Programming Microservice Choreographies: a security use case

Saverio Giallorenzo[1], Fabrizio Montesi[1], Marco Peressotti[1], and Luisa Zeppelin[2]

[1] University of Southern Denmark
[2] University of Hamburg

1 Introduction

To be effective, microservices typically coordinate with one another by following *choreographies*, i.e., coordination plans based on message passing [1, 6, 4]. Choreographic Programming is an emerging paradigm for the productive and correct implementation of choreographies, where choreographies are specified as software artifacts from a global viewpoint, and then a compiler automatically translates them to sets of compliant endpoint implementations [3]. In this setting, *compliance* means that when all endpoints are run together, they interact exactly as defined by the initial choreographies.

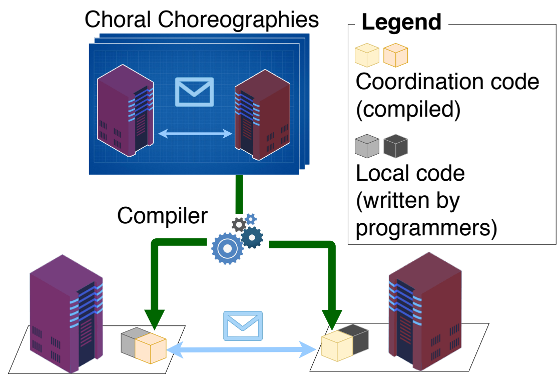

Figure 1: The Choral compiler generates compliant-by-construction coordination libraries (yellow boxes) for each microservice involved in a choreography. The implementor of each microservice can then combine its respective coordination library with the local implementation of the core functionalities of the microservice (gray boxes).

The Choral language (choral-lang.org) has been recently proposed as the first choreographic programming language that can be adopted in the mainstream [2] (Figure 1). In Choral, choreographies are written in an extension of Java where objects can be collaboratively implemented by multiple roles (the participants of the choreography), and then a Java library that implements each role is automatically generated (in the future, Choral will support different target languages). These Java libraries can then be used in the implementation of a microservice system, to ensure that all microservices will communicate correctly, i.e., accordingly to the choreographies that have been agreed upon.

In this presentation, we will give a brief overview of the paradigm of choreographic programming and its incarnation in Choral. Then, we will illustrate how Choral can be applied to the programming of microservices in practice, by exploring an implementation of a security protocol—a multiparty distributed authentication protocol.

2 Choral and the Example

The key idea of Choral is to extend Java's data types to *role parameters*. Thanks to this extension, a choreography can be represented as an object in Choral.

As a simple example to grasp the basics of Choral, consider the following class `HelloRoles`, which defines a choreography for two roles `A` and `B`.

Choral Code

```
1 class HelloRoles@(A, B) {
2   public static void run(SymChannel@(A,B)<String> channel) {
3     String@A a = "Hello from A"@A;
4     String@B b = channel.com(a);
5     System@B.out.println(b);  }}
```

Class `HelloRoles` is parameterised over the roles `A` and `B`, denoted by the notation `@(A, B)`. The class defines a simple choreography in method `run`, which takes as parameter a bidirectional channel between the two roles. Notice how, differently from Java, each variable and string literal is located at a role by the `@`-notation. Line 3 assigns the string `"Hello from A"` located at `A` (`"Hello from A"@A`) to variable `a` of type "String at `A`" (`String@A`). Then, Line 4 uses the communication method (`com`) of `channel` to transfer the string in `a` to `B`, which stores it in variable `b`. In Line 5, `B` prints the received value.

The distributed authentication protocol that we will present is inspired by OpenID [5]. In the protocol, an `IP` ("Identity Provider") authenticates a `Client` to access a third-party `Service`. We can codify the protocol as the Choral class below. The syntax `expr >> o::m` is a shorthand for `o.m(expr)` (Choral borrows the forward chaining operator from F#).

Choral Code

```
1  public class DistAuth@(Client, Service, IP){
2    private TLSChannel@(Client, IP)<Object> ch_Client_IP;
3    private TLSChannel@(Service, IP)<Object> ch_Service_IP;
4    public DistAuth(...) { ... } // omitted
5    private static String@Client calcHash(String@Client salt, String@Client pwd) { ... } //omitted
6
7    public AuthResult@(Client, Service) authenticate(Credentials@Client credentials) {
8      String@Client salt = credentials.username
9       >> ch_Client_IP::<String>com >> ClientRegistry@IP::getSalt >> ch_Client_IP::<String>com;
10     Boolean@IP valid = calcHash(salt, credentials.password)
11      >> ch_Client_IP::<String>com >> ClientRegistry@IP::check;
12     if (valid) {
13       /* IP sends an authentication token to both Client and Service */
14     } else {
15       /* IP sends a failure message to both Client and Service */
16     }
17   } }
```

Method `authenticate` (lines 7–17) is the entry point and consists of three phases. In the first phase, lines 8–9, the `Client` communicates its `username` to `IP`, which `IP` uses to retrieve the corresponding salt in its local database `ClientRegistry`; the salt is then sent back to `Client`. The second phase (lines 10–11) resolves the authentication challenge: `Client` computes its hash with the received salt and its locally-stored password, and sends this to `IP`; `IP` then checks whether the received hash is valid, storing this information in its local variable `valid`. The result of the check is a `Boolean` stored in the `valid` variable located at `IP`. In the third phase (lines 12–16), `IP` decides whether the authentication was successful or not by checking `valid`. In both cases, `IP` informs the `Client` and the `Service` of its decision. In case of success, `IP` sends to the others an authentication token that they can use for further interactions (we omit the code for creating and sending the token).

For more details, the interested reader can consult the Choral website, where a full version of this example is also given: https://choral-lang.org.

References

[1] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[2] S. Giallorenzo, F. Montesi, and M. Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020.

[3] F. Montesi. *Choreographic Programming.* Ph.D. Thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

[4] Object Management Group. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/, 2011.

[5] OpenID Foundation. OpenID Specification. https://openid.net/developers/specs/, 2014.

[6] W3C. WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2004.