Medley: An Event-Driven Lightweight Platform for Service Composition

Elyas Ben Hadj Yahia[1,3](✉), Laurent Réveillère[1], Yérom-David Bromberg[2], Raphaël Chevalier[3], and Alain Cadot[3]

[1] LaBRI, Université de Bordeaux, 33400 Talence, France
{elyas.bhy,reveillere}@labri.fr
[2] IRISA, Université de Rennes, 35000 Rennes, France
david.bromberg@irisa.fr
[3] CProDirect, 33700 Mérignac, France
{raphael.chevalier,a.cadot}@cprodirect.fr

Abstract. Distributed applications are evolving at a frantic pace, critically relying on each other to offer a host of new functionalities. The emergence of the service-oriented paradigm has made it possible to build complex applications as a set of self-contained and loosely coupled services that work altogether in concert. However, the traditional vision of Service-Oriented Architectures (SOA) based on web service specifications does not meet the trend of many major service providers. Instead, they promote microservices, a refinement of SOA focusing on lightweight communication mechanisms such as HTTP. Therefore, existing approaches for orchestrating the composition of various services become unusable in practice.

In this paper, we introduce Medley, an event-driven lightweight platform for service composition. Medley is based on a domain-specific language for describing orchestration and a compiler that produces efficient code. We have used Medley to develop various compositions, involving a large number of existing services. Our evaluation shows that it scales both on a mainstream server and an embedded device while consuming a reasonable amount of resources.

Keywords: Web composition · Domain-specific languages · Services orchestration · Event-driven programming · Microservices

1 Introduction

Since the early days of distributed computing there was a primitive notion of services that took its origins from RPC mechanisms [1]. The concept of services was significantly refined across the last decades to have a strong impact on the distributed computing landscape, in particular, due to the emergence of the Service Oriented Architecture (SOA) paradigm notably via the use of the Web Service stack as defined by the WS-* specifications.

© Springer International Publishing Switzerland 2016
A. Bozzon et al. (Eds.): ICWE 2016, LNCS 9671, pp. 3–20, 2016.
DOI: 10.1007/978-3-319-38791-8_1

From a higher perspective, SOA has promoted at least two major trends that have a long term impact. First, it has promoted a standardized way to build an application (that can itself be seen as a service) as a set of well specified, independent, self-contained and loosely coupled services that work altogether in concert. Second, it has proven that services act as a valuable paradigm to design complex applications.

As a result, we live in a service-oriented world. Applications ranging from the simplest smartphone application to the Web's most complex one, strive, in one way or another, to interact with value-added services, potentially made themselves from other services. In other terms, applications are increasingly built using the SOA paradigm and integrate a plethora of composable services. Moreover, as services are autonomous and deployed, undeployed or upgraded independently from each other, SOA enables application developers to have a fine-grained control on how to smoothly update their applications and how to make them scalable in production.

Hence, nowadays, the development of SOA-based applications comes together with continuous service development and continuous service integration practice[1]. This new trend coupled with the steady proliferation of services is not without challenges, and potentially obsoletes the traditional vision of SOA [19], along with their classical implementations based on the WS-* specifications (such as SOAP, WSDL, BPEL). For instance, the use of BPEL, the *de facto* standard, as a workflow to compose a plethora of services may be inadequate according to the developers' expectations. In fact, BPEL is a low-level and verbose language that describes *how* services need to be composed instead of defining *what* should be realized. Consequently, the quantity of code developers have to write in BPEL is proportional to the number of services they want to compose. Therefore, the complexity of the code increases making most often the use of BPEL not really suitable in practice. Furthermore, existing workflow languages typically require strongly-typed and well-defined interfaces from composed services. However, defining such interfaces is not the trend anymore due to the fast proliferation of services that most often expose their Web APIs without any contracts (such as with REST for instance) [15]. Thus, there is a need to write some glue code to compose services in an ad hoc and fast manner.

From another perspective, with the emergence of continuous service integration and development (often referred to as *DevOps*), workflow languages need to support not only static composition of well-specified services, but also on-the-fly integration of services that have not been previously planned at design time [23,25]. Finally, workflow languages are usually bundled with an execution engine such as an Enterprise Service Bus (ESB). However, ESBs are well known to be heavyweight containers. It does not meet the trend of lightweight containers, as popularized by Docker, which enables developers to deploy their service compositions wherever they want, such as personal clouds, according to privacy requirements [10].

[1] J. Lewis and M. Fowler, Microservices
http://martinfowler.com/articles/microservices.html.

Hence, the SOA paradigm has to evolve. Well known service providers such as Netflix, SoundCloud, Amazon, Spotify, have already widely adopted a refinement of the SOA paradigm named microservices. Microservices is no more than a SOA instance constrained to the basics of HTTP, i.e. with a RESTful style, without the WS-* specifications, and coupled with a variety of tools to promote fast deployment and undeployment of services. However the challenge to compose services stays open to microservices practitioners that are free to use the programming language they want.

In this paper, we introduce Medley, an event-driven lightweight platform for service composition. Medley's architecture is specifically designed to tackle the aforementioned four problematic issues encountered when orchestrating a composition of various services. Additionally, Medley meets the current trends in terms of continuous service integration and development as expected to promote a continuously evolving SOA. Our approach is based on a domain-specific language (DSL) for describing orchestrations using high-level constructs and domain-specific semantics. Once defined, a Medley specification is compiled into low-level code run on top of an event-driven process-based and lightweight platform. By providing an abstraction layer between the low-level implementation and the high-level business logic, the language allows users to express compositions with fine-grain tuning of both control flow and data flow.

The rest of this paper is organized as follows. Section 2 presents the range of issues that arise when orchestrating a composition of several services. Section 3 describes the Medley architecture and introduces a DSL for describing orchestrations. Section 4 describes the main challenges in code generation, and then presents an event-driven lightweight platform to support execution of service compositions. Section 5 demonstrates the efficiency of Medley and its scalability both on servers and embedded devices, and presents a comparative study of supported features. Section 6 discusses related work. Finally, Sect. 7 concludes and presents future work.

2 Issues in Service Composition

The composition of heterogeneous services is a daunting task for many developers. Several languages, including BPEL, have been proposed to ease the orchestration of service compositions. However, they all fail in the context of microservices. We illustrate issues developers have to face in the remainder of this section.

Complexity of Orchestrations. An orchestration may require monitoring a service for new events or state changes. This monitoring can be performed either synchronously by repeatedly polling the endpoint (pull mode) or by registering a callback for an asynchronous notification (push mode). When services only support polling, clients have to initiate a request to the server to retrieve the current state of the service. Then, the client compares this state with the previous one to detect any changes. Despite the advantages of push mode, developing applications based on the asynchronous paradigm is known to be challenging for

many developers. When data needs to be propagated between subsequent asynchronous actions, the corresponding information has to be stored by the runtime system at the point of the asynchronous call. The runtime system then passes it back to the stored continuation function when the corresponding response is received. Integrating services based on active polling may also be challenging for the developer. She needs to set up a reasonable frequency for polling to avoid resources waste while preserving good responsiveness. When the same service is used several times, its invocations could be factorized among several clients. However, identifying such global optimization opportunities is difficult when the orchestration code is hard-written and each composition is developed independently from each other.

Heterogeneity of Unspecified Interfaces. Existing orchestration languages such as BPEL require strongly-typed and well-defined interfaces from composed services. They rely on description languages like WSDL that have been extensively and successfully used for many years. The microservices architecture, however, promotes the use of RESTful services for which such description languages do not necessarily exist. Therefore, off-the-shelf tools are unpractical in that context. In addition, services that provide similar content are often heterogeneous either in the communication paradigm they rely on (synchronous *vs.* asynchronous) and in the format of data they provide. As an example, consider a custom daily news digest where a user receives an email containing information formatted to her liking about her favorite news. The developer has to specify how to interact with these news providers, what information to retrieve and how to aggregate data to produce a digest. As the number of services increase, this task becomes laborious.

Dynamicity of Service Composition. Compositions of services are usually statically specified and make explicit the connections between the interacting composed services. This design-time coupling prevents an orchestration to dynamically adapt its behavior to new services being deployed, undeployed or upgraded. Microservices architecture, however, promotes dynamicity although not providing insights on how to achieve it in practice. Supporting adaptation at runtime is known to complexify the task of the developer as she needs not only to focus on the orchestration of several services but also on how to smoothly react to service changes. As an example, consider the custom daily news digest orchestration scenario. To prevent failure in case the mail service become unavailable for some time, the user would like to specify a pool of mail services that can be used indifferently. However, defining such mail service selection policy in languages such as BPEL requires explicit handling of errors by the user.

Privacy Preservation of Execution Platforms. Orchestration languages including BPEL rely on execution environments such as ESB. Although powerful, they are well known to be heavyweight containers. Therefore, their deployment requires a significant amount of resources and renders their usage in small or personal entities more difficult. Sharing an execution platform, however, implies that personal sensitive data goes outside the boundaries of the personal network of the user.

To preserve privacy, one would want to deploy its own execution platform in her house or company. The emergence of low cost embedded devices such as the Raspberry Pi makes it possible. Execution platforms targeting these constrained devices need to be both lightweight and scale well with the increasing number of services users want to compose.

3 Medley Platform

To abstract away the low-level details when composing heterogeneous services from the users, we introduce Medley, a lightweight platform coupled with a DSL that enables users to express service compositions from a more abstract level as opposed to several other languages, such as BPEL. Overall, Medley enables users to reason and to focus on business logic rather than be disrupted by technical implementation details and issues. In the remainder of the section, we introduce our approach to create composite services with the Medley platform and we present its associated DSL.

3.1 Approach

Based on a particular set of services to compose (See Fig. 1 ❶), a user specifies, via the use of the Medley DSL, two kinds of information: (i) how to assemble together the services, (ii) the composition logic (See Fig. 1 ❷). In particular, with Medley, services are mapped to processes, and the process workflow is expressed in terms of patterns of events. Accordingly, the user is expressing in a simpler manner which processes to invoke according to events that may occur. The written specification is then given as input to the Medley compiler (See Fig. 1 ❸). The compiler in turn generates the adequate low-level code enabling communications among the assembled processes. In fact, service orchestration is instantiated as an event-based inter-process communication, conceptually similar to what we can encounter in traditional POSIX systems (See Fig. 1 ❹). Each service orchestration mapped to a set of processes (e.g. C_1 to C_n) is isolated

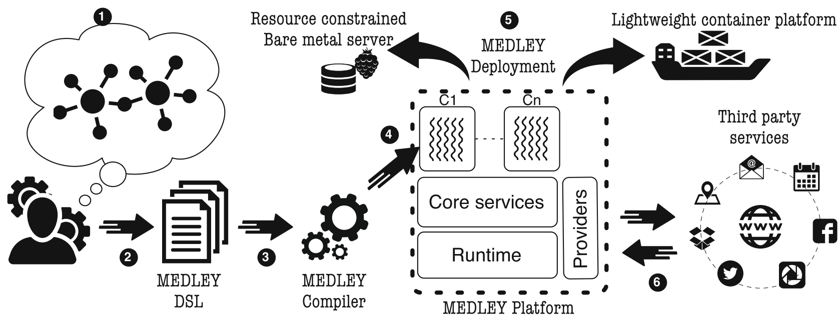


Fig. 1. Steps involved in the scenario described above

from each other, and run in a sandbox. Hence, for instance, several users can deploy different services orchestration without interferences among each other.

The Medley platform is lightweight, and hence can itself be run and deployed on either resource-constrained bare metal servers like Raspberry Pi, or on lightweight container platforms such as Docker (See Fig. 1 ❺). Finally, the Medley platform takes charge transparently, on the behalf of the users, of the interaction with third-party services (Fig. 1 ❻) as expected by the users according to the Medley specifications they have written.

3.2 The Medley DSL

The Medley DSL allows users to declare and configure processes to use and to compose. In particular, the Medley DSL enables users to express how to compose processes altogether according to the events that can occur on their respective output streams. Figure 2 gives an overview of a composition of a set of processes and enables us to introduce the DSL, with the help of Fig. 3 that gives a subset of the grammar.

Figure 2 describes a composition that periodically (line 2, 13) checks for new high-priority issues created on a specific GitHub[2] repository. If a new issue is detected, it notifies the user by sending her an email containing the issue's URL. The email service is selected from a pool of interchangeable services, enabling fault-tolerance on service unavailability. It also notifies the user if an error is encountered with the GitHub service when polling for new issues. Furthermore, this example enables us to highlight some key language operators of the Medley DSL, and concepts of the Medley platform.

```
1  composition {
2   process tick = require("Medley/Tick");
3   process getNewIssues = require("Github/GetNewIssues");
4   getNewIssues.init({"credentials": "<label>"});
5   process gmail = require("Gmail/SendEmail");
6   process outlook = require("Outlook/SendEmail");
7   // ...
8   pool process sendEmail = require("Medley/Pool");
9   sendEmail.add(gmail, outlook);
10  sendEmail.init({"strategy": "round-robin"});
11  on (tick:out) do {
12    stream issues = getNewIssues.invoke({
13      "repository": "medley/repo"
14    });
15    on (issues:out as issue) do {
16      if ({{$.issue.priority}} == "high") {
17        sendEmail.invoke({
18          "to": "john@doe.com",
19          "body": "New issue: {{$.issue.url}}"
20        });
21    } }
22    on (issues:err as error) do {
23      sendEmail.invoke({
24        "to": "john@doe.com",
25        "body": "Error encountered while fetching new issues: {{$.error.message}}"
26      });
27 } } }
```

Fig. 2. A composition example using Medley DSL

[2] A Git repository hosting service.

Language Operators. The `init` method (Fig. 2, line 4, 11) allows the user to configure the process with initialization parameters. These parameters persist throughout the lifecycle of the process instance. The `invoke` method (line 14, 19, 26) allows the user to invoke a process with a set of arguments. When a process is invoked, it returns a reference to its output stream (line 14). The events of an output stream are tagged according to their types: `out` for successful executions (line 13, 17), `err` for erroneous executions (line 25), and `close` to signal the end of stream. Thus, users can listen to these event types using the `on` construct (line 13, 17, 25), then react according to the event type. Process invocations separated by semi-colons are executed in an asynchronous manner. As such, `p1.invoke(); p2.invoke();` represents a parallel execution of both `p1` and `p2` processes. If a sequential ordering is required, `on` blocks can be nested (line 13, 17). Due to the intrinsic nature of the web, some requests may never receive a response. In that case, we make sure to set a timeout on aggregation operations, and flush the memory if the aggregated services take too much time to respond.

Additionally, users may need to aggregate data from different sources before performing an action. For this purpose, we introduce the `and` operator. It allows users to express synchronization points when dealing with asynchronous processes. The `and` operator is implemented as a built-in process that generates an output event only when it receives an event from both its two input streams. Incoming events are buffered in a circular FIFO memory enabling the runtime to provide load shedding by discarding events that occur more frequently from one source than the other. For each discarded event, an error event is generated on the output stream allowing the composition to react to it.

The language also provides basic control flow constructs, with the `if/else` keywords. These constructs provide filtering capabilities on data from output events and can be used to conditionally execute a branch of the program. For example, Fig. 2 (line 18) shows how to express the invocation of the `sendEmail` process only when the value of the priority field is high.

Core Services. In addition to the set of operators provided by the language grammar, we provide a wide range of core services to facilitate the use of the language and enrich the expressiveness of compositions. For instance, a `require` function (Fig. 2, lines 2–9) is available globally and serves as an import mechanism for instanciating processes. `require` returns a new instance of the specified process. Processes are looked up by name and loaded from MEDLEY's internal repository which includes a set of predefined services. For instance, to periodically check the existence of a new issue, a predefined process `Medley/Tick` is used, which emits tick events at a predefined frequency.

Providers. In MEDLEY, integration of third-party services is achieved through process providers. A process provider is in charge of developing the interaction logic with the desired service, by implementing a MEDLEY process, and then deploying it in MEDLEY's process repository, in a plugin-like fashion. Once deployed, the process is indexed and becomes available for use on the platform (Fig. 2, lines 2–9).

```
comp     ::= composition { decl+ rule+ }
decl     ::= pool? process ident = require ( string );
           | ident.init ( json? );
           | ident.add ( ident* (, ident)* );
rule     ::= on event do { action+ }
event    ::= evt | event and evt | ( event )
evt      ::= evt_kind (as ident)?
evt_kind ::= ident : out | ident : err | ident : close
action   ::= stream ident = ident.invoke ( json? ) ;
           | ident.invoke ( json? ) ;
           | if ( expr ) action (else action)?
           | rule
expr     ::= ! expr | expr binop expr | ( expr )
           | ident | string | integer | float | jsonpath
           | method ( expr* (, expr)* )
jsonpath ::= {{ string }}
method   ::= ident | jsonpath . ident
binop    ::= < | > | <= | >= | == | != | && | ||
```

Fig. 3. Subset of the DSL grammar

Process Pools. We also define a construct to specify pools of interchangeable processes, using the `pool` keyword. More specifically, it consists in a set of processes that share a common interface, and are semantically equivalent (i.e. they can fulfill the same functional need, such as sending an email. See Fig. 2 for an example). A process pool is typically used to allow a composition to dynamically adapt to service outages, all while being transparent to the developer.

3.3 Data Processing

A crucial aspect in composing multiple web services is being able to reuse and pass data from a service to another. In our DSL, we provide the necessary mechanisms to have fine-grain control over the data, such as on-the-fly substitution and evaluation expressions, as well as document traversal and templating.

Substitution. To extract data from inbound events, we use JSONPath [12] expressions. JSONPath is the XPath [30] equivalent for JSON. It provides a set of operators to traverse JSON documents from their root (noted as `$`), and selectors to match queries on document attributes. In the snippet presented in Fig. 2 (line 21, 28), we use the double curly braces notation `{{...}}` as delimiters for JSONPath expressions. The document root `$` represents the payload of the incoming event. At runtime, these expressions are evaluated: placeholders are replaced with the corresponding values. To enrich the expressiveness of event processing, we provide an additional set of primitive methods on JSONPath expressions, to perform arithmetic operations and text processing, such as `contains`, `startsWith`, `endsWith`, etc.

Evaluation. In addition to data substitution, we also provide an environment for evaluating expressions on JSON primitive types. The evaluation environment is accessed through `<@ expr @>` delimiters, where `expr` is the expression to evaluate. As such, users can easily manipulate and transform data through evaluated expressions. At runtime, a pre-processing phase takes place, where JSONPath expressions are first substituted with the appropriate values, and then eval environments are evaluated.

4 Implementation

Our implementation of Medley comprises a compiler for the Medley domain-specific language and a runtime system. The runtime system relies on Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine. From the Medley specification of an orchestration, the compiler generates JavaScript code that can then be linked with the runtime system. The generated code runs on devices ranging from desktop computers to resource-constrained devices such as home appliances. The runtime system defines various utility functions and amounts to about 1,200 lines of JavaScript code. The Medley compiler is around 600 lines of code. We first describe the main challenges in code generation, and then present the runtime system.

4.1 Code Generation

The main challenges in generating code from a Medley specification are the propagation of data through subsequent process invocations, and the routing of events through publishers and subscribers.

Data Propagation. An orchestration usually defines a hierarchy of handlers, the actions inside an `on` clause. Code inside a handler can access not only the data associated to its input event but also its inner events. Figure 4 shows an example of orchestration in which a handler manipulates data (line 4) associated to one of its inner events (line 1). Because each process invocation is asynchronous, data associated to events must be maintained across multiple invocations, resulting into a hierarchy of data. Maintaining data hierarchy can, however, have serious performance penalty. Furthermore, propagating the whole payload of an event might not be necessary when only a subset of the data is required at a later stage.

The Medley compiler implements a backward dataflow analysis to identify data fragments that must be maintained across multiple process invocations.

```
1   on (foo:out as f) do {
2     stream bar = getBar.invoke();
3     on (bar:out) do {
4       p.invoke({"id": "{{$.f.id}}"});
5   } }
```

Fig. 4. Hierarchy of handlers

$$\frac{e = \langle l, d, \delta \rangle \qquad \vDash \texttt{on}\ (e)\ \texttt{do}\ \{stmt_1; \ldots; stmt_n\}}{e \vDash action_1 \quad \ldots \quad e \vDash action_n} \tag{1}$$

$$\frac{e_1 = \langle l_1, d_1, \delta_1 \rangle \qquad e_2 = \langle l_2, d_2, \delta_2 \rangle \qquad \vDash \texttt{on}\ (e_1\ \texttt{and}\ e_2)\ \texttt{do}\ \{stmt_1; \ldots; stmt_n\}}{\begin{array}{c} e_1 \Rightarrow \langle and_{in}, \{(l_1, d_1)\}, \delta_1 \rangle \qquad e_2 \Rightarrow \langle and_{in}, \{(l_2, d_2)\}, \delta_2 \rangle \\ and_{out} \vDash stmt_1 \quad \ldots \quad and_{out} \vDash stmt_n \end{array}} \tag{2}$$

$$\frac{e = \langle l, d, \delta \rangle \qquad e \vDash p.\texttt{invoke}(j)}{e \Rightarrow \langle p_{in}, j, \delta \cup \{(l, d)\} \rangle} \tag{3}$$

$$\frac{e = \langle l, d, \delta \rangle \qquad e \vDash \texttt{stream}\ s = p.\texttt{invoke}(j)}{e \Rightarrow \langle p_{in}, j, \delta \cup \{(l, d)\} \rangle \qquad p_{out} = \langle l', d', \delta' \rangle \qquad p_{out} \Rightarrow \langle s, d', \delta' \rangle} \tag{4}$$

Fig. 5. Rewrite rules for event routing

These data fragments are implemented as an environment structure that is added to the event payload. Processes forward this environment from their input channel to their output channel, adding information only when it may be required at later stage. To reduce memory footprint, the environment structure contains only references to data stored inside a global environment maintained by the runtime system. The MEDLEY developer does not need to be aware of these details.

Event Routing. Each process in MEDLEY has its own input channel for listening to events and output channel for publishing events. Events associated to a process are isolated in the namespace of the process, preventing them from interfering with other processes. To implement the logic described in the MEDLEY specification, the compiler generates a set of rewrite rules. Rewrite rules are used to intercept events, rename them, and publish them under a new event name. Rewrite rules are described as inference rules with a sequence of premises above a horizontal bar and a judgment below the bar (see Fig. 5). An event is described as $\langle l, d, \delta \rangle$, where l is the label name of the event, d the data associated to it, and δ the environment structure of the call hierarchy. A rewrite rule of the form $e_1 \Rightarrow e_2$ means that once the event e_1 occurs, the runtime system raises the event e_2. A judgment of the form $e \vDash stmt$ means that the runtime systems interprets the statement *stmt* when the event e occurs. In other words, *stmt* is the callback associated to e. The second rule shows how MEDLEY implements the and operator by rewriting each event into the input event of the and process. This process is provided as a builtin process. When it receives both the events $\langle and_{in}, \{(l_1, d_1)\}, \delta_1 \rangle$ and $\langle and_{in}, \{(l_2, d_2)\}, \delta_2 \rangle$ on its input channel, it generates the event $\langle and_{out}\{(l_1, d_1), (l_2, d_2)\}, \delta_1 \cap \delta_2 \rangle$ on its output channel. The third and fourth rules are for invoking a process p. In that case, we rewrite the event e that trigger the invocation of p as the input event of p.

4.2 Runtime System

The runtime system relies on Node.js as the backing messaging system. It encapsulate each composition in a scoped environment by assigning it a unique namespace. Therefore, events generated within a composition are restricted

to their composition scope, and cannot leak over to other compositions. The runtime system is also responsible of managing the lifecycle of a process. It provides basic operations to initialize, start, stop and destroy a process instance. The `init` operation is usually used to pass user credentials to the process instance so as to interact with third-party services.

Our current implementation supports most of client authentication methods ranging from HTTP Basic Auth [9], API keys, to OAuth protocols. To handle these authentication mechanisms, MEDLEY provides a dedicated user interface through which users can authorize third-party services by providing their credentials. The runtime system also supports OAuth 2.0 refresh tokens (used in Google APIs, for example). Refresh tokens are short-lived access tokens that expire after a predefined amount of time. A process can not be started unless all its credentials have been correctly set.

During its lifecycle, a composition may have to handle several kind of errors. A process may throw an error on its output channel (events of type `err`) based on its internal implementation. An error may indicate that a request to a third-party service has failed, that authentication has failed or any other service specific errors. These errors are reported as events and thus are accessible at the language level. Therefore, users can describe in their orchestration their own error handling policies. In addition, the runtime system catches errors such as network failures. In that case, it rolls back the failed process and tries it again later, increasing the time interval between each successive retry. When too many errors are raised by a composition, the system may decide to kill the running instance and release corresponding resources.

5 Evaluation

To assess our approach, we first present a performance evaluation of our implementation and then describe a comparative study of the supported features.

5.1 Performance

The MEDLEY specification used for our experiments is depicted in Fig. 6. It consists in periodically polling a stock exchange service for a quote, and notifying the user by SMS if the value of the stock quote is above 100 USD. The period corresponds to the time elapsed between two successive executions of a composition. To measure the intrinsic scalability of our implementation, the processes used in our experiments do not actually communicate with third-party services. Instead, we simulate real-world latency by defining a randomized delay for response times between 50 and 100 ms. Similarly, we mock the behavior of the stock exchange service. The value it returns is randomized and varies between 80 and 120 USD.

We run our experiments on two different kinds of hardware platforms, from embedded devices to mainstream servers. The server we use is powered by 2 quadcore AMD Opteron 4386 CPUs at 3 GHz and 16 GB of RAM. We configure our runtime system to use a pool of 7 working threads, and one thread for

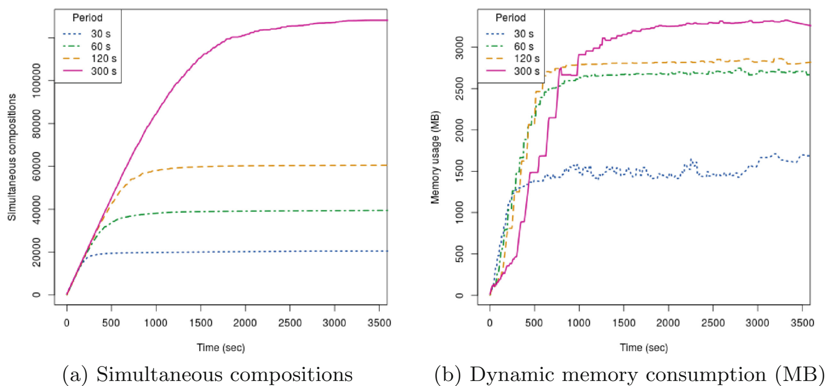
```
1  composition {
2   process tick = require("Mock/Tick");
3   process getQuote = require("Mock/GetQuote");
4   process sendSms = require("Mock/SendSms");
5   // ...
6   on (tick:out) do {
7     stream quote = getQuote.invoke({ "symbol": "MSFT" });
8     on (quote:out as q) do {
9       if ({{"$.q.value"}} > 100) {
10        sendSms.invoke({
11          "text": "Current price: {{$.q.value}}",
12          "number": "+33601234567"
13        });
14 } } } }
```

Fig. 6. MEDLEY specification of the stock exchange composition

the main process. Therefore, we allocate one thread on each physical core of the server. We increase the memory limit of our underlying execution engine to 4 GB which is its current maximum on 64-bit systems. As an embedded system candidate, we use the Raspberry Pi 2 model B with 1 GB RAM and 1 quadcore BCM2836 CPU. We configure our runtime system to use a total of 4 threads, mapping each of them on a physical core. We raise the memory limit to 1 GB, which is the maximum of memory available on this device.

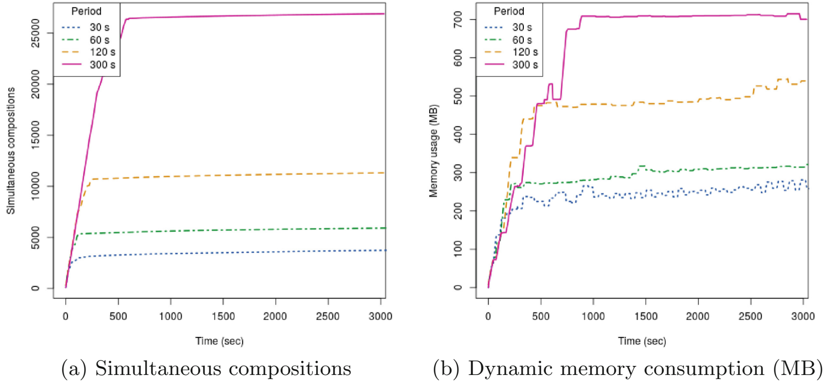
Our benchmarks measure the memory footprint of the MEDLEY runtime when gradually increasing the number of simultaneous compositions. We perform a staged rollout by instanciating and starting a new composition every 10 ms, and collect a snapshot of memory usage every second. The period used in our experiments vary from 30 s to 5 min. A small period increases responsiveness but requires much more resources as the composition needs to be executed more often.

Performance results on the server are shown in Fig. 7 while those for the embedded device are shown in Fig. 8. On the server, the total number of simultaneous compositions varies from at least 22,000 with a period of 30 s to up to 125,000 with a period of 5 min. Similarly, the Raspberry Pi 2 enables at least 4,000 simultaneous compositions with a period of 30 s to up to 27,000 with



(a) Simultaneous compositions (b) Dynamic memory consumption (MB)

Fig. 7. Benchmark results on a server



(a) Simultaneous compositions (b) Dynamic memory consumption (MB)

Fig. 8. Benchmark results on an embedded device

a period of 5 min. When the period is too small or the number of simultaneous compositions is too high, the event queue of the runtime becomes full and no composition can be instantiated anymore. We are currently investigating the scheduling of the various compositions out of the main thread to significantly increase the scalability of the runtime. As illustrated in Figs. 7b and 8b, the memory consumption of the runtime follows the same growth as the number of simultaneous compositions. In the worst case, the runtime consumes up to the total of memory allocated to it. Our current implementation relies on Node.js which limits the memory of a single process to 4 GB. However, as compositions are independent from each others, it would be possible to increase the number of simultaneous compositions by distributing them over a cluster of several instances of Node.js processes.

5.2 Features

We present a comparative study of the features supported by Medley compared to Bite [27], S [3] and the WS-BPEL standard [20]. We select these solutions because they address the problem of composing web services and provide a language to describe such compositions. We rely on the work of Sheng et al. [28] to identify the following features:

- *Dynamic typing*: the ability to manipulate arbitrarily-typed data structures.
- *Dynamic service selection*: the ability to select and bind services at runtime.
- *Exception handling*: the ability to handle and respond to runtime errors.
- *Hybrid service support*: the ability to compose services of different types (REST, SOAP, etc.).
- *Language extensibility*: the ability to extend the language and provide new features.
- *Scoping*: the ability to define and use nested blocks and localized variables.

Table 1. Comparison of features

	Medley	Bite	S	WS-BPEL
Dynamic typing	+	+	+	−
Dynamic service selection	+	−	−	−
Exception handling	+	∼	∼	+
Hybrid service support	+	−	−	∼
Language extensibility	+	+	−	−
Scoping	+	−	+	+

(+) Supported, (−) Not supported, (∼) Partial support.

Table 1 summarizes the results of our comparative study. All approaches support dynamic data typing except for BPEL, where data types are defined by their corresponding WSDL interface. Furthermore, even though all solutions enable static binding of services, MEDLEY also provides a construct to handle pools of services, enabling dynamic binding based on user-defined strategies. All four solutions also support handling runtime exceptions, although at different levels. For instance, Bite enables defining exception handlers at the activity and composition levels, while S just relies on standard error handlers provided by the JavaScript language. On the other hand, MEDLEY enables reacting to error events from the output streams of the invoked processes. As for the supported types of web services, they all enable composing RESTful services except BPEL, even though recent works aim to address this aspect by proposing extensions to BPEL. Moreover, since services are wrapped and exposed as processes in MEDLEY, we can easily integrate other types of web services such as SOAP. Since the adaptation is handled at the process level (by the process provider), it is transparent at the language level, enabling the composition of hybrid services. Regarding language extensibility, MEDLEY can be easily extended by implementing new processes, whereas the same can be achieved in Bite by implementing new activity types, allowing further customization of these languages. This aspect is not covered in S and BPEL. Table 1 also shows that scoping is supported by all solutions except Bite, since it relies on a lightweight composition model.

6 Related Work

Ever since service-oriented architectures (SOA) emerged, the aspect of composing web services became prevalent and necessary. To this end, several languages and tools have been developed [28]. Among them, the most popular is WS-BPEL (Business Process Execution Language) [20]. It provides a model for describing the behavior of a composition based on its interactions with the composed services. BPEL is standardized by the OASIS organization since 2004 and relies heavily on WSDL [4] interfaces to define links with partner services. However, not all web services are exposed through a WSDL interface.

Nowadays legacy web services are rapidly decaying, in favor of the more flexible REST architecture style [7]. Although REST became the building block for major service providers, it lacks an official standard for describing interfaces. Some initiatives including Swagger [29], WADL [11] and RAML [26] have been proposed for describing REST interfaces but they are all far from wide adoption by the majority of service providers. A consortium of several major API vendors came together to found the OpenAPI Initiative [21] (founded in November 2015), as an effort to standardize how REST APIs are described. Therefore, there is a fundamental mismatch between the REST architectural style and SOA orchestration solutions, since these solutions are not directly applicable [32].

Several efforts have been made to support composition of RESTful services. Some approaches such as Bite [5,27] and S [3] define a domain-specific language to express compositions. Bite follows a workflow model while S is an extension of JavaScript. Both of them require services to be statically binded and provide limited support for error handling. As opposed, Medley has the notion of pool of services that enables dynamic binding based on user-defined strategies. This mechanism also improves robustness by providing fault-tolerance to service unavailability. Other approaches propose to extend BPEL by addition new activities to manipulate REST resources as first-class entities [22,24]. However, in practice, popular BPEL orchestration engines have limited support for composing REST services.

Among existing solutions, some tackle service composition using a goal-driven semantic approach [13,16,31]. They rely on ontologies and on reasoning engines to dynamically select services that fulfill the user-provided requirements. However, very few REST services have a well-defined semantic description thus limiting the applicability of these techniques in practice.

In the commercial world, several SaaS (Software-as-a-Service) solutions and integration platforms have been built around the concept of composing these emerging services, providing user-friendly web applications in which users can describe simple orchestration scenarios. For instance, Zapier[3] and IFTTT[4] allow end-users to express compositions as pairs of *(trigger, action)*, such as *"on trigger do action"*. However, *action* is limited to one per *trigger*, which hinders the expression of more complex scenarios. Workato[5], Azuqua[6], Node-RED[7] and NoFlo[8] on the other hand do not have this restriction, and enable users to express more complex compositions. However, these platforms do not provide error handling mechanisms to the users; they merely log the errors encountered. Node-RED provides a generic *"catch all errors"* node, that serves as a single access point for all errors. On the other hand, Medley provides error output streams on a per-process basis, which allows for localized error handling.

[3] https://zapier.com/.
[4] https://ifttt.com/.
[5] https://www.workato.com/.
[6] http://azuqua.com/.
[7] http://nodered.org/.
[8] http://noflojs.org/.

Additional processes can be attached to these error streams to execute error handling logic, such as invoking other services.

Nonetheless, most of these solutions may not be suitable for large businesses or organizations which handle sensitive and business-critical data. Instead, the emergence of private personal clouds [18] enables these actors to deploy a secure environment, where sensitive data is not exposed over external services. Further works investigate how to enforce access-control levels on data at a fine grain [2]. Medley being lightweight and embeddable enables these actors to deploy private orchestration platforms, granting them control over the confidentiality of data and business processes.

As for the existing aforementioned orchestration languages, Medley draws its inspiration from several existing concepts, such as flow-based programming, and process algebras. Medley applies both of these concepts to the particular context of microservice composition. The notion of Flow-Based Programming (FBP) was first introduced by John Paul Morrison in the early 1970s [17]. FBP introduces the concepts of processes, bounded buffers, information packets, named ports, and separate definition of connections. FBP views an application as a network of asynchronous processes communicating by means of streams of structured data chunks known as *information packets*. Information packets are passed between the inputs and outputs of processes. Each process may have multiple inputs and outputs, and multiple processes may be connected to a specific inport or outport. FBP encourages loose coupling of components, relying on linking black boxes in order to build microservice architectures. This approach is applied in Medley, complemented by an event-driven communication layer.

Process algebras are abstract languages used to specify the execution of concurrent processes. Languages like FSP, CSP, LOTOS provide the necessary semantics to express interactions (emission, reception) between two or more processes [6,8,14]. These formalisms are founded on algebraic laws, enabling one to reason formally on a system and perform various model-checking techniques to verify properties, variants and invariants of said system. Medley reuses process algebra principles in a more concrete manner to express dataflow between third-party services. Medley defines the required mapping between the user-provided input and the system input, and enables reasoning on data types and type compatibility, which is otherwise not possible.

7 Conclusion and Future Work

In this paper, we have presented Medley, an event-driven lightweight platform for service composition. Medley is based on a domain-specific language for describing orchestration and a compiler that produces efficient code. We have used Medley to develop various compositions, involving a large number of existing services. Generated compositions consume a reasonably low amount of resources and the platform scales well both on a mainstream server and an embedded device such as a Raspberry Pi. Compared to traditional approaches based on BPEL or ESB, Medley enables smooth adaptation at run-time of

compositions of services based on their availability. In addition, we show through several examples that Medley raises the level of abstractions enough to hide to the programmer intricacies of underlying communication paradigms. The Medley platform is currently under beta test and will be shortly distributed as a product of CProDirect. We are working on a visual editor on top of the Medley language for defining orchestrations.

There are a number of interesting avenues of future work. The first is to extend the language to specify when a change of a remote resource has to be reported as a new event in the case of polling. In this regard, we are currently defining new algorithms to efficiently compute diffs of XML or JSON documents. Complementary to this, we are investigating dataflow analyses of orchestrations to detect compositions that may expose sensitive data to unauthorized users.

Acknowledgment. This work was partially supported by CProDirect and the French funding agency ANRT under contract CIFRE-2013/0891.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Data-Centric Systems and Applications. Springer, Heidelberg (2004)
2. Biswas, P., Patwa, F., Sandhu, R.: Content level access control for openstack swift storage. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 123–126. ACM (2015)
3. Bonetta, D., Peternier, A., Pautasso, C., Binder, W.: S: a scripting language for high-performance RESTful web services. ACM SIGPLAN Not. **47**(8), 97–106 (2012)
4. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., et al.: Web services description language (WSDL) 1.1 (2001)
5. Curbera, F., Duftler, M., Khalaf, R., Lovell, D.: Bite: workflow composition for the web. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 94–106. Springer, Heidelberg (2007)
6. Ferrara, A.: Web services: a process algebra approach. In: Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 242–251. ACM (2004)
7. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
8. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Tool support for model-based engineering of web service compositions. In: Proceedings of the 2005 IEEE International Conference on Web Services, ICWS 2005, pp. 95–102. IEEE (2005)
9. Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: Rfc 2617: Http authentication: Basic and digest access authentication (1999). https://tools.ietf.org/html/rfc2617
10. Fuchs, A., Gürgens, S.: Preserving confidentiality in component compositions. In: Binder, W., Bodden, E., Löwe, W. (eds.) SC 2013. LNCS, vol. 8088, pp. 33–48. Springer, Heidelberg (2013)
11. Hadley, M.J.: Web application description language (wadl) (2006)

12. JSONPath. http://goessner.net/articles/JsonPath/. Accessed 05 September 2015
13. Klusch, M., Gerber, A.: Fast composition planning of owl-s services and application. In: 4th European Conference on Web Services, ECOWS 2006, pp. 181–190. IEEE (2006)
14. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In: Donohoe, P. (ed.) Software Architecture. IFIP, vol. 12, pp. 35–49. Springer, New York (1999)
15. Maximilien, E.M., Wilkinson, H., Desai, N., Tai, S.: A domain-specific language for web APIs and services mashups. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 13–26. Springer, Heidelberg (2007)
16. Mayer, S., Inhelder, N., Verborgh, R., Van de Walle, R., Mattern, F.: Configuration of smart environments made simple: combining visual modeling with semantic metadata and reasoning. In: 2014 International Conference on the Internet of Things (IOT), pp. 61–66. IEEE (2014)
17. Morrison, J.P.: Flow-Based Programming: A new approach to application development. CreateSpace (2010)
18. Na, S.H., Park, J.Y., Huh, E.N.: Personal cloud computing security framework. In: 2010 IEEE Asia-Pacific Services Computing Conference (APSCC), pp. 671–675. IEEE (2010)
19. Newman, S.: Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly Media, Sebastopol (2015)
20. OASIS: Web services business execution language version 2.0 (2007)
21. OpenAPI. https://openapis.org/. Accessed 14 January 2016
22. Pautasso, C.: BPEL for REST. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 278–293. Springer, Heidelberg (2008)
23. Pautasso, C.: On composing RESTful services. Software Service Engineering (09021) (2009)
24. Pautasso, C.: RESTful web service composition with BPEL for REST. Data Knowl. Eng. **68**(9), 851–866 (2009)
25. Pautasso, C., Alonso, G.: Flexible binding for reusable composition of web services. In: Gschwind, T., Aßmann, U., Wang, J. (eds.) SC 2005. LNCS, vol. 3628, pp. 151–166. Springer, Heidelberg (2005)
26. RAML. http://raml.org/. Accessed 05 September 2015
27. Rosenberg, F., Curbera, F., Duftler, M.J., Khalaf, R.: Composing restful services and collaborative workflows: a lightweight approach. IEEE Internet Comput. **12**(5), 24–31 (2008)
28. Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X.: Web services composition: a decade's overview. Inf. Sci. **280**, 218–238 (2014)
29. Swagger. http://swagger.io/. Accessed 05 September 2015
30. Urpalainen, J.: An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors (2008)
31. Zhao, H., Doshi, P.: Towards automated restful web service composition. In: IEEE International Conference on Web Services, ICWS 2009, pp. 189–196. IEEE (2009)
32. Zur Muehlen, M., Nickerson, J.V., Swenson, K.D.: Developing web services choreography standards—the case of rest vs. soap. Decis. Support Syst. **40**(1), 9–29 (2005)