client, a.worker])
    end
  end

  # Main application loop req → response → repeat
  def run_application([c: Client, w: AppWorker]) do
    c.request() ~> w.req
```

```
    with w.{new_state, resp, close?} <- w.app(@state, req) do
      w.@set!(@state, new_state)
      w.resp ~> c.resp
      c.render(resp)

      if w.close? do
        c.close_conn()
      else
        run_application([c, w])
      end
    end
  end

  # Application server crashed
  event Client, {:DOWN, app_worker, _, _} do
    # Check that we're getting this message from a downed worker
    if app_worker ==  @self.worker do
      @self.report_500()
      @self.close_conn()
      @exit
    end
  end

  # Client disconnected suddenly; shutdown app
  event AppWorker, {:DOWN, client, _, _} do
    if client == @self.client do
      @exit
    end
  end
end
```

5.6. High-performance PubSub system

This PubSub system is similar to the one in Section 5.1, but the `Backbone` process delegates broadcasting messages to special `TopicServer` processes, which track lists of their subscribers. The role of the `Backbone` process is to maintain a mapping of topic to `TopicServer` instance, and to spin up new `TopicServer` instances as needed.

```
defchor do
  # Role specs
  role Backbone,
       topics: %{TopicServer}

  role Publisher,
       backbone: Backbone

  role Subscriber

  role TopicServer,
       upstream: Backbone,
       clients: [Subscriber]

  # Role initializers
  init Publisher(backbone) do
    {nil, [backbone: backbone]}
```

```
end

init Backbone() do
  {nil, %{}}
end

init TopicServer(backbone, topic) do
  {topic, %{upstream: backbone, clients: []}}
end

# Choreographies

# Invoke like:
# PubSub.subscribe([self(), Registry.lookup(Backbone)], :cookies_ready)
def subscribe([s: Subscriber, b: Backbone], s.topic) do
  s.topic ~> b.t
  s.@self ~> b.subscriber_ref
  assoc_topic([b], b.t, b.subscriber_ref)
end

def assoc_topic([b: Backbone], b.topic, b.subscriber_ref) do
  # Client ref gets passed around but not used; shouldn't need to
  # project anything for this function.
  with b.maybe_ts <- b.@get(:topics, topic, nil) do
    if b.maybe_ts do
      b.subscriber_ref ~> b.maybe_ts.sr
      b.maybe_ts.@update(:clients, fn cs -> [sr | cs] end)
      b.maybe_ts.@monitor(cs) # subscribe to crashes
    else
      with b.ts <- b.@spawn(TopicServer, [@self, topic]) do
        b.subscriber_ref ~> b.ts.sr
        b.ts.@update(:clients, fn cs -> [sr | cs] end)
        b.ts.@monitor(cs)

        b.@update(:topics, fn ts -> Map.put(ts, topic, ts) end)
      end
    end
  end
end

def send_msg([p: Publisher], p.topic, p.msg) do
  p.{topic, msg} ~> p.backbone.{t, m}
  with p.backbone.ts <- p.backbone.@get(:topics, t) do
    if p.backbone.ts do
      p.backbone.m ~> p.backbone.ts.m
      broadcast([p.backbone.ts], p.backbone.ts.m)
    else
      p.error("Unknown topic #{topic}")
    end
  end
end

def broadcast([ts: TopicServer], ts.msg) do
  for ts.s <- ts.subscribers do
    ts.msg ~> ts.s.msg
    ts.s.report(msg)
  end
```

```
  end

  event TopicServer {:DOWN, pid, _, _} do
    @self.@update(:subscribers, fn cs -> Enum.drop(cs, pid) end)
    @self.("Lost process #{pid} on topic #{@self.@get(@state)}") ~> @self.upstream.msg
    @self.upstream.log(msg)
  end
end
```

6. Landscape of related work

There are two branches of work related to event-driven choreographies: existing choreographic programming systems and multi-party session types. Both describe distributed systems from a top-down perspective and project that description into components for each actor. Choreographies can be be described by multi-party session types. [14] This section will look at both approaches to global-view programming for distributed systems.

Within multi-party session types and choreographic programming, there is a further distinction between work that focuses on the theory and work that focuses on pragmatic application. Both fields enjoy a large body of work establishing the theoretical foundations of their respective systems. Choreographic programming has somewhat more practical application than multi-party session types; Chorex is primarily focused on building a practical CP system. Developing a metatheory is explicitly out-of-scope for Chorex: we wish to examine the gap between bringing some of these theoretical systems into reality.

6.1. Theory

6.1.1. Multi-party session types

Multi-party session types (MPST) [15] are a tool for specifying and verifying distributed systems. Like choreographic programming, a MPST is a global view of the distributed system with all interactions between parties described in a top-down fashion. The global type can then be projected into types for each of the participants. MPSTs are just a type system: they are useful for verifying that existing systems comply with the overall protocol. *Building* a system from scratch, on the other hand, puts a large burden on the implementer: the global type can guide the programmer in creating a program to satisfy the local type, but it lacks the ease-of-use that choreographic programming enjoys from automatically constructing an implementation from the specification. Since Honda et al.'s paper in 2016, there have been many other papers on MPSTs. This related works section will only cover those that I have found to be relevant to my dissertation.

Viering et al. [16] introduce a MPST system that introduces `try/catch` into the type system. This is similar to the original version of Chorex. A follow-up paper by some of the same authors [17] adds a more general monitoring system that will be a little closer to what the next version Chorex aims to accomplish with monitoring: a process can monitor any other process, and the system allows for participant replacement. They use their system to build a session-typed version of the Apache Spark data analytics framework. One difference is that, since they do not enjoy the guarantees that the BEAM provides—specifically, perfect detection of crashed processes—they have a more general notion of failure detection relying on timeouts. This paper has some good example programs that are generally interesting in the distributed programming space.

6.1.2. Choreographies

Choreographies began as a W3C specification for describing protocols on the web. [18] The intent was that a choreography could describe an interaction between multiple distributed components, and then some tools could check for *compliance* to the specification. Achieving and demonstrating compliance is difficult; mechanically extracting a compliant implementation from a specification, on the other hand, is far more feasible. In his doctoral dissertation [8], Montesi showed the correspondence of choreographies and the π-calculus and introduced *Chor*—the first choreographic language designed for programming in directly.

Chorex took a lot of inspiration from Pirouette [9], which was the first functional, typed, higher-order choreographic theory paper; Hirsch et al. developed a model for higher-order choreographies in a typed functional language and proved in Rocq that projection was deadlock-free. As a follow-up to Pirouette, λQC from [19] extends Pirouette with actor references. Like Pirouette, λQC is a *theory* paper and does not have an accompanying implementation. In λQC, when selecting an actor from a pool, all actors in the pool need to get notified of the selection, as a form of knowledge-of-choice broadcasting. Chorex might be able to obviate the need for this communication. Chorex took inspiration for its syntax and endpoint projection strategy from Pirouette, especially with regards to the projection of conditionals.

6.2. Practice

The flagship implementation for choreographic programming is Choral [3], an object-oriented choreography compiler that targets Java. The biggest real-world use of Choral to date has been an implementation of IRC [4]. This IRC system solved a problem reminiscent of event-driven programming: they have a special `Events` library which allows actors in the choreography to handle messages arriving at either end of a communication channel. The library enqueues events until the local choreography is ready to handle them. Event handlers are written as a choreography with both parties: sending a message in the choreography gets translated into enqueuing a message on the other participant's event queue. While this works well enough for the IRC implementation, there is still a great deal of glue code that lives outside the choreography, and it is not clear how to generalize from the two-party interface the `Events` library provides into a fully general event-driven choreography.

MultiChor [11] is a Haskell library for choreographic programming that features *census polymorphism*: the ability to abstract over the number of participants in a choreography. They include constructs to distribute a value to a set of processes and gather values from a set back down to a single process. However, they do not have constructs to generate new participants on-the-fly.

Choral and MultiChor represent opposite ends of the implementation spectrum. Choral is a stand-alone compiler and using Choral in a larger Java project requires a separate build step to perform endpoint projection to various `.jar` files from the Choral source code. MultiChor, on the other hand, is a Haskell library that performs endpoint projection at runtime and does not require a separate build step. MultiChor's forerunner HasChor [6] was similar in this regard and suffered from efficiency issues: HasChor had to broadcast knowledge-of-choice to *all* participants in the choreography to ensure soundness. MultiChor overcomes these features is a relatively language-agnostic way, and they make implementations for Haskell, Rust, and TypeScript.

Chorex sits near the middle: by using Elixir's macro system to perform endpoint projection at compile time, we get the benefits of a separate build step to analyze the choreography and ensure sound knowledge-of-choice broadcasting. At the same time, we avoid the pitfalls of Choral and enjoy the library-only advantages that MultiChor et al. do: macros are built into Elixir, so the user of Chorex does not have to go through a separate build step. Chorex is not the only CP system to perform endpoint projection through macros. Klor [20] and Choret [21] provide choreographic programming libraries for Clojure and Racket, respectively.

7. Timeline

2025 December Proposal defense
2026 January Start building new Chorex compiler
2026 July Finish compiler
2026 November Submit PLDI paper
2026 December Begin dissertation writing
2027 January *Panic*
2027 October Dissertation defense

Bibliography

[1] F. Montesi, *Introduction to Choreographies*, 1st ed. Cambridge University Press, 2023. doi: 10.1017/9781108981491.

[2] M. Felleisen *et al.*, "The Racket Manifesto," *LIPIcs, Volume 32, SNAPL 2015*, vol. 32, pp. 113–128, 2015, doi: 10.4230/LIPICS.SNAPL.2015.113.

[3] S. Giallorenzo, F. Montesi, and M. Peressotti, "Choral: Object-oriented Choreographic Programming," *ACM Trans. Program. Lang. Syst.*, vol. 46, no. 1, pp. 1–59, Mar. 2024, doi: 10.1145/3632398.

[4] L. Lugović and F. Montesi, "Real-World Choreographic Programming: Full-Duplex Asynchrony and Interoperability," *Programming*, vol. 8, no. 2, p. 8, Oct. 2023, doi: 10.22152/programming-journal.org/2024/8/8.

[5] A. Wiersdorf and B. Greenman, "Chorex: Restartable, Language-Integrated Choreographies," *Programming*, vol. 10, no. 3, Oct. 2025, doi: 10.22152/programming-journal.org/2025/10/20.

[6] G. Shen, S. Kashiwa, and L. Kuper, "HasChor: Functional Choreographic Programming for All (Functional Pearl)," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, pp. 541–565, Aug. 2023, doi: 10.1145/3607849.

[7] A. Wiersdorf, S. Chang, M. Felleisen, and B. Greenman, "Type Tailoring," *LIPIcs, Volume 313, ECOOP 2024*, vol. 313, pp. 44:1–44:27, 2024, doi: 10.4230/LIPICS.ECOOP.2024.44.

[8] F. Montesi, "Choreographic Programming," phdthesis, 2013. [Online]. Available: https://www.fabriziomontesi.com/files/choreographic-programming.pdf

[9] A. K. Hirsch and D. Garg, "Pirouette: Higher-Order Typed Functional Choreographies," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–27, Jan. 2022, doi: 10.1145/3498684.

[10] E. Graversen, A. K. Hirsch, and F. Montesi, "Alice or Bob?: Process Polymorphism in Choreographies," *J. Funct. Prog.*, vol. 34, p. e1, 2024, doi: 10.1017/S0956796823000114.

[11] M. Bates, S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. P. Near, "Efficient, Portable, Census-Polymorphic Choreographic Programming." Accessed: Jan. 23, 2025. [Online]. Available: http://arxiv.org/abs/2412.02107

[12] D. Plyukhin, M. Peressotti, and F. Montesi, "Ozone: Fully Out-of-Order Choreographies," *LIPIcs, Volume 313, ECOOP 2024*, vol. 313, pp. 31:1–31:28, 2024, doi: 10.4230/LIPICS.ECOOP.2024.31.

[13] M. Trudel, "ThousandIsland." Accessed: Oct. 13, 2025. [Online]. Available: https://github.com/mtrudel/thousand_island

[14] M. Carbone and F. Montesi, "Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, in POPL '13. New York, NY, USA: Association for Computing Machinery, Jan. 2013, pp. 263–274. doi: 10.1145/2429069.2429101.

[15] K. Honda, N. Yoshida, and M. Carbone, "Multiparty Asynchronous Session Types," *J. ACM*, vol. 63, no. 1, pp. 1–67, Mar. 2016, doi: 10.1145/2827695.

[16] M. Viering, T.-C. Chen, P. Eugster, R. Hu, and L. Ziarek, "A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems," in *Programming Languages and Systems*, A. Ahmed, Ed., Cham: Springer International Publishing, 2018, pp. 799–826. doi: 10.1007/978-3-319-89884-1_28.

[17] M. Viering, R. Hu, P. Eugster, and L. Ziarek, "A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 124:1–124:30, Oct. 2021, doi: 10.1145/3485501.

[18] "Web Services Choreography Description Language Version 1.0." Accessed: Nov. 04, 2025. [Online]. Available: https://www.w3.org/TR/ws-cdl-10/

[19] A. Samuelson, A. K. Hirsch, and E. Cecchetti, "Choreographic Quick Changes: First-Class Location (Set) Polymorphism," 2025, doi: 10.48550/ARXIV.2506.10913.

[20] L. Lugović and S.-S. Jongmans, "Lovrosdu/Klor." Accessed: Nov. 04, 2025. [Online]. Available: https://github.com/lovrosdu/klor

[21] A. Bohosian and A. K. Hirsch, "Choreographies as Macros," *Electron. Proc. Theor. Comput. Sci.*, vol. 420, pp. 12–21, May 2025, doi: 10.4204/EPTCS.420.2.