Microservices: a Language-based Approach

Claudio Guidi, Ivan Lanese, Manuel Mazzara, Fabrizio Montesi

Abstract Microservices is an emerging development paradigm where software is obtained by composing autonomous entities, called (micro)services. However, microservice systems are currently developed using general-purpose programming languages that do not provide dedicated abstractions for service composition. Instead, current practice is focused on the deployment aspects of microservices, in particular by using containerization. In this chapter, we make the case for a language-based approach to the engineering of microservice architectures, which we believe is complementary to current practice. We discuss the approach in general, and then we instantiate it in terms of the Jolie programming language.

1 Introduction

Microservices [6, 19, 3] is an architectural style stemming from Service-Oriented Architectures (SOAs) [11]. Its main idea is that applications are composed by small independent building blocks – the (micro)services – communicating via message passing. Recently, microservices have seen a dramatic growth in popularity, both in terms of hype and of concrete applications in real-life software [19]. Several companies are involved in a major refactoring of their backend systems [2] in order to improve scalability [4].

Claudio Guidi
italianaSoftware srl, Imola, Italy e-mail: cguidi@italianasoftware.com

Ivan Lanese
Focus Team, University of Bologna/INRIA, Italy e-mail: ivan.lanese@gmail.com

Manuel Mazzara
Innopolis University, Russian Federation e-mail: m.mazzara@innopolis.ru

Fabrizio Montesi
University of Southern Denmark e-mail: fmontesi@imada.sdu.dk

Current approaches for the development of server-side applications use mainstream programming languages. These languages, frequently based on the object-oriented paradigm, provide abstractions to manage the complexity of programs and their organization into modules. However, they are designed for the creation of single executable artifacts, called monoliths. The modules of a monolith cannot execute independently, since they interact by sharing resources (memory, databases, files,...). Microservices support a different view, enabling the organization of systems as collections of small *independent* components. Independent refers to the capability of executing each microservice on its own machine (if needed). This can be achieved because services have clearly defined boundaries and interact purely by means of message passing. Microservices inherit some features from SOAs, but they take the same ideas to a much finer granularity, from programming in the large to programming in the small. Indeed, differently from SOAs, microservices highlight the importance for services to be *small*, hence easily reusable, easily understood, and even easily rebuilt from scratch if needed. This recalls the single responsibility principle of object-oriented design [12].

Since it is convenient to abstract from the heterogeneity of possible machines (e.g., available local libraries and other details of the OS), it is useful to package a service and all its local dependencies inside a *container*. Container technologies, like Docker [13], enable this abstraction by isolating the execution of a service from that of other applications on the same machine. Indeed, in the literature about microservices, the emphasis is on deployment: since microservices live inside containers they can be easily deployed at different locations. A major reason for this focus is that microservices are thought since their inception as a style to program in the cloud, where deployment and relocation play key roles. Even if microservices have now evolved well beyond cloud computing, the emphasis on deployment and containerization remains [6].

In this chapter, while supporting the current trend of microservices, we advocate for moving the emphasis from deployment to development, and in particular to the programming language used for development. We think that the chosen language should support the main mechanism used to build microservice architectures, namely service composition via message passing communications. Furthermore, in order to master the related complexity, we support the use of well-specified interfaces to govern communication. Mainstream languages currently used for the development of microservices do not provide enough support for such communication modalities. In particular, service coordination is currently programmed in an unstructured and ad-hoc way, which hides the communication structure behind less relevant low-level details.

While the idea of current methodologies for developing microservices can be summarized as “it does not matter how you develop your microservices, provided that you deploy them in containers”, the key idea behind our methodology is “it does not matter how you deploy your microservices, provided that you build them using a microservice programming language”. We describe our methodology in general, and support the abstract discussion by showing how our ideas are implemented in the Jolie language [18, 10].

2 Language-based Approach

The fine granularity of microservices moves the complexity of applications from the implementation of services to their coordination. Because of this, concepts such as communication, interfaces, and dependencies are central to the development of microservice applications. We claim that such concepts should be available as first-class entities in a language that targets microservices, in order to support the translation of the design of a microservice architecture (MSA) into code without changing domain model. This reduces the risk of introducing errors or unexpected behaviours (e.g., by wrong usage of book-keeping variables).

What are then the key ingredients that should be included in a microservice language? Since a main feature of microservices is that they have a small size, realistic applications are composed by a high number of microservices. Since microservices are independent, the interactions among them all happen by exchanging messages. Hence, programming an MSA requires to define large and complex *message exchange structures*. The key to a "good" microservice language is thus providing ways to modularly define and compose such structures, in order to tame complexity. We discuss such ways in the rest of this section.

Interfaces In order to support modular programming, it is necessary that services can be deployed as "black boxes" whose implementation details are hidden. However, services should also provide the means to be composed in larger systems. A standard way of obtaining this is to describe via *interfaces* the functionalities that services provide to and require from the environment. Here, we consider interfaces to be sets of *operations* that can be remotely invoked. Operations may be either fully asynchronous or follow the typical request-response pattern. An operation is identified by a name and specifies the data types of the parameters used to invoke it (and possibly also of the response value).

Once we accept that interfaces are first-class citizens in microservices, it makes sense to have operators to manipulate them. Since interfaces are sets of operations, it is natural to consider the usual set-theoretical operators, such as union and intersection. For example, a gateway service may offer an interface that is the union of all the interfaces of the services that it routes messages to.

Ports Microservices may run in heterogeneous environments that use different communication technologies (e.g., TCP/IP sockets, Bluetooth, etc.) and data protocols (e.g., HTTPS, binary protocols, etc.). Moreover, a microservice may need to interact with many other services, each one possibly offering and/or requiring a different interface. A communication *port* concretely describes how some of the functionalities of a service are made available to the network, by specifying the three key elements above: interface, communication technology, and data protocol. Each service may be equipped with many ports, of two possible kinds. *Input ports* describe the functionalities that the service provides to the rest of the MSA. Conversely, *output ports* describe the functionalities that the service requires from the rest of the MSA. Ports should be specified separately from the implementation of a service, so that one can see what a service provides and what it needs without having to check its actual implementation. This recalls the use of type signatures for

functions in procedural programming, with the difference that here the environment is heterogeneous and we thus need further information (communication medium, data protocol).

Example 1. Consider an online shopping service connected to both the Internet and a local intranet. This service may have 2 input ports, `Customers` and `Admin`, and 1 output port, `Auth`. Input port `Customers` exposes the interface that customers can use on the web, using HTTPS over TCP/IP sockets. Input port `Admin` exposes the administration controls of the service to the local intranet, using a proprietary binary protocol over TCP/IP sockets. Finally, output port `Auth` is used to access an authentication service in the local network.

Workflows Service interactions may require to perform multiple communications. For example, our previous `Customers` service may offer a "buy and ship" functionality, implemented as a structured protocol composed by multiple phases. First, the customer may select one or more products to buy. In the second phase, the customer sends her destination address and selects the shipment modality. Finally, the customer pays, which may require the execution of an entire sub-protocol, involving also a bank and the shipper. Since structured protocols appear repeatedly in microservices, supporting their programming is a key issue. Unfortunately, the programming of such *workflows* is not natively supported by mainstream languages, where all possible operations are always enabled. For example, consider a service implemented by an object that offers two operations `login` and `pay`. Both operations are enabled at all times, but invoking `pay` before `login` raises an error. This causal dependency is programmed by using a book-keeping variable, which is set when `login` is called and is read by method `pay`. Using book-keeping variables is error-prone, in particular it does not scale when the number of causality links increases [15].

A microservice language should therefore provide abstractions for programming workflows. For example, one can borrow ideas from BPEL [20], process models [14] or behavioral types [9], where the causal dependencies are expressed syntactically using operators such as sequential and parallel compositions, e.g., `login;pay` would express that `pay` becomes available only after `login`. All the vast literature on business process modeling [22] can offer useful abstractions to this regard.

Processes A workflow defines the blueprint of the behavior of a service. However, at runtime, a service may interact with multiple clients and other external services. In our online shopping example, service `Customers` may have to support multiple users. Beyond that, the authentication service may be used both by service `Customers` and by other services, e.g., a `Billing` service. This is in line with the principle that microservices can be reused in different contexts. A service should thus support multiple executions of its workflow, and such executions should operate concurrently (otherwise, a new request for using the service would have to wait for previous usages to finish before being served).

A *process* is a running instance of a workflow, and a service may include many processes executing concurrently. The number of processes changes at runtime,

since external parties may request the creation of a new process, and processes may terminate. Each process runs independently of the others, to avoid interference, and, as a consequence, it has its own private state.

3 The Jolie language

Jolie [18, 10] is a language that targets microservices directly. Jolie was designed following the ideas discussed in Section 2. These were initially targeted at offering a language for programming distributed systems where all components are services, which later turned out to become the microservices paradigm.

Jolie is an imperative language where standard constructs such as assignments, conditionals, and loops are combined with constructs dealing with distribution, communication, and services. Jolie takes inspiration from WS-BPEL [20], an XML-based language for composing services, and from classical process calculi such as CCS [14] (indeed, the core semantics of Jolie is formally defined as a process calculus [8, 16]), but transfers these ideas into a full-fledged programming language. While we refer to [18, 10] for a detailed description of the Jolie language and its features, we discuss below the characteristics of Jolie that make it an instance of the language-based approach to microservices that we are advocating for.

The connection between Jolie and MSAs is at a very intimate level, e.g., even a basic building block of imperative languages like variables has been restructured to fit into the microservice paradigm. Indeed, microservices interact by exchanging data that is typically structured as trees (e.g., JSON or XML, supported in HTTP and other protocols) or simpler structures (e.g., database records). Thus, Jolie variables always have a tree structure [16], which allows the Jolie runtime to easily marshal and unmarshal data. It is clear that such a deep integration between language and microservice technologies cannot be obtained by just putting some additional library or framework on top of an existing language (which, e.g., would rely on variables as defined in the underlying language), but requires to design a new language from the very basic foundations.

A main design decision of the Jolie language is the separation of concerns between behavior and deployment information [17, 18]. Here with deployment information we mean both the addresses at which functionalities are exposed, and the communication technologies and data protocols used to interact with other services. In particular, as discussed in the previous section, each Jolie service is equipped with a set of ports: input ports through which the service makes its functionalities available, and output ports used to invoke external functionalities. Thanks to this separation of concerns, one can easily change how a Jolie microservice communicates with its environment without changing its behavior.

Example 2. If the online shopping service of Example 1 is implemented in Jolie, its `Customers` input port can be declared as:

```
1 inputPort Customers {
```

```
  Location: "socket://www.myonlineshop.it:8000"
  Protocol: https
  Interfaces: CustomersInterface
  }
```

The port declaration specifies the location where the port is exposed, which includes both the communication technology, in this case a TCP/IP socket, and the actual URL. Also, it declares the data protocol used for communication, in this case HTTPS. Finally, the port refers to the interface used for communication, which is described separately, and which can take, e.g., the form:

```
interface CustomersInterface {
  RequestResponse: getList( void )( productIdList ),
                   getPrice( productId )( double ),
                   ...
  }
```

This interface provides two request-response operations, `getList` and `getPrice`, and their signature. Types **`void`** and **`double`** are built-in, while `productIdList` and `productId` are user defined, hence their definition has to be provided.

Having ports and interfaces as first-class entities in the language allows one to clearly understand how a service can be invoked, and which services it requires. Furthermore, the same functionality can be exposed in different ways just by using different input ports, without replicating the service. For instance, in the example above, one can define a new port providing the same functionality using the SOAP protocol. Finally, part of the port information, namely location and protocol, can be changed dynamically by the behavior, improving the flexibility.

Jolie also supports workflows [21]. Indeed, each Jolie program is a workflow: it includes receive operations from input ports and send operations to output ports, combined with constructs such as sequence, conditional and loop. Notably, it also provides parallel composition to enable concurrency, and input-guarded choice to wait for multiple incoming messages. Input ports are available only when a corresponding receive is enabled, otherwise messages to this port are buffered.

Example 3. Jolie also provides novel workflow primitives that can be useful in practical scenarios. An example is the provide-until construct [15], which allows for the programming of repetitive behavior driven by external participants. Using standard workflow operators and provide-until, we can easily program a workflow for interacting with customers in our online shopping service from Section 2.

```
main {
  login()( csets.sid ) { csets.sid = new };
  provide
    [ addToCart( req )( resp ) { /* ... */ } ]
    [ removeFromCart( req )( resp ) { /* ... */ } ]
  until
```

```
7       [ checkout( req )( resp ) { pay; ship } ]
8       [ logout() ]
9       }
```

The workflow above starts by waiting for the user to login (operation `login` is a request-response that returns a fresh session identifier `sid`, used for correlating incoming messages [16] from the same client later on). We then enter a provide-until construct where the customer is allowed to invoke operations `addToCart` and `removeFromCart` multiple times, until either `checkout` or `logout` is invoked. In the case for `checkout`, we then enter another workflow that first invokes procedure `pay` and then procedure `ship` (each procedure defines its own workflow).

From a single workflow multiple processes are generated. Indeed, at runtime, when a message reaches a service, correlation sets [16] (kept in the special **`csets`** structure in the example above) are used to check whether it targets an already running process. If so, it is delivered to it. If not, and if it targets an initial operation of the workflow, a new process is spawned to manage it. Notably, multiple processes with the same workflow can be executed either concurrently, or sequentially. This last option is mainly used to program resource managers, which need to enforce mutual exclusion on the access to the resource.

4 Conclusions and Related Work

We made the case for a linguistic approach to microservices, and we instantiated it on the Jolie language. Actually, any general-purpose language can be used to program microservices, but some of them are more oriented towards scalable applications and concurrency (both important aspects of microservices). Good examples of the latter are Erlang [5] and Go [7]. Between the two, Erlang is the nearest to our approach: it has one of the most mature implementations of processes, and some support for workflows based on the actor model (another relevant implementation of actors is the Akka framework [1] for Scala and Java). However, Erlang and Go do not separate behavior from deployment, and more concretely do not come with explicitly defined ports describing the dependencies and requirements of services. WS-BPEL [20] provides many of the features we described, including ports, interfaces, workflow and processes. However, it is just a composition language, and cannot be used to program single services. Also, WS-BPEL implementations are frequently too heavy for microservices.

Our hope is that other languages following the language-based approach would emerge in the near future. This would also allow one to better understand which features are key in the approach, and which ones are just design decisions that can be changed. For instance, it would be interesting to understand whether language support for containerization would be useful, and which form it could take. Such support is currently absent in Jolie, but it would provide a better integration with the

classic approach focused on deployment. A related topic would be to understand how to improve the synergy between Jolie and microservices on one side, and the cloud and IoT on the other side.

References

1. Akka. http://akka.io/.
2. N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. Microservices: Migration of a mission critical system. https://arxiv.org/abs/1704.04173.
3. N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Springer, 2017.
4. N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina. Microservices: How to make your application scale. In *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*. Springer, 2017.
5. Erlang. https://www.erlang.org/.
6. M. Fowler and J. Lewis. Microservices. *ThoughtWorks*, 2014.
7. Go language. https://golang.org/.
8. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A calculus for service oriented computing. In *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.
9. H. Hüttel et al. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.
10. The Jolie language website. http://www.jolie-lang.org/.
11. M.C. MacKenzie et al. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.
12. R.C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
13. D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.
14. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
15. F. Montesi. Process-aware web programming with Jolie. *Sci. Comput. Program.*, 130:69–96, 2016.
16. F. Montesi and M. Carbone. Programming services with correlation sets. In *ICSOC*, volume 7084 of *LNCS*, pages 125–141. Springer, 2011.
17. F. Montesi, C. Guidi, and G. Zavattaro. Composing services with JOLIE. In *ECOWS*, pages 13–22. IEEE Computer Society, 2007.
18. F. Montesi, C. Guidi, and G. Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
19. S. Newman. *Building microservices*. O'Reilly Media, Inc., 2015.
20. OASIS. *Web Services Business Process Execution Language Version 2.0*, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.
21. Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016*.
22. Zhixian Yan, Manuel Mazzara, Emilia Cimpian, and Alexander Urbanec. Business process modeling: Classifications and perspectives. In *Business Process and Services Computing: 1st International Working Conference on Business Process and Services Computing, BPSC 2007, September 25-26, 2007, Leipzig, Germany.*, page 222, 2007.