

SEArch: an execution infrastructure for service-based software systems^{*}

Carlos G. Lopez Pombo ^{**1,2}[\[0000-0002-0248-5019\]](mailto:cglopezpombo@unrn.edu.ar), Pablo Montepagano³, and Emilio Tuosto⁴[\[0000-0002-7032-3281\]](mailto:emilio.tuosto@gssi.it)

¹ Centro Interdisciplinario de Telecomunicaciones, Electrónica, Computación y Ciencia Aplicada - CITECCA, Universidad Nacional de Río Negro - Sede Andina.

² Consejo Nacional de Investigaciones Científicas y Técnicas - CONICET.

cglopezpombo@unrn.edu.ar

³ Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.

pmontepagano@dc.uba.ar

⁴ Gran Sasso Science Institute

emilio.tuosto@gssi.it

Abstract. The shift from monolithic applications to composition of distributed software initiated in the early twentieth, is based on the vision of *software-as-service*. This vision, found in many technologies such as RESTful APIs, advocates globally available services cooperating through an infrastructure providing (access to) distributed computational resources. Choreographies can support this vision by abstracting away local computation and rendering interoperability with message-passing: cooperation is achieved by sending and receiving messages. Following this choreographic paradigm, we develop SEArch, after Service Execution Architecture, a language-independent execution infrastructure capable of performing transparent dynamic reconfiguration of software artefacts. Choreographic mechanisms are used in SEArch to specify interoperability contracts, thus providing the support needed for automatic discovery and binding of services at runtime.

1 Introduction

In the past two decades the paradigm generally known as *Service-Oriented Computing* (SOC) has become predominant in software development. This paradigm comprises many variants such as —among others— cloud computing, fog and edge computing, and many forms of distributed computing associated with what is known as the *Internet of Things*.⁵ Key to service-oriented computing is the pos-

^{*} The authors want to thank Ignacio Vissani for his indispensable contributions to the design of SEArch.

^{**} On leave from Instituto de Ciencias de la computación CONICET-UBA and Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.

⁵ Although the terminology has evolved, SOC is still widely used to denote these software systems.

sibility of dynamically search and combine distributed computational resources exposed as services interacting over an existing communication infrastructure. This vision of software systems can be found in applied technologies, such as RESTful APIs, and has put forward what is commonly know as the API Economy. In this context many aspects of software development are facilitated, but service integration becomes non-trivial. A common interaction mechanism in SOC is (some form of remote) procedure call (for instance using JSON-RPC or via the HTTP protocol) since it resembles the typical function or procedure call of programming languages.

Choreographic approaches [1] propose an alternative interaction mechanism by conceptually separating the local computations of the components from their communication aspects. Under this approach, interoperability is understood at a more abstract level decoupled from any computational aspect. Within choreographic approaches we found the subclass of message-passing systems, a type of system where cooperation is achieved by the simple actions of sending and receiving messages through existing communication channels.

Asynchronous Relational Networks (ARNs) [2] yield a formalisation of the elements of an interface theory for service-oriented software architectures. More precisely, ARNs are a formal orchestration model based on hypergraphs whose hyperedges are interpreted either as processes or as communication channels. The nodes (or points) that are only adjacent to process hyperedges are called *provides-points*, while those adjacent only to communication hyperedges are called *requires-points*: the former constitute the interface through which a service exports its functionality while the latter yields the interface through which an activity expects certain service to provide a functionality. In the operational semantics of ARNs given in [3] actions performed by a component can dynamically trigger an automatic and transparent process of discovery and binding of a compliant service. The composition of ARNs (i.e., how binding is viewed from a formal perspective) is obtained by “fusing” provides-points and requires-points, subject to a certain compliance check between the contract associated to them. Later, [4] used *communicating finite state machines* (CFSM) [5] as a formal language for determining service interoperability automatically.

More recently, [6] has proposed data-aware CFSMs, an extension of CFSMs with *assertions*, namely first-order formulae associated to the communication actions. Besides, [6] has introduced a bisimulation relation for data-aware CFSMs and implemented an algorithm for checking bisimilarity of data-aware CFSMs. In this setting, given participants A and B, and a first order formula $\alpha(x)$, where x is a free variable, an action $AB!y\langle v \rangle \mid \alpha(x)$ is interpreted as: participant A sends to participant B a message of type y with value v guaranteeing that $\alpha(v)$ holds. Dually, $AB?y\langle v \rangle \mid \alpha(x)$ is interpreted as participant A receives from participant B a message of type y with value v assuming that $\alpha(v)$ holds. The rationale behind data-aware CFSMs is that assertions act as functional contracts predicating over the data exchanged by the components that participate in the communication.

In this work we introduce the language-independent execution infrastructure [SEArch](#), after Service Execution Architecture, based on the operational seman-

tics given to ARNs and the interoperability and functional compliance criterion supported by data-aware CFSMs. In particular, we give the architecture of SEArch (Section 2) and its main implementation details (Section 3). Also, we showcase SEArch on an on-line business cart (Section 4). In 5 we draw some conclusions and discuss further lanes of research.

2 A conceptual view of SEArch

There is a wide range of service-oriented architectures (SOAs) dictating design principles for SOC, each one with its own idiosyncrasy [7,8,9,10]. We embrace those that hinge on three main concepts: a *service provider*, a *service client*, and a *service broker*. The latter handles a *service repository*, a catalogue of service descriptions searched for in order to discover services required at runtime. In fact, the service broker is instrumental to the *discovery* of services according to a contract and of their *binding*, the composition mechanism that permits to “glue” services together at runtime as advocated by some SOAs.

To support SOC, SEArch offers a mechanism for populating registries and composing service-based application. Registering a service is, in principle, very simple: the service provider sends the service broker a request for registering a service attaching a (signed) package containing the contract and the unique resource identifier (URI) of the provided service.

The execution process of a service-based system in SEArch is significantly more complex. Figure 1 depicts the workflow. When launched, a component registers their communication channels to its *middleware*, each of which has its corresponding contract formalised as a set of data-aware CFSMs, one for each requires-point. This is required because the middleware has to mediate the communication with other components. In fact, when the component a running application, say C , tries to interact with another component, the middleware C , say M , captures the attempt and checks whether the communication session for that communication channel has been created. If no such session exists, the dynamic reconfiguration process triggers as follows:

1. M sends the service broker the contract of the communication channel;
2. for each data-aware CFSM R in the contract, the service broker queries the service repository for candidates;
3. the service repository returns a list candidates in the form of $\langle Pr, u \rangle$, where Pr is a data-aware CFSM and u is the URI of the service⁶;
4. the service broker checks whether the provision contract Pr is bisimilar to the requirement contract R ;
5. once the service broker has found services satisfying all the requirement contracts, it returns the set of URIs to M ;

⁶ SEArch is parametric in the implementation of the service repository so we assume it is not capable of checking compliance using behavioural contracts. We only rely on its capability of returning a list of candidates, obtained by using potentially more efficient and less precise criteria, for example, an ontology.

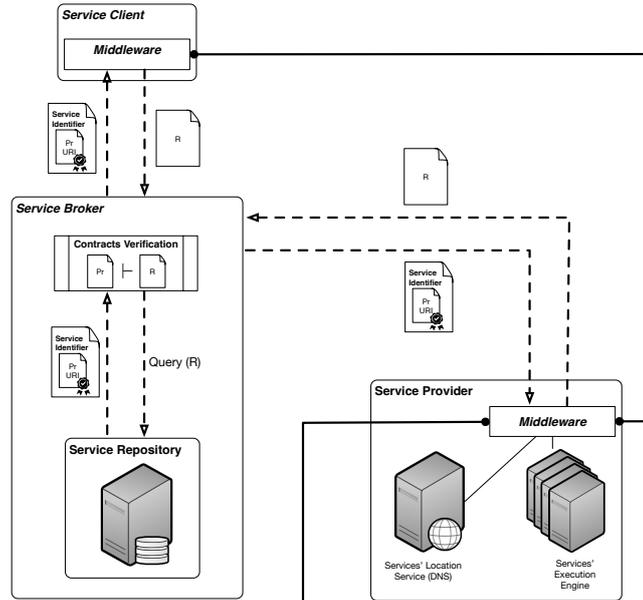


Fig. 1. Service execution procedure in SEArch

6. M opens a communication with the service middleware of each service returned by the provider requiring the execution of an instance of the corresponding service.

Then, M sends to or receives from the service middleware of the partner component the actual message; and the execution process proceeds. Notice that the execution of the service might also have its own requirements putting it as the originator a new dynamic reconfiguration.

The schematic view discussed before establish several requisites over the implementation of middleware and the service broker. We organise the discussion by considering these elements and their role in the execution architecture:

The middleware provides a private and a public interfaces. The former implements functionalities accessible by service clients and service providers. The public interface implements the capabilities needed for interacting with service brokers and other service middlewares.

The private interface consists of:

- **RegisterApp** to register a service and expose it in the execution infrastructure. This functionality opens a bidirectional (low level) communication channel with the service middleware that will remain open in order to support the (high level) communication with other services.
- **RegisterChannel** to register communication channels expressing requirements. This functionality provides the middleware with the relevant information for triggering the reconfiguration of the system and managing the

communications. The functionality can be used by any software artefact running in the host, regardless if it a service or client application.

- `AppSend` / `AppRecv` to communicate with partner components
- `CloseChannel` to close a communication channel.

The reader should note the asymmetry between the existence of a function for explicitly closing a communication sessions, and the lack of one for opening it. The reason for this asymmetry resides in that, on the one hand, transparency in the dynamic reconfiguration of the system is a key feature of SEArch but, on the other hand, it is in general not possible to determine whether a communication session will be used in the future.

The public interface consists of:

- `InitChannel` to accept the initiation of a point-to-point (low-level) communication channel. This operation allows the service broker to initiate the communication infrastructure that will connect the service executing behind the service middleware to the other participants in a communication session being setup.
- `StartChannel` to receive notification about point-to-point communication channels. This operation formally notifies the service middleware that the brokerage of participants according to a communication channel description was successful and the communication session has been properly setup.
- `MessageExchange` to exchange messages between service middlewares.

Figure 2 shows the application infrastructure and how buffers are used to provide point-to-point communication with external services. Within the infrastructure, it is possible to identify the structural design of the middleware.

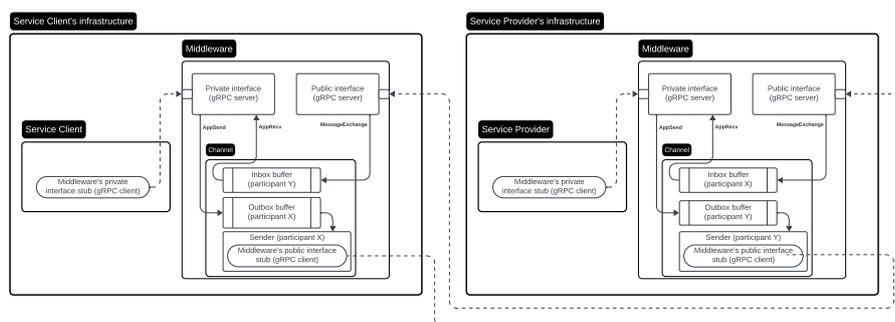


Fig. 2. Structural design of the point-to-point communication between a service client and a service provider.

The **service broker** exposes only two functionalities in a public interface:

- **BrokerChannel** to issue requests for brokerage. This operation allows a service middleware to request for the brokerage of communication channel, and the subsequent creation of a communication session over which the chosen services will communicate.
- **RegisterProvider** to issue requests for the registration of a service provider. This operation is the external counterpart of the functionality **RegisterApp** through which the service middleware provides the service providers the possibility of being offered as services available in the execution infrastructure.

Figure 3 shows the sequence diagrams offering a high level view of the processes of registration of service to the service broker.

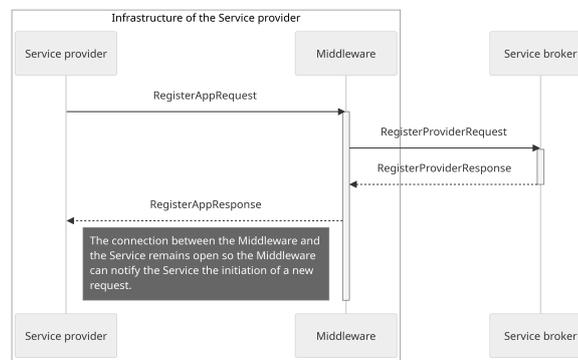


Fig. 3. Sequence diagram of the process of registration of a service.

Figure 4 shows the process of brokerage of a communication channel given the interfaces of the middleware and the service broker, detailed above in this section.

The process of brokering a communication channel for building a session (cf. Fig. 4) is significantly more complex. The service client uses the communication channel in the message **AppSend** (step ③). Concurrently, the service middleware begins the brokerage process by sending the contract to the service broker (step ④) and queues the message while acknowledging the service client with a message of type **AppSendRespond** (step ⑤). If the service client has to receive a message (**AppRecv**), the middleware captures the attempt triggering the brokerage and going through the same process for initiating the communication session. In this case, the service client will remain blocked until the expected message arrives.

The service broker, upon receiving the contract, queries the service repository for candidates and executes the compliance checks. Each compliance check can

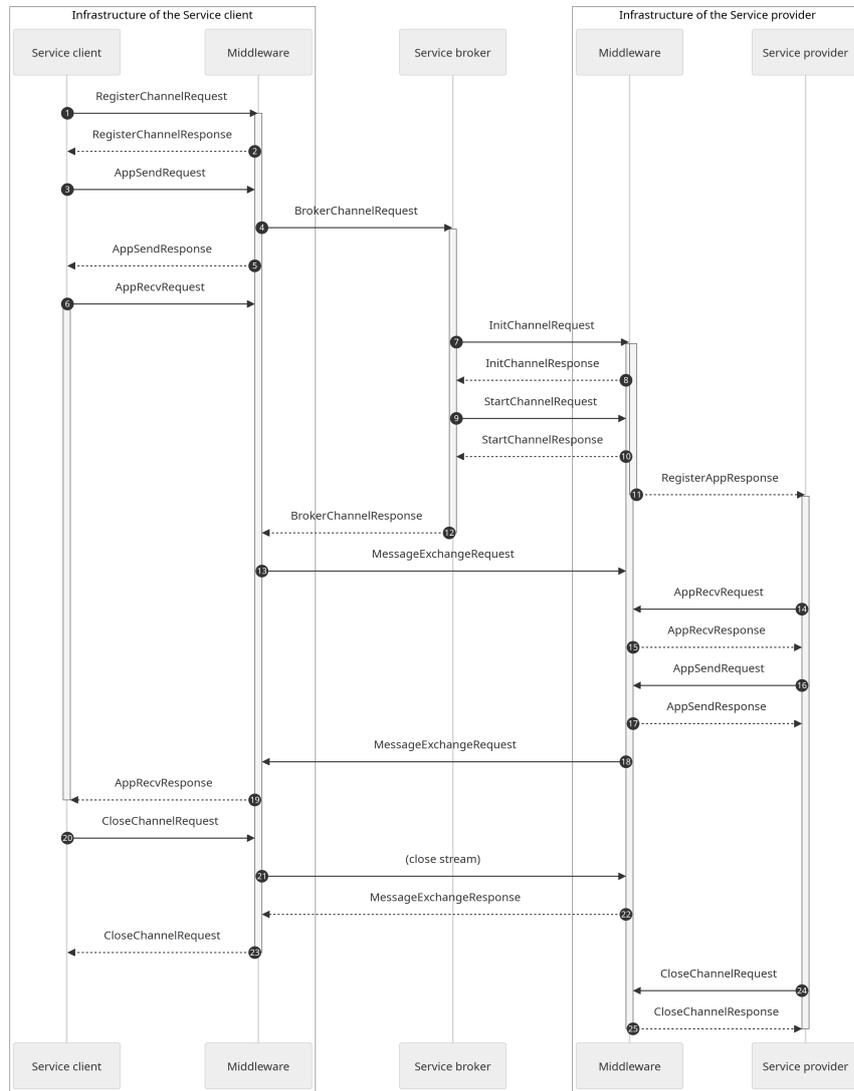


Fig. 4. Sequence diagram of the process of brokering a communication channel.

be too costly so the service broker implements a cache for storing precomputed positive results.

After choosing concrete providers for the participants in the contract, the service broker performs two successive rounds of messages with the chosen service provider. In the first round, a message of type `InitChannelRequest` is sent (step 7) to tell the service middlewares that a communication session involving its service provider is being initiated; the message also contains the URIs of all

the other participants in the communication session. At the same time, this message allows the service broker to verify that the provider is indeed online. Upon reception of this message, a service middleware must accept incoming messages for this channel and queue them for the eventual reception by the service provider. If all the service providers respond successfully with a message of type `InitChannelResponse` (step 8), then the service broker performs a second round with a messages of type `StartChannelRequest` (step 9), to confirm that the communication session has been initiated.

After receiving both initialization messages, the service middleware sends the service provider a message of type `RegisterAppResponse` (step 11) containing the UUID of the new communication session. Then, each service provider can start communicating over this session according to their contracts. Once a session is initiated, the service middlewares establish unidirectional streams with each other to send messages. In Fig. 4 the service middleware of the service client opens a stream with the other service middleware by sending a message of type `MessageExchange` (step 13). After the service provider has received the message (steps 14 - 15), it sends a message (steps 16 - 17) forcing the service middleware of the service provider to establishes a stream in the opposite direction (step 18).

Finally, the service client can close the channel by sending a message of type `CloseChannelRequest` (step 20) to its service middleware, closing the stream used to communicate with the service middleware of the service provider.

3 Implementation

The main objective of `SEArch`⁷ is to provide transparent integration of services offering abstractions for the interoperability of heterogeneous software artefacts. This is mainly achieved by the middleware featured by `SEArch`, which mediates all the interactions between software components. To this end, we implemented the lower layer of the communication infrastructure over `gRPC`⁸, `Protocol Buffers`⁹. The former is a high-performance RPC framework offering an easy and scalable solution to the problem of microservices integration; the latter is a typed and structured data packet serialisation format, used as an interface description language, which provide a high-level solution for system level communication.¹⁰ Both `gRPC` and `Protocol Buffers` aim to provide a general, yet easy to use, tool for developing communication infrastructure between systems. For this reason, there are compilers that interpret message and service definitions from `Protocol Buffer .proto` files, and generate code in a wide variety of

⁷ Available at <https://github.com/pmontepagano/search>.

⁸ Available at <https://grpc.io>.

⁹ Available at <https://protobuf.dev/>.

¹⁰ Although widely used, HTTP is not an ideal option for `SEArch` due to two severe limitations: HTTP supports request-response and it has no native support for typed messages (schemas).

programming languages for manipulating those messages with native classes and types.¹¹

We implemented SEArch in Go [11], a language with native support for channel-based concurrency (goroutines) and gRPC. Such flexibility is of particular relevance to us as it provides a high degree of portability, specially in order to cope with the heterogeneity of the computational resources available, that could be integrated to the SEArch ecosystem.

As said, we adopt CFSMs to model contracts for interoperability; more specifically, we use an implementation in Go of CFSMs¹² which we extended with a bisimilarity test. Test for bisimilarity is used by SEArch service brokers as interoperability compliance criterion when selecting service providers. There exists extensions of the CFSMs enabling their use for describing functional aspects of participants [6] and quality-of-service non-functional aspect [12], but they were not yet been added to `nickng/cfsm` so, for the sake of this presentation, the compliance check will only reflect interoperability. Choosing Go as a programming language was key for building a solution satisfying the most important hypothesis of the computational model; the order of messages is preserved. This hypothesis is vital to the correctness of a message-passing communicating systems, thus it becomes an important constraint of the implementation. Channels in Go are similar to Unix pipes, being thread-safe FIFO queues. This, together with the use of an RPC of type stream, allowed for the implementation of inbox and outbox buffers for each participant, together with a sender routine in charge of processing the Outbox; also providing a simple implementation for `MessageExchange` and `AppRecv` which essentially act as enqueue and dequeue operations, respectively.

In general, testing bisimilarity of CFSMs is computationally costly, resulting in a bottleneck. To tackle this problem the service broker features a cache associating lists of compliant services to requirement contract. The implementation consists of a very simple schema shown in Fig. 5 done as an *object-relational mapping* (ORM) in `ent`¹³, a simple and powerful entity framework specifically designed for Go. Whenever a requirement contract produces a miss in the cache table, or when the service repository returns candidates that have not been checked, the compliance checks are performed concurrently by launching separate goroutines¹⁴. For the moment, the concurrent execution of compliance checks only profits from parallelism locally (multi-core and multi-CPU servers) but it provides no support for multi-server architectures like clusters, cloud computing, etc.

¹¹ Until July of 2023 gRPC and Protocol Buffers officially supports C# / .NET, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python and Ruby among others; <https://grpc.io/docs/languages/>.

¹² Available at <https://github.com/nickng/cfsm>.

¹³ Available at <https://entgo.io>.

¹⁴ The library `conc` (available at <https://github.com/sourcegraph/conc>) was used to prevent leaks (routines that execute indefinitely).

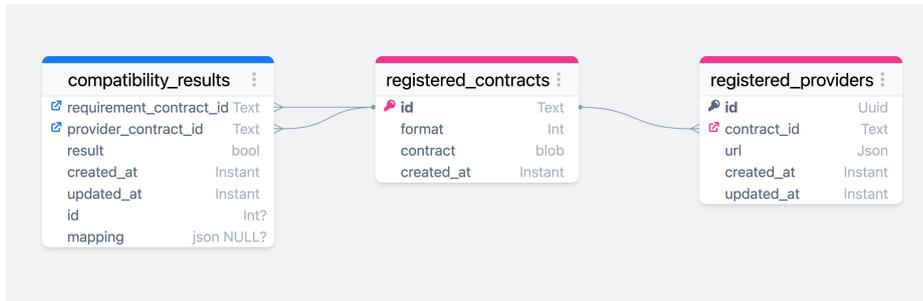


Fig. 5. ORM schema definition for the cache.

4 An online credit card payment service

This section shows a case study where an online shop relies on a third party payment service. The application involves three participants: a client (developed in Java), the seller (a backend server developed in Python), and a payment service (developed in Go). Figure 6 shows an abstract view of the three components where contracts are represented by the gray boxes (The source code of these components can be found at <https://github.com/pmontepagano/search/examples/credit-card-payments>). The rotated V-shape box repre-

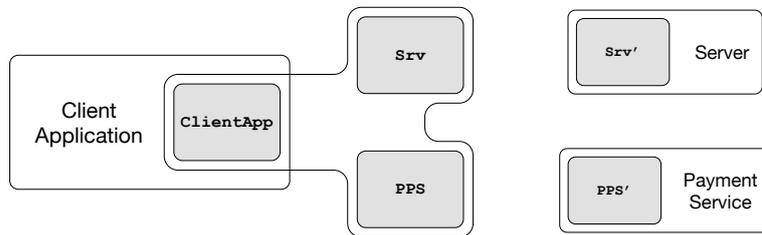


Fig. 6. A view of Client (left), seller (top right), and payment service (bottom right)

sents the client’s communication channel specified by the contracts `ClientApp`, `Srv`, and `PPS`, where the latter two are to be interpreted as requirements to be fulfilled by other participants. The contracts of seller and payment service are to be interpreted as provisions by the corresponding services.

We now detail each component.

4.1 Client application

As said, the client component is implemented in Java. We focus on how channel registration and communication is rendered in the client.

The communication channel of the client consists of the CFSMs in Fig. 7. This specification dictates that `ClientApp` starts by sending a `PurchaseRequest`

```

1  .outputs ClientApp
2  .state graph
3  q0 Srv ! PurchaseRequest q1
4  q1 Srv ? TotalAmount q2
5  q2 PPS ! CardDetailsWithTotalAmount q3
6  q3 PPS ? PaymentNonce q4
7  q4 Srv ! PurchaseWithPaymentNonce q5
8  q5 Srv ? PurchaseOK q6
9  q5 Srv ? PurchaseFail q7
10 .marking q0
11 .end
12
13 .outputs Srv
14 .state graph
15 q0 ClientApp ? PurchaseRequest q1
16 q1 ClientApp ! TotalAmount q2
17 q2 ClientApp ? PurchaseWithPaymentNonce q3
18 q3 PPS ! RequestChargeWithNonce q4
19 q4 PPS ? ChargeOK q5
20 q4 PPS ? ChargeFail q6
21 q5 ClientApp ! PurchaseOK q7
22 q6 ClientApp ! PurchaseFail q8
23 .marking q0
24 .end
25
26 .outputs PPS
27 .state graph
28 q0 ClientApp ? CardDetailsWithTotalAmount q1
29 q1 ClientApp ! PaymentNonce q2
30 q2 Srv ? RequestChargeWithNonce q3
31 q3 Srv ! ChargeOK q4
32 q3 Srv ! ChargeFail q5
33 .marking q0
34 .end

```

Fig. 7. Communication channel specification

to the seller (`Srv`) and, after receiving the `TotalAmount` from the seller, the client sends `CardDetailsWithTotalAmount` to the payment server (`PPS`); then the purchase is completed as shown in Fig. 7. The other CFSMs behave accordingly.

Below we report the snippet for the creation of the communication channel between the client application and its middleware's private interface by resorting the Java stubs generated by Protocol Buffer.

```

1  import io.grpc.ManagedChannelBuilder;
2  import ar.com.montepagano.search.v1.PrivateMiddlewareServiceGrpc;
3
4  // Get the stub to communicate with the middleware
5  ManagedChannel channel = ManagedChannelBuilder.forTarget(
6  "middleware-client:11000").usePlaintext().build();
7  PrivateMiddlewareServiceGrpc.PrivateMiddlewareServiceBlockingStub stub =
8  PrivateMiddlewareServiceGrpc.newBlockingStub(channel);

```

```

1 RegisterChannelRequest request = RegisterChannelRequest.newBuilder().setRequirementsContract(
2   contract).build();
3 RegisterChannelResponse response = stub.registerChannel(request);
4 var channelId = response.getChannelId();

```

Relying on the snippet above, the next one shows how the client registers the communication channel to the middleware: The object `contract`, passed to the function `SetRequirementsContract`, is an instance of the Java class `GlobalContract` (automatically generated from Protocol Buffer type), built from the specification in Fig. 7.

The snippet shown below, exhibits the client application invoking the middleware's operation `AppSendRequest` for sending a message `PurchaseRequest` (see line 3 of Fig. 7), to participant `Srv`, over the channel identified by `channelId` (the channel identifier is received at the moment of the registration of communication channel, see line 6 of channel registration snippet). Analogously, the

```

1 // Send PurchaseRequest with each item quantities and the shipping address
2 Map<String, Integer> items = new HashMap<>();
3 for (int selectedBook : selectedBooks) {
4   String bookTitle = bookTitles[selectedBook - 1];
5   if (items.containsKey(bookTitle)) {
6     items.put(bookTitle, items.get(bookTitle) + 1);
7   } else {
8     items.put(bookTitle, 1);
9   }
10 }
11 var body = ByteString.copyFromUtf8(String.format(
12   "{\nitems\": %s, \nshippingAddress\": \"%s\n\"",
13   gson.toJson(items), shippingAddress));
14 var msg = AppMessage.newBuilder().setType("PurchaseRequest").setBody(body).build();
15 var sendreq = AppSendRequest.newBuilder().setChannelId(
16   channelId).setRecipient("Srv").setMessage(msg).build();
17 var sendresp = stub.appSend(sendreq);
18 if (sendresp.getResult() != Middleware.AppSendResponse.Result.RESULT_OK) {
19   System.out.println("Error sending PurchaseRequest. Exiting...");
20   System.exit(1);
21 }

```

snippet below shows the client application invoking the middleware's operation `AppRecvRequest` for receiving a message `TotalAmount` (see line 4 of Fig. 7), from participant `Srv`, also over the channel identified by `channelId`.

```

1 var recvreq = Middleware.AppRecvRequest.newBuilder().setChannelId(
2   channelId).setParticipant("Srv").build();
3 var recvresp = stub.appRecv(recvreq);
4 if (!recvresp.getMessage().getType().equals("TotalAmount")) {
5   System.out.println("Error receiving TotalAmount. Exiting...");
6   System.exit(1);
7 }
8 var total_amount = gson.fromJson(recvresp.getMessage().getBody().toStringUtf8(), double.class);
9 System.out.println("Total amount: " + total_amount);

```

```

1 from lib.search import v1 as search
2
3 async def main(grpc_channel):
4     stub = search.PrivateMiddlewareServiceStub(grpc_channel)
5     registered = False
6     logger.info("Connected to middleware. Waiting for registration...")
7     async for r in stub.register_app(
8         search.RegisterAppRequest(
9             provider_contract=search.LocalContract(
10                format=search.LocalContractFormat.LOCAL_CONTRACT_FORMAT_FSA,
11                contract=PROVIDER_CONTRACT,
12            )
13        ):
14    ):
15        if registered and r.notification:
16            logger.info(f"Notification received: {r.notification}")
17            # Start a new session for this channel.
18            asyncio.create_task(session(grpc_channel, r.notification))
19        elif not registered and r.app_id:
20            # This should only happen once, in the first iteration.
21            registered = True
22            logger.info(f"App registered with id {r.app_id}")
23        else:
24            logger.error(f"Unexpected response: {r}. Exiting.")
25            break
26
27     grpc_channel.close()

```

4.2 Required services

The seller's server application was developed in Python; the snippet shown below exhibits the procedure for registering the backend server through a gRPC channel received as parameter.

The process starts on line 4 by opening a communication channel through the middleware's private interface, by resorting the Python stubs generated by Protocol Buffer. Then, the middleware's operation `RegisterAppRequest` is invoked with a local contract (i.e., a CFSM described as a single finite state machine like the ones shown Fig. 7). The result of the process depends on whether the service has been already registered or not; if the service has been properly registered by the broker, the latter will return an application identifier for the middleware to be able to refer to the service in its local host. Line 18 shows the asynchronous invocation to the function `session` which implements the server.

We omit the details of the Go implementation of the payment service because they are analogous to the seller implementation.

5 Conclusions and future work

In this work we combined well established languages and tools from the fields of service-oriented architectures, language semantics, and behavioural types to develop `SEArch`. The execution infrastructure of `SEArch` hinges as ARNs and has a complete operational semantics that enables analysis based on LTL formulae (see [3, Sec 4]). Behavioural types, more specifically CFSMs, were used as interoperability contracts that can be automatically analysed, thus providing the

means for checking service compliance with respect to a requirement contract. This last feature provides an answer to the problem of automatic service discovery at runtime. A careful selection of tools allowed us to implement a middleware and a service broker that jointly provide transparent creation and deletion of communication sessions, according to high-level behavioural contracts. This yields a general dynamic reconfiguration mechanism for this type of service-based software artefacts.

The infrastructure of **SEArch** is in a functional prototype stage and, consequently, the implementation left space for many extensions. Some of them are related to different aspects of scalability, for example, the current implementation features a service repository coupled to the implementation of the service broker. An alternative, and more scalable, design might implement the repository as a separate agent, allowing horizontal scaling of the role and separating the registration process from the broker service. This might also enable the broker to access multiple repositories. Another hurdle for scalability resides the centralised implementation of the compliance check in the service broker; separating the analysis of contracts as a service used by the broker might allow implementation over clusters of computers that might even perform off-line checks for precomputing the content of the cache we proposed, and implemented, to make the brokering more efficient.

From Fig. 1 it is easy to observe that choosing a service relies on abstract notions of provision and requirement contract. As we mentioned in Section 1, CFSMs has been extended with both, functional information and quality-of-service non-functional information making QoS-enriched Data-aware CFSMs a type of contract fit for describing many aspects relevant for both, service compliance and service selection. We plan to implement both extensions, and their associated verification algorithms, within **ChorGram** [13,14].

The compliance mechanism featured by **SEArch** relies on bisimilarity of CFSMs. An interesting line of work is to embed in **SEArch** other compliance mechanisms based on different types of contracts, and their associated tools. Some options are tools like **CAT** [15] which is based on *contract automata* [16,17,18] or contract-oriented middlewares like the one in [19,20] which supports timed behavioural types or the one in [21].

Recently tools for inferring behavioural specifications from code have been proposed. For instance, **KmcLib** [22] extracts CFSMs from Ocaml code, the tool in [23] infers behavioural types from Java code, and **ChorEr** [24] extracts choreography automata [25] from Erlang code, **Contractor**¹⁵ is a tool that extracts automata models from C programs. Composing these tools with **SEArch** opens the possibility of smoothly integrate services to our infrastructure.

Lastly, the correct execution of software system in **SEArch** requires that the implementation of service providers honours the contract they expose. This may not hold for incorrect implementations or in adversarial settings where malicious providers could be present. In this work we adopted an approach in which the act of invoking the middleware's operation `RegisterAppRequest`,

¹⁵ Available at <http://lafhis.dc.uba.ar/dependex/contractor/Welcome.html>.

and consequently triggering the invocation of the service broker's operation `RegisterProviderRequest` makes the provider fully responsible for any inconsistency that might occur during execution. However, SEArch does not provide any mechanism for assigning the blame. Detecting these types of violations can be attained statically (e.g., with approaches like the one in [26,27] or using behavioural contracts to synthesise a program skeletons which, after the implementation, could be statically analysed by tools akin to Dafny [28]. However, runtime verification is required when code cannot be analysed (e.g., third-party components). In particular, one can use monitors capable of auditing the communication session, according to the global contract specifying the communication channel. Under this view, it is paramount to provide analysis tools for the development of services in order to ensure compliance between the implementation and the provision contract. Extending SEArch for runtime verification is scope for future work.

References

1. World Wide Web Consortium: Web services description language (wsdl) version 2.0 part 1: Core language. On-line Available at <https://www.w3.org/TR/wsdl120/>.
2. Fiadeiro, J.L., Lopes, A.: An interface theory for service-oriented design. *Theoretical Computer Science* **503** (2013) 1–30
3. Vissani, I., Lopez Pombo, C.G., ȚuȚu, I., Fiadeiro, J.L.: A full operational semantics for asynchronous relational networks. In Diaconescu, R., Codescu, M., ȚuȚu, I., eds.: Proceedings of 22st International Workshop on Algebraic Development Techniques (WADT 2014). Volume 9463 of Lecture Notes in Computer Science., Sinaia, Romania, Springer-Verlag (September 2015) 131–150
4. Vissani, I., Lopez Pombo, C.G., Tuosto, E.: Communicating machines as a dynamic binding mechanism of services. In Gay, D., Alglave, J., eds.: Proceedings of 8th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES. Volume 203 of Electronic Proceedings in Theoretical Computer Science. (April 2016) 85–98
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. *Journal of the ACM* **30**(2) (1983) 323–342
6. Anabia, D.N.S.: Bisimulación de data-aware communicating finite state machines con propiedades en las acciones. Master's thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (November 2023) Advisors: Carlos G. Lopez Pombo and Hernán C. Melgratti.
7. MuleSoft: 8 principles of service-oriented architecture. On-line at <https://blogs.mulesoft.com/digital-transformation/soa-principles/> (2022)
8. IBM: What is service-oriented architecture (soa)? On-line at <https://www.ibm.com/topics/soa> (2024)
9. Microsoft: Service-oriented architecture. On-line at <https://learn.microsoft.com/en-gb/dotnet/architecture/microservices/architect-microservice-container-applications/service-oriented-architecture> (2022)
10. Oracle: Oracle soa suite. On-line at <https://www.oracle.com/middleware/technologies/soasuite.html> (2024)

11. Donovan, A.A., Kernighan, B.W.: The Go Programming Language. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Co., Inc. (2015)
12. Lopez Pombo, C.G., Martinez Suñé, A.E., Tuosto, E.: A dynamic temporal logic for quality of service in choreographic models. In Ábrahám, E., Dubslaff, C., Tarifa, S.L.T., eds.: Proceedings of 20th International Colloquium on Theoretical Aspects of Computing - ICTAC 2023. Volume 14446 of Lecture Notes in Computer Science., Lima, Perú, Springer-Verlag (December 2023) 119–138
13. Lange, J., Tuosto, E., Yoshida, N. River Publishers Series in Automation, Control and Robotics. In: A Tool for Choreography-Based Analysis of Message-Passing Software. River Publisher (2017) 125–146
14. Coto, A., Guanciale, R., Lange, J., Tuosto, E.: **ChorGram**: tool support for choreographic development. Available at <https://bitbucket.org/eMgssi/chorgram/src/master/> (2015)
15. Basile, D., Degano, P., Ferrari, G., Tuosto, E.: Playing with our CAT and communication-centric applications. In Albert, E., Lanese, I., eds.: Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings. Volume 9688 of Lecture Notes in Computer Science., Springer (2016) 62–73
16. Basile, D., ter Beek, M.: Contract automata library. *Sci. Comput. Program.* **221** (2022) 102841
17. Basile, D., ter Beek, M.H., Pugliese, R.: Synthesis of orchestrations and choreographies: Bridging the gap between supervisory control and coordination of services. *Logical Methods in Computer Science* **16**(2) (2020)
18. Basile, D., ter Beek, M.: A runtime environment for contract automata. In Chechik, M., Katoen, J., Leucker, M., eds.: Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings. Volume 14000 of Lecture Notes in Computer Science., Springer-Verlag (2023) 550–567
19. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A., Pompianu, L.: Contract-Oriented Programming with Timed Session Types. In Gay, S., Ravara, A., eds.: Behavioural Types: from Theory to Tools. River (2017) 27–48
20. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A., Pompianu, L.: A contract-oriented middleware. In Braga, C., Ölveczky, P., eds.: Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers. Volume 9539 of Lecture Notes in Computer Science., Springer-Verlag (2015) 86–104
21. Atzei, N., Bartoletti, M., Murgia, M., Tuosto, E., Zunino, R.: Contract-Oriented Design of Distributed Applications: a Tutorial. In Gay, S., Ravara, A., eds.: Behavioural Types: from Theory to Tools. Automation, Control and Robotics. River (2017) 1–26
22. Imai, K., Lange, J., Neykova, R.: Kmclib: Automated Inference and Verification of Session Types from OCaml Programs. In Fisman, D., Rosu, G., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, Cham, Springer International Publishing (2022) 379–386
23. Vasconcelos, C., Ravara, A.: From object-oriented code with assertions to behavioural types. In Seffah, A., Penzenstadler, B., Alves, C., Peng, X., eds.: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017, ACM (2017) 1492–1497
24. Genovese, G.: ChorEr: un analizzatore statico per generare Automi Coreografici da codice sorgente Erlang. Master’s thesis, University of Bologna (2023)

25. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In Bliudze, S., Bocchi, L., eds.: *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*. Volume 12134 of *Lecture Notes in Computer Science.*, Springer-Verlag (2020) 86–106
26. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. *Logical Methods in Computer Science* **12**(4) (2016)
27. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. In Beyer, D., Boreale, M., eds.: *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*. Volume 7892 of *Lecture Notes in Computer Science.*, Springer-Verlag (2013) 305–320
28. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Volume 6355 of *Lecture Notes in Computer Science.*, Springer (2010) 348–370