Adaptability as a Programming Pattern in SEArch

Carlos G. Lopez Pombo*
Centro Interdisciplinario de Telecomunicaciones, Electrónica, Computación y Ciencia Aplicada
Universidad Nacional de Río Negro - Sede Andina and CONICET
cglopezpombo@unrn.edu.ar

Pablo Montepagano
Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires
pmontepagano@dc.uba.ar

Emilio Tuosto
Gran Sasso Science Institute
emilio.tuosto@gssi.it

We recently introduced an execution infrastructure for service-based software dubbed SEArch, after Service Execution Architecture. SEArch is a language-independent execution infrastructure capable of performing transparent dynamic reconfiguration of software artefacts.

We argue here that our framework can be used to support adaptability. Our conviction is grounded on the underlying execution model of SEArch which hinges on a (*service*) *broker* that procures services at run-time based on formal contracts specified by components that "consume" services. Therefore, cloud applications can adapt to dynamic changes by modifying the contracts used to search for the services by-need.

1 Introduction

Adaptability can be intuitively understood as the property of a system to suitably react to changes to its execution environment. Often, adaptation impact on the behaviour of the system without necessarily affecting its architecture. For instance, fault-tolerance can be tackled by replicating components without affecting the overall architecture. Embracing, as we do, the view that this is not necessarily the case for cloud applications raises the question of one can support adaptation mechanisms that may change the architecture in controllable ways. Arguably, the problem boils down to adopt suitable programming features allowing developers to implement countermeasures to changes in the execution context.

In this paper we explore the possibility of (re-)using known mechanisms borrowed from the *service-oriented computing* (SOC) paradigm.[1] Our exploration hinges on an infrastructure, dubbed SEArch (after Service Execution Architecture) [1], for the discovery and binding of distributed services featuring language-oblivious and data-aware functional and behavioural compliance of services. We showcase how SEArch can support architectural reconfigurations dictated by adaptability. After a brief survey on SEArch, its architecture, and API (cf. Section 2, essentially borrowed from [1]), we illustrate its usage to tackle the adaptability scenarios described in [2]. We will argue that these scenarios can be uniformly

*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

[1] Although the terminology has evolved, SOC is still widely used and, we believe, still applies to variants such as cloud computing, fog and edge computing, or the many forms of distributed computing associated with what is known as the *Internet of (Every)Things*.

Submitted to:
WACA 2025

treated in SEArch with a single programming pattern. Finally, we draw some conclusions and discuss further work in 4.

2 An Overview of SEArch

There is a wide range of service-oriented architectures (SOAs) dictating design principles for SOC, each one with its own idiosyncrasy [3, 4, 5, 6]. 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 applications. 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. The computational model behind SEArch is supported by *Asynchronous Relational Networks* (ARNs) [7] as the formalisation of the elements of an interface theory for service-oriented software architectures. In ARNs there are two types of elements, *processes* and *communication channels* both equipped with *ports*. Ports of processes are called *provides-points*, while those of communication channels are called *requires-points*. In the operational semantics of ARNs given in [8] 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 contracts associated to them. Under this approach, interoperability is understood at a more abstract level decoupled from any computational aspect. We represent both provision and requirement contracts as *extended Communicating Finite State Machines* [9] (extended CFSM for short), an extension of *communicating finite-state machines* [10] with formal specifications of the functional and non-functional behaviour of service provided / required (depending of the nature of the port); providing a semantic notion of compatibility between provision and requirement contracts based on a bisimilarity check.

Figure 1 depicts the workflow. When launched, a component registers its communication channels to its *middleware*, each of which has its corresponding contract formalised as a set of extended 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 of 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 is triggered as follows:

1. M sends the service broker the contract of the communication channel;
2. for each extended CFSM `R` in the contract, the service broker queries the service repository for candidates;
3. the service repository returns a list of candidates in the form $\langle \texttt{Pr}, u \rangle$, where `Pr` is a extended CFSM and u is the URI of the service[2]; the broker checks whether the provision contract `Pr` is compatible with the requirement contract `R`, which is done by resorting to bisimilarity check;

[2] SEArch is parametric in the implementation of the service repository so we assume it is not capable of checking compli-

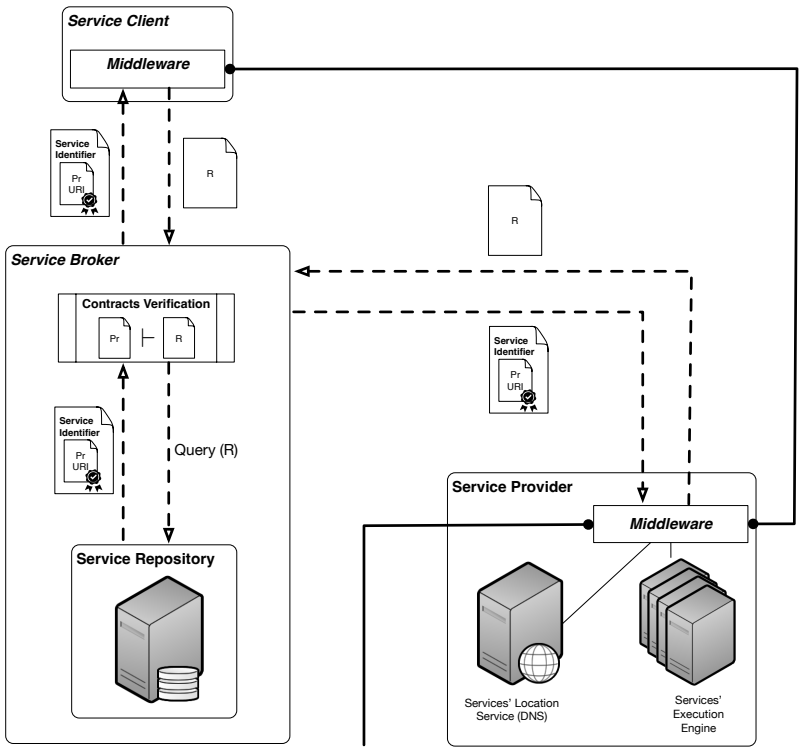

Figure 1: Service execution procedure in SEArch

4. once the service broker has found services satisfying all the requirement contracts, it returns the set of URIs to M;
5. M opens a communication with the middleware of each service returned by the provider requiring the execution of the corresponding service.

Then, M sends to or receives from the middleware of the partner component the actual message; and the execution process proceeds. Notice that during the execution new requirements might crop up; for instance, because some brokered services need further services.[3] This will initiate a new brokerage phase to discover the newly required services. The schematic view discussed before establishes 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 interface. 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 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 middleware that will remain open in order to support the (high level) communication with other services.

ance 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.

[3]This implies that a service may have one or more required-points associated to communication channels of its own.

- `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 of whether it is 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 session, 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 broker to initiate the communication infrastructure that will connect the service executing behind their middleware to the other participants in a communication session being setup.
- `StartChannel` to receive notifications about point-to-point communication channels. This operation formally notifies the 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 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.

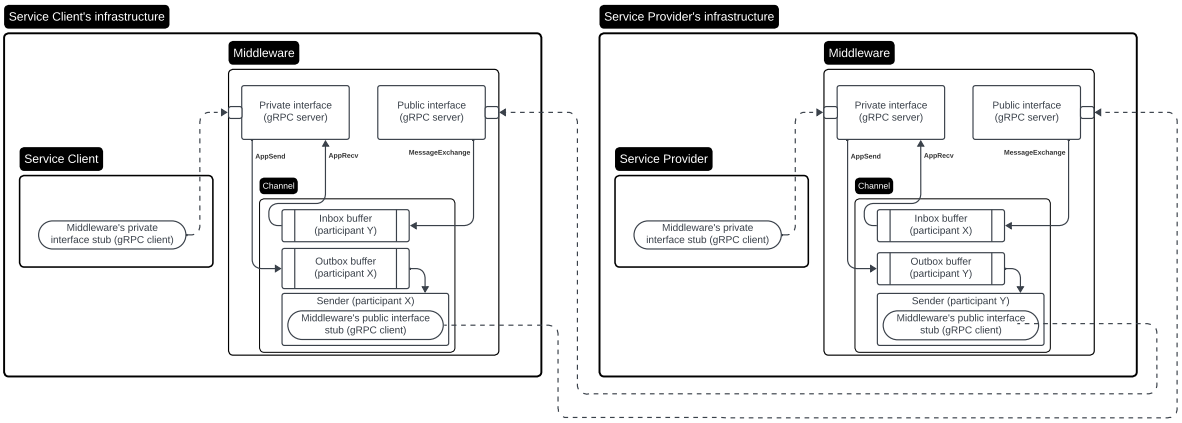

Figure 2: Point-to-point communication between a service client and a service provider.

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

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

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

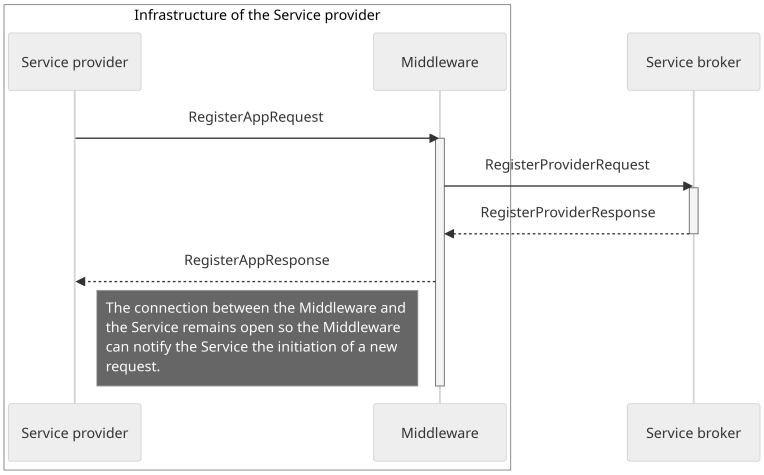

Figure 3: Sequence diagram of the process of registration of a service.

Figure 4 shows the brokerage process 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 ❸). Concurrently, the middleware begins the brokerage process by sending the contract to the service broker (step ❹) and enqueues the message while acknowledging the service client with a message of type `AppSendRespond` (step ❺). 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 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 message exchanges with the chosen service provider. In the first round, a message of type `InitChannelRequest` is sent (step ❼) to tell the 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

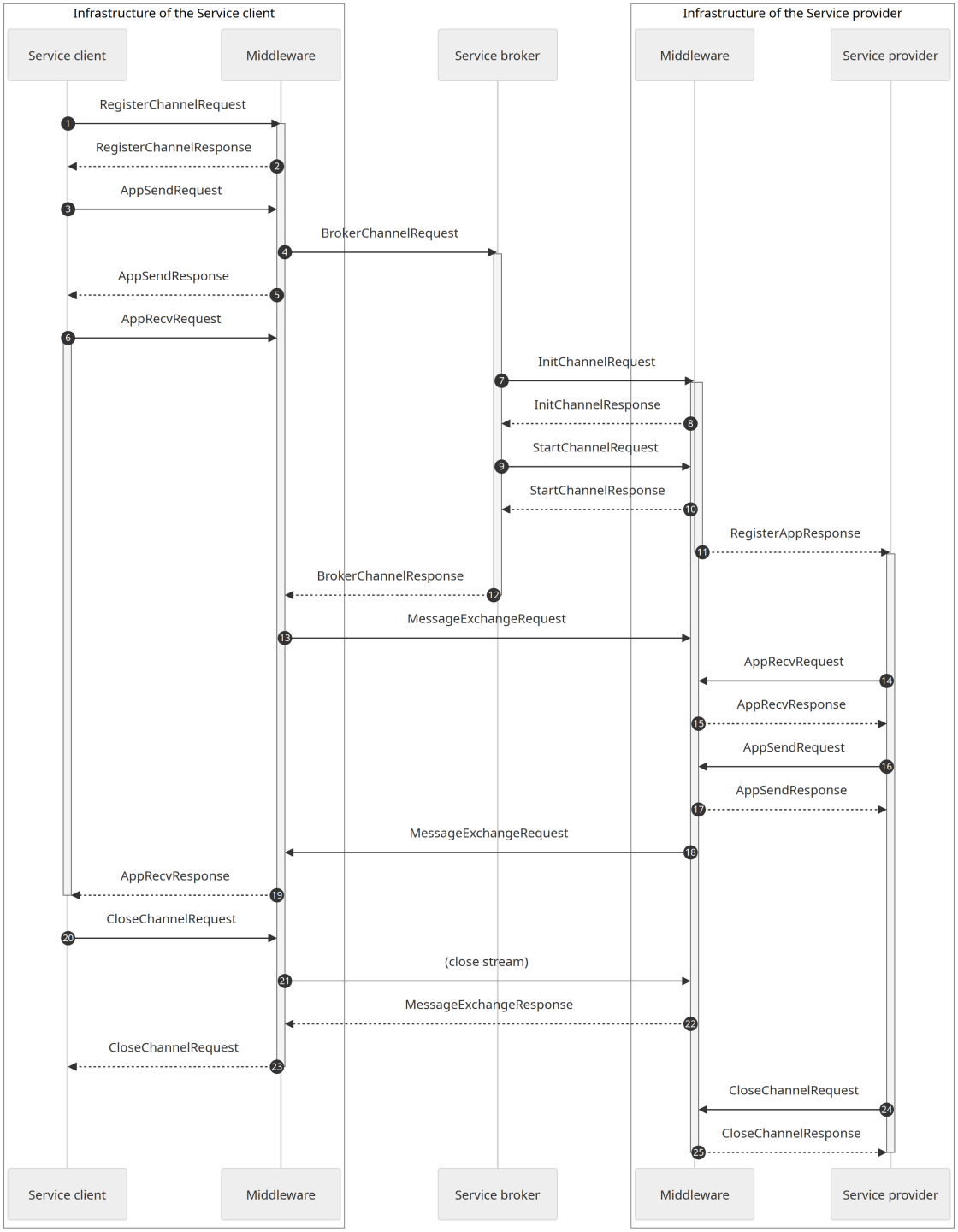

Figure 4: Sequence diagram of the process of brokerage of a communication channel.

to verify that the provider is indeed online. Upon receipt of this message, a middleware must accept incoming messages for this channel and enqueue them for the eventual reception by the service provider.

If all the service providers respond successfully with a message of type `InitChannelResponse` (step ❽), then the service broker performs a second round with a message of type `StartChannelRequest` (step ❾), to confirm that the communication session has been initiated.

After receiving both initialization messages, the middleware sends the service provider a message of type `RegisterAppResponse` (step ⓫) containing the UUID of the new communication session. Then, each service provider can start communicating over this session according to its contracts. Once a session is initiated, the middlewares establish unidirectional streams with each other to send messages. In Fig. 4 the middleware of the service client opens a stream with the other middleware by sending a message of type `MessageExchange` (step ⓭). After the service provider has received the message (steps ⓮- ⓯), it sends a message (steps ⓰- ⓱) forcing the middleware of the service provider to establish a stream in the opposite direction (step ⓲).

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

3 Adaptable TeaStore with SEArch

In a quest for a notion of adaptation, the authors of [11] advocate a conceptual framework grounded on a neat separation of concerns. Starting from the observation that the behaviour of a component is determined by a program defining *control* and manipulating *data*, adaptation requires to consider, and possibly, modify specific computational or physical resources. The conceptual framework in [11] envisages *adaptation* as the possibility of a component to manipulate and take decisions according to a well-identified set of *control data*. For instance, an adaptive behaviour may need to account for memory usage, network traffic, or environmental physical conditions like e.g., temperature. Embracing this view, allows one to define adaptation as the property of a component of controlling and possibly modifying at run-time its specific set of control data.

This acceptation implicitly require a *monitoring* functionality that can retrieve and provided to the component the control data necessary for enforcing adaptive behaviour.

We describe our programming pattern in Section 3.1 and then demonstrate how to apply our pattern to a TeaStore scenario in Section 3.2.

3.1 A programming pattern for adaptability in SEArch

We now discuss how the API and the infrastructure of SEArch can suitably support the conceptual framework of adaptation proposed in [11] in a general way. We argue, in fact, that this conceptual framework can be captured by a programming pattern that SEArch can naturally support.

An high-level view of services/applications executing in nodes of a SEArch network is given in Fig. 5. As shown, the service broker is connected to the SEArch middlewares of all the nodes in the network that execute their own services/applications. Zooming into one of the nodes –the large box in Fig. 5– one can see three main elements (the 3D boxes):

- a service providing some functionality (leftmost inner 3D box),
- the local middleware (bottom-most 3D box) and
- a detailed view of a service/application implemented as an adaptive system (top-most 3D box).

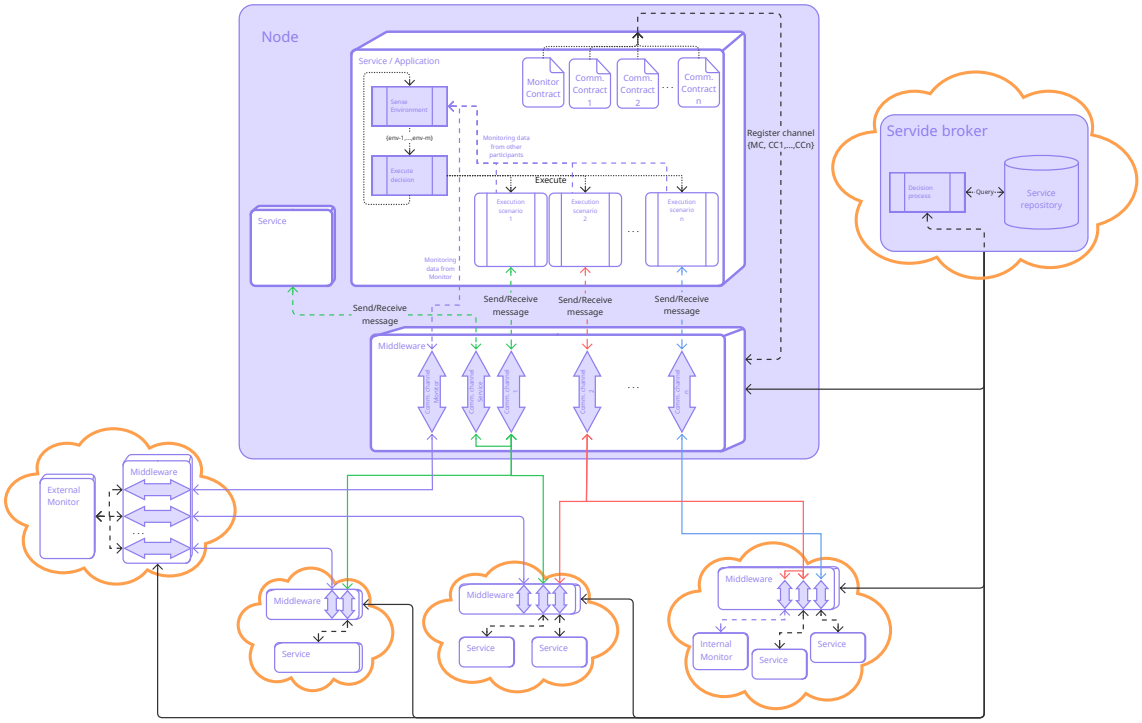

Figure 5: Architectural view of a programming pattern for implementing adaptive systems in SEArch

The latter element features a *Control loop* and the implementation of n alternative *Execution-scenarios* together with their associated *Communication contracts*. The control loop is conceptually split in two phases: first the control data is acquired from the monitoring activity and then a decision is taken on how to adapt (if at all).

As we discussed above, we take inspiration from [11] so adaptability can be understood as the capacity of an agent to respond to changes in the environment. Thus, a significant aspect of our approach relies on the monitoring of the execution environment in a way that such information can interact and modify the behaviour of the component. Our programming pattern supports two different monitoring strategies that can be used both in isolation, or combined. The *Sense environment* phase in the control loop is fed with the information collected by the monitoring activity, turns this information into control data, and analyses the control data to choose a specific scenario. The monitoring activity can be directly implemented by the component (*internal monitoring*) or outsourced to an external service (*external monitoring*). Internal monitoring is supported in SEArch by simply connecting a non-brokered service to the middleware of the component (see the lilac dashed arrow going from the execution scenarios to the *Sense environment* phase). External monitoring is supported by interacting with other participants connected through the communication channels connected to a separate monitoring service dynamically discovered and bound through the SEArch infrastructure.

The programming pattern we envisage is as follows:

1. the invocation of a service/application triggers the registration of the n alternative communication channels and the monitoring channel, whenever the latter is required
2. the *Control loop* starts a continuous check of the monitoring information collected and exchanged

by the *Monitor*, evaluate the environmental conditions based on this information, and chooses an *Execution scenario*; finally it dispatches the execution of the code implementing it,

3. the *Monitor* eventually detects a change in the environmental conditions and sending the newly collected data to the participants along the communication channel set up by the middleware for that specific scenario. This information is consumed by the *Control loop* which, again, evaluates the environmental conditions possibly requiring the execution to abort and restart in a different *Execution scenario*.

3.2 Adaptable TeaStore scenarios in SEArch

In this section we discuss how SEArch can provide support for implementing some salient aspects of the adaptation scenarios discussed in [2, Sec. 4]. We consider the scenario discussed in [2, Sec. 4.2] which we briefly summarise.

Cyberattack on External Services In this scenario the system is supposed to react when a service experiences a cyberattack. The system is deployed in a configuration that uses external services, say *Image Provider*, *Auth* (the authenticator), and *Database*. For simplicity, we only consider an implementation consisting of the *WebUI* and the *Image Provider*, for which there are two implementations, one as an external service and a fallback local implementation to be used whenever the external service is not available, for example, in the event of a cyberattack. Also, leaving aside functional and non-functional aspects of the interactions, we concentrate on the interoperability aspects so that contracts can be simply formalised as communicating finite state machines in Fig. 6. The contract for an external *Image Provider* service is formalised in Fig. 6a which also features a dedicated monitor for assisting in the detection of undesired events (in this case, a cyberattack). The contract for the fallback *Image Provider* service is in Fig. 6b; intuitively, in this case the monitoring activity is internalised in the *Image Provider* service.

The implementation of the adaptable *WebUI* is as follows:

1. the communication with the local middleware is established,
2. both global contracts shown in Fig. 6 are registered in the middleware. Fig. 6a expresses the standard behaviour of the system in which images are provided by an external *Image Provider* service monitored by a local *Monitor* and Fig. 6b shows the fallback local *Image Provider* service to be used in the case that external *Image Provider* becomes unavailable. In this case, the monitoring information notifying that the external *Image Provider* is under attack is sent through the transition `6 WebUI ! attack 7` of the participant `ImgP` and received by the participant `WebUI` through the transition `5 ImgP ? attack 6`,
3. the *WebUI* is initialised with both, the local and the external *Image Providers* channels, and selects the external one as the current image provider channel,
4. the main control flow of the *WebUI* is dispatched and the application starts requesting images through the function `get_image`,
5. the function `get_image` request the image from the current image provider through the function `req_image` (shown in the contract of Fig. 6a as the message `0 ImgP ! req 5` of the participant `WebUI`),
6. if the image is received (shown in the contract of Fig. 6a as the message `5 ImgP ? img 0` of the participant `WebUI`), the execution continues; but if the participant `ImgP` sends the message `attack`

```
.outputs ImgP
.state graph
  0 WebUI ? req 4
  0 WebUI ? stop 2
  2 Monitor ! stop 8
  4 Monitor ? allFine 5
  4 Monitor ? attack 6
  5 WebUI ! img 0
  6 WebUI ! attack 7
.marking 0
.end
```

```
.outputs Monitor
.state graph
  0 ImgP ? stop 2
  0 ImgP ! allFine 0
  0 ImgP ! attack 3
.marking 0
.end
```

```
.outputs WebUI
.state graph
  0 ImgP ! req 5
  0 ImgP ! stop 3
  5 ImgP ? attack 6
  5 ImgP ? img 0
.marking 0
.end
```

(a) Contract for the scenario with an external *Image Provider* service and a dedicated *Monitor* for the channel

```
.outputs ImgP
.state graph
  0 WebUI ? req 1
  0 WebUI ? stop 2
  1 WebUI ! img 0
.marking 0
.end
```

```
.outputs WebUI
.state graph
  0 ImgP ! req 1
  0 ImgP ! stop 3
  1 ImgP ? img 0
.marking 0
.end
```

(b) Contract for the fallback scenario where the implementation of the *Image Provider* is local

Figure 6: Contracts enabling adaptability with respect to the *Image Provider*

(shown in the contract of Fig. 6a as the message `6 WebUI ! attack 7` of the participant `ImgP`), an exception is thrown, triggering the adaptation by configuring the current image provider channel to point to the local image provider channel and then, requesting the image through the function `req_image` (shown in the contract of Fig. 6b as the message `0 ImgP ! req 1` of the participant `WebUI` and then the execution continues.

4 Conclusions

The key idea driving the design and implementation of SEArch was to develop an execution architecture providing dynamic and transparent reconfiguration of service-based software artifacts. Under SEArch, services are brokered by a dedicated component capable of resolving compliance between requirements and provisions, by statically analysing formal interoperability contracts. Noteworthy, contracts are formal and, besides specifying the expected communication protocol among participants, they can also declare functional and non-functional constraints. For simplicity, we considered here only contracts specifying the communication protocols, but SEArch can easily support adaptation scenarios using (non-)functional contracts as well.

The possibility of describing each alternative execution as a different communication channel provides the support for considering alternative executions, hence adapation scenarios, depending on the control data monitored from the environment. Therefore, supporting adaptability as the possibility of choosing the best fitted implementation according to data monitored from the execution environment;

presented in this work as a programming pattern. Although not shown here, our programming patter can be used to implement in SEArch also the other adaptation scenarios described in [11] once one has identified the relevant control data.

References

[1] Lopez Pombo, C.G., Montepagano, P., Tuosto, E.: SEArch: an execution infrastructure for service-based software systems. In Castellani, I., Tiezzi, F., eds.: Proceedings of Coordination Models and Languages - 26nd IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024. Volume 14676 of Lecture Notes in Computer Science., Springer-Verlag (June 2024) 314–330

[2] Bliudze, S., Palma, G.D., Giallorenzo, S., Lanese, I., Zavattaro, G., Ndadji, B.A.Z.: Adaptable teastore. On-line (2024) Available at https://arxiv.org/abs/2412.16060.

[3] MuleSoft: 8 principles of service-oriented architecture. On-line at https://blogs.mulesoft.com/digital-transformation/soa-principles/ (2022)

[4] IBM: What is service-oriented architecture (soa)? On-line at https://www.ibm.com/topics/soa (2024)

[5] Microsoft: Service-oriented architecture. On-line at https://learn.microsoft.com/en-gb/dotnet/architecture/microservices/architect-microservice-container-applications/service-oriented-architecture (2022)

[6] Oracle: Oracle soa suite. On-line at https://www.oracle.com/middleware/technologies/soasuite.html (2024)

[7] Fiadeiro, J.L., Lopes, A.: An interface theory for service-oriented design. Theoretical Computer Science **503** (2013) 1–30

[8] Vissani, I., Lopez Pombo, C.G., Ţuţu, I., Fiadeiro, J.L.: A full operational semantics for asynchronous relational networks. In Diaconescu, R., Codescu, M., Ţuţu, 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

[9] Lopez Pombo, C.G., Martinez Suñé, A.E., Melgratti, H.C., Senarruzza Anabia, D.N., Tuosto, E.: SEArch: an execution infrastructure for service-based software systems. In Giusto, C.D., Ravara, A., eds.: Proceedings of Coordination Models and Languages - 27nd IFIP WG 6.1 International Conference, COORDINATION 2025, Held as Part of the 20th International Federated Conference on Distributed Computing Techniques, DisCoTec 2025. Volume TBD of Lecture Notes in Computer Science., Springer-Verlag (June 2025) TBD

[10] Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM **30**(2) (1983) 323–342

[11] Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In de Lara, J., Zisman, A., eds.: Fundamental Approaches to Software Engineering, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 240–254