Control-Flow Patterns for Decentralized RESTful Service Composition

JESUS BELLIDO and ROSA ALARCÓN, Pontificia Universidad Catolica de Chile
CESARE PAUTASSO, University of Lugano

The REST architectural style has attracted a lot of interest from industry due to the nonfunctional properties it contributes to Web-based solutions. SOAP/WSDL-based services, on the other hand, provide tools and methodologies that allow the design and development of software supporting complex service arrangements, enabling complex business processes which make use of well-known control-flow patterns. It is not clear if and how such patterns should be modeled, considering RESTful Web services that comply with the statelessness, uniform interface and hypermedia constraints. In this article, we analyze a set of fundamental control-flow patterns in the context of stateless compositions of RESTful services. We propose a means of enabling their implementation using the HTTP protocol and discuss the impact of our design choices according to key REST architectural principles. We hope to shed new light on the design of basic building blocks for RESTful business processes.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia

General Terms: Design, Standardization

Additional Key Words and Phrases: Web services, service composition, REST, control flow, business processes, control-flow patterns

ACM Reference Format:
Bellido, J., Alarcón, R., and Pautasso, C. 2013. Control-flow patterns for decentralized RESTful service composition. *ACM Trans. Web* 8, 1, Article 5 (December 2013), 30 pages.
DOI: http://dx.doi.org/10.1145/2535911

1. INTRODUCTION

A REST architecture is defined by a set of architectural constraints that aims to guarantee the scalability of the interaction between architectural components, the uniformity of the interfaces between such components, and its independent evolution [Fielding 2000]. REST's central element is the *resource* consisting of server-side conceptual entities that can be globally addressed and referenced through URIs and whose *state* is passed to clients through *representations*, encoded in various *media types* (e.g., HTML) [Richardson and Ruby 2007]. A REST service can be seen as a set of such resources that provide coherent access to the state and the functionality of a software component published on the Web. Traditionally, Web services are described by a WSDL

This work is supported by the Center for Research on Educational Policy and Practice (CONICYT), Grant 11080143.
Authors' addresses: J. Bellido (corresponding author) and R. Alarcón, Computer Science Department, Pontificia Universidad Catolica de Chile, Av. Vicuna Mackenna 4860, Santiago, Chile; email: jbellido@uc.cl; C. Pautasso, Faculty of Informatics, University of Lugano (USI), via Giuseppe Buffi 13 CH-6904, Lugano, Switzerland.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1559-1131/2013/12-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2535911

document and use the SOAP [Mitra and Lafon 2010] protocol to communicate. These WSDL/SOAP services can be arranged into composite services that enable a business process executed by a process engine [Pautasso 2009c]. The business process defines the set of interactions among multiple services that are required to achieve a goal; services' interactions (i.e., machine-to-machine) are regulated through simple and complex control patterns (e.g., branches, parallel flow, sequential invocation, discriminator) that determine the partial order of the service's operations invocation [Hamadi and Benatallah 2003; Russell et al. 2006].

From the control-flow perspective, service composition is seen as an orchestration in which the coordination of the control flow is centralized in a single component (e.g., the composed resource behaves as an orchestrator), or as a choreography, in which control flow is distributed among a set of participant services. Concerning the state of the composed service, we distinguish *stateful* composed services if the information about the progress of the interaction with the participant services is kept locally on the composed service, or *stateless* if the composed service does not maintain local state, but instead maps a user agent's requests to origin services directly [Pautasso 2009b]. Current proposals for RESTful service composition are mainly stateful.

Control flow in a RESTful system is driven by the choices performed by humans through flexible user interfaces and clients (e.g., Web browsers). This approach, also known as *follow your nose*, is possible because the resource's representations include the required *controls* to change the state of the resource (e.g., to post an updated state) or the *links* to retrieve related resources. Resources are connected through controls and links, resulting in a hypermedia graph that determines the set of possible state transitions (*hypermedia* constraint). The semantics of such links and controls, however, can be only understood at the application domain level (e.g., a POST control could imply a payment placement), so that when machine-clients instead of humans must choose which links or actions to follow, this becomes a nontrivial task.

Despite the possibility of explicitly annotating links in order to make clear their purpose, the semantics of such annotations still require a shared understanding for the machine-clients that participate in a service composition, increasing then the coupling between components. In addition, since resource providers keep control of the resources, they can evolve independently of the client's expectations, that is, resource URIs, supported methods, representations, and possible state transitions (i.e., links and controls) can change unexpectedly. Hence, clients must minimize their assumptions about the resources (e.g., about URIs or URI template structures) and how they are related (e.g., the underlying hypermedia).

Traditional WSDL/SOAP-based service control-flow patterns rely on an operation centric and a centralized style that doesn't comply with REST constraints. By implementing such control-flow patterns in compliance with REST constraints, we present a novel perspective on how to provide fully decentralized support for control flow. In this article, we present a subset of the control-flow patterns well known in the business process management community [van der Aalst et al. 2003] in a way that supports a composition style for RESTful services that is fully decentralized, hypermedia-aware, and stateless. In order to maintain a loose coupling between the resources participating in a REST service composition, we minimize the shared understanding of components by placing control-flow semantics at the protocol level through extensions to the HTTP protocol *status codes*. We rely also on minimal ReLL [Alarcón and Wilde 2010] descriptions and a fully decentralized model based on callback connectors that allow us to implement stateless REST service composition. A realistic scenario based on long-running business processes is presented to illustrate the advantages of our approach; we also discuss our design considering its impact on the key architectural properties of REST.

This article is organized as follows: Section 2 introduces basic concepts regarding REST services composition; Section 3 presents a motivating scenario and the rationale of our approach. Section 4 presents our proposal for a set of basic and advanced control-flow patterns for REST services. Our reference implementation is described in Section 5, while Section 6 presents a comparative evaluation of different QoS attributes. Section 7 discusses the impact of our approach on key REST architectural properties. Finally, Section 8 presents our conclusions and proposals for future work.

2. BACKGROUND

REST architectural components comprise origin servers, gateways, proxies, and user agents that are associated through connectors, such as clients and server interfaces, caches, resolvers, and tunnels. A REST architecture is determined by the roles of the architectural components, their limitations, and their behavior when they interact with each other, instead of determining the component's implementation details or the protocol syntax. The cornerstone design element of REST is the resource. Resources provide a globally addressable and uniformly accessible abstraction over a service's data and functionality. Resource's states are transferred across architectural components through representations; components operate on the resources through the metadata information (e.g., headers), links, and controls (e.g., a form allowing to POST new information) embedded in the representations [Fielding 2000]. Resources' links and controls give shape to a distributed hypermedia graph that determines the set of possible actions and state transitions that user agents can perform.

Traditional Web service composition is based on the availability of endpoints that expose the service's interface but hide its implementation, invocation effects, and semantics. With RESTful services, the implementation, effects, and semantics instead are fully exposed through links and standardized operations. Thus there is a need to research how to avoid violating the basic constraints (e.g., statelessness, uniform interface, and hypermedia) and principles (e.g., dynamic binding) of REST when composing services.

Control-flow patterns are the basic building blocks in traditional service composition but are also conceived for stateful, centralized workflows that compromise scalability and loose coupling [Pautasso and Wilde 2009]. In this article, we extend the current research on RESTful service composition by presenting control-flow patterns designed in a way that complies with RESTful constraints by exploiting link processing (see Section 2.2) and HTTP interactions (see Section 2.3). Our approach is both stateless and decentralized (as exemplified in Section 3), whereby the state of the composed service and the responsibility for interacting with the participant services are deferred back to the user agents.

In order to comply with the REST constraints, we avoid introducing additional architectural components but propose to extend the uniform interface, (specifically the HTTP redirection codes) at the protocol level, since this approach allows normative organizations to introduce standards for the behavior of user agents without breaking current implementations. In addition, for fully automatic RESTful service composition, links semantics need to be understood at the application domain level so that machine-clients can choose the proper operation. Unfortunately, there is not yet an agreement on a way to convey such semantics. In this article, we rely on ReLL (Section 2.4), which is a hypermedia-centric description of RESTful services.

2.1. Service Composition in REST

Service composition is the process that assembles component services into new, composed services which can be recursively used as components for other services [Benatallah et al. 2003; Peltz 2003]. The result of composing REST services is a new

REST service that behaves as an intermediary between the user agents that consume it and the origin services that provide the REST service components [Pautasso 2009b]. REST service composition poses additional challenges with respect to traditional Web service composition, for instance, *dynamic late binding*, that is, binding the resources to the composed service at runtime, must be supported, since actual resources' URIs can only be discovered when inspecting the corresponding representations.

JOpera is one of the most mature platforms for supporting REST services composition; it satisfies most REST service language composition requirements [Nierstrasz and Meijler 1995; Pautasso 2009a]. Visual editors supports the design of manual, centralized, stateful service composition that can be executed by an orchestration engine; it also addresses control- and dataflow as well as data transformations, and the resulting service composition is written as a BPEL extension for REST [Pautasso 2009c]. Similarly, Bite [Rosenberg et al. 2008] proposes a BPEL-inspired workflow composition language describing both control- and dataflow. Bite can mint URIs for resources created but cannot inspect representation content and selectively retrieve the URIs served by the service and support the hypermedia constraint.

Decker et al. [2009] present a formal model for REST process enactment based on Petri Nets, PNML (Petri Net Markup Language), and an execution engine. They partially support dynamic late binding by minting URIs for created resources but do not fully support the hypermedia constraint, neither complex guard conditions such as the existence of information stored on the client side (e.g., cookies) nor content negotiation (only XML as media type). Garrote [Hernández and García 2010] proposes a formal model for a semantic REST service inspired by the triple space computing models and process calculus. The calculus describes unambiguously both semantic RESTful resources and composition workflows. The proposal considers the hypermedia constraint but still lacks a typed theory for representing typed patterns (e.g., transactions) and typed resources as well as complex workflow patterns. Zhao et al. [2011] also propose a formalization of RESTful services composition based on linear logic. A set of additional business axioms as well as the selection of the corresponding control-flow pattern (e.g., alternative, sequence, etc.) is defined on design time. The hypermedia constraint and the dynamic late binding property are not addressed at all in the proposal; hence, evolvability of the composed services is severely compromised.

Other approaches rely on Semantic Web technologies. For instance, in Verborgh et al. [2012] invocation of controls can be performed through an N3 extension called REST-desc, and representations served can be later processed. By differentiating links that convey meaning from links that imply interaction (e.g., controls such as a POST), it is possible to build a semantic user agent. Although promising, it is not clear how complex control-flow patterns, other than sequential invocation and conditional invocation, can be represented; or how dynamic binding and hypermedia could be supported. In He et al. [2012], Semantic Web technologies are used to model contextual information from users, sensors, and things so that machine-clients can make sense from the responses. URIs are used to identify abstract concepts and physical objects whose state is read through GET requests. Authors identify sequences (chains of service requests), conditional selection of responses and merging of responses. They also store intermediate states as resources on the server side in order to gain scalability.

2.2. Links Processing

Clients process the links embedded in resource representations in the following order: *protocol*, *hypermedia*, and *application* [Zuzak et al. 2011] (Figure 1). At the protocol level, clients generate new requests based on control data (e.g., a "`303 See Other`" status code), and protocol-level links (e.g., the `Location` header parameter accompanying the status code). At the hypermedia level, clients generate requests based on the

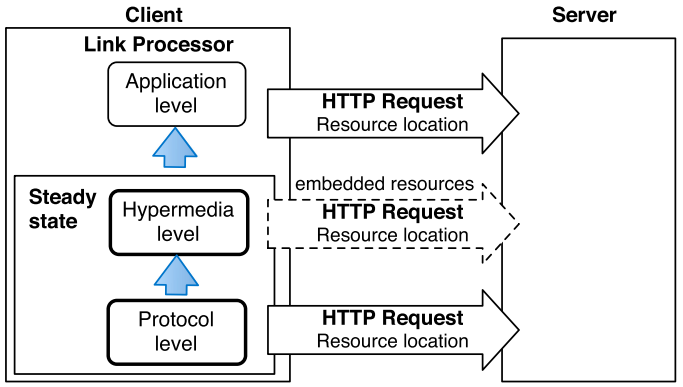

Fig. 1. Links processing levels.

links to resources embedded in the representation (e.g., the `src` attribute of the `<img>`, `<script>` HTML elements), that must be fetched in order to achieve a steady state (i.e. there are no resources pending to be fetched).

Choosing among any of those levels as a foundation for complex control flow has its own trade-offs. For the case of HTTP, extensibility is supported at the level of status codes, media types, code on demand, and metadata (headers). Implementing control flow at the hypermedia-level requires a shared understanding of either the metadata that accompanies links (`rel`), specialized media types (e.g., RDF, EDI, etc.), specialized tags, or content marks (e.g., keywords) for nonstructured content. The resource description framework (RDF) represents Web information in a minimally constraining and flexible way. An RDF expression is a collection of triples. In each triple, a predicate indicates a relationship between subject and object [Beckett and McBride 2004]. Electronic data interchange (EDI) enables technologies for conducting business-to-business transactions according to predefined information format and rules without human interference [Narayanan et al. 2009]. Even though the problem of media types that do not support tags or links can be overcome through Web linking [Nottingham 2010] (link headers), the problem of defining generic data structures and parser rules in order to achieve a shared understanding between clients and servers still remains and introduces stronger coupling between parties.

Since the application level deals with user interaction (or application goals), it will be necessary to rely on such a level if the control operation requires user participation (e.g., choosing an alternative) which also requires the user (or another machine such as a service) to understand the links/control semantics at the application domain level. At the protocol-level, however, the coupling between parties diminishes, since they must agree on the meaning of the protocol itself (i.e., status codes). Since status codes regulate interaction between clients (user agents) and origin servers, it will also be possible for Web intermediaries to handle messages (e.g., for caching, routing, supporting load balancing, etc.), without requiring to parse the message content, that is, visibility of the interaction between components can also be supported. For these reasons, our approach focuses on modeling control-flow patterns at the protocol level.

2.3. HTTP-Based Interaction

Interaction between REST architectural components is based on message interchange, being HTTP (Hypertext Transfer Protocol [Fielding et al. 1999]) the primary protocol, although REST does not restrict communication to a particular protocol. In an HTTP interaction, a client sends a request message to a server which produces a response

message. The first line in the response contains a 3-digit integer (status code) among other information [Fielding et al. 1999].

Status codes are categorized as indicated by its first digit so that architectural components behave in a determined way, for instance, the `3xx` codes are reserved for redirection. Redirection codes allow servers more control of the interaction, since they can guide the client to contact further resources. Unfortunately such redirection is automatically executed only if it was caused by a `GET` or a `HEAD` request; otherwise the user agent needs the end-user confirmation to move forward the request. If the user agent is driven by a machine-client it must either understand the consequences of the redirection at a domain level or blindly follow the redirection. Furthermore, for the case of `POST` operations, a `303 (See Other)` code can be issued indicating that the response can be found under a different URI and should be retrieved using the GET method on such resource so that it will be impossible for a server to instruct the client to issue a `POST` message for a different resource. Unlike `303`, the `307 (temporary redirect)` code allows redirected clients to interact with a new resource using any method; however, it must contain the original information and must be confirmed by the end user. More complex actions (such as a condition evaluation) are not considered.

Since HTTP codes are extensible, a client application may not understand the meaning of an unregistered code (e.g., `353`), but it must understand its category and handle the message accordingly, that is, if the `353` status code cannot be recognized it must be treated like a `300` code (redirection).

2.4. Resource-Oriented Service Description

Ideally, a REST service description must allow clients to deal with (1) dynamic late binding where resource URIs are discovered from a representation at runtime, (2) the uniform interface (i.e., a shared understanding of the interface semantics), (3) dynamic typing through content-type negotiation, and (4) exception handling [Pautasso 2009b]. Existing REST service description proposals (e.g., WADL [Hadley 2009], WSDL 2.0 [Chinnici et al. 2009]), have gained limited acceptance, since they are focused on an operation-centric approach. Current approaches do not consider the uniform interface as a shared, implicit understanding. Instead, they explicitly detail the interaction in such a way that changes on the origin servers break clients so that evolvability is limited due to the introduced coupling. Even though there is still a debate regarding the need for a formal description for RESTful services, the existence of such descriptions may facilitate the automatization of machine-client and RESTful services interaction. Our approach relies on ReLL [Alarcón et al. 2010], a hypermedia-centric service description language with minimal coupling between clients and services [Bellido et al. 2011].

2.4.1. ReLL. The Resource Linking Language describes a set of assumptions that clients expect from servers, such as resources identifiers, representations, links to other resources, controls for state changes, and a mechanism to obtain the links from the representation so that dynamic late binding is possible. A ReLL REST service description is a declarative, partial hypermedia where resources and links are typed so that clients can make sense of the underlying model and navigate by brute force (e.g., a crawler) or purposefully (e.g., following a path towards a goal). The description is intentionally left incomplete to deal with the independent evolution of client and services (e.g., a service provider may change resource URIs, the connection may fail, the response of a message includes an unexpected media types, etc.). Therefore, clients use the description as a navigation map and can fail gracefully in the sense that they must stop their behavior and notice the cause of failure (i.e., an assumption has failed).

```
1  <resource xml:id="products" type="r:catalog">
2     <uri match="((http:\/\/([a-zA-Z0-9\.])+)\/products)" type="regex"/>
3     <representation type="application/xhtml+xml"/>
4     <link location="header" type="r:select" target="cart">
5        <protocol type="http">
6           <request method="post"
7            payload="(productId=\"((http:\/\/([a-zA-Z0-9\.])+)\/product)\/\d+\")
8                    \&(shopper\=\"\d+\")" type="regex"/>
9        </protocol>
10    </link>
11 </resource>

12 <resource xml:id="cart" type="r:shoppingCart">
13    <uri match="((http:\/\/([a-zA-Z0-9\.])+)\/cart)" type="regex"/>
14    <representation type="application/xhtml+xml"/>
15    <link location="header" type="r:inspect" target="productList">
16       <protocol type="http">
17          <request method="get" payload="(shopper\=\"\d+\")" type="regex"/>
18       </protocol>
19    </link>
20    <link location="body" type="r:buy" target="r:Payment">
21       <selector name="varPayment"
22        select="//div[@id='optionsPayment']//:a/@href" type="xpath"/>
23       <protocol type="http">
24          <request method="get" payload="(shopper\=\"\d+\")" type="regex"/>
25       </protocol>
26    </link>
27 </resource>
```

Fig. 2. ReLL description for the resources *catalog* of products and *shoppingCart*.

Partial knowledge in ReLL prevents the automatic generation of client code and hence reduces coupling between clients and servers.

Figure 2 presents an example of a ReLL description for two interlinked resources of type `catalog` and `shoppingCart`. Resources have internal identifiers (e.g., `products` and `cart`) that may differ from their corresponding types, since the former are limited in scope. Lines 2 and 3 indicate a way to validate the expected URI so that a machine-driven user agent could verify at runtime whether a part of the service interface (the URI) has changed by executing a regular expression. Lines 3 and 14 indicate the resources' media types so that a user agent can negotiate the proper representation. A `link` structure is declared in lines 4, 15, and 20. It declares a typed relation (e.g., `select`) between an origin resource (e.g., `product`) and a target resource (e.g., `cart`), as well as the required information for extracting the link (`selector` and `location`), and actually following it (`protocol`). Validation for control payload (e.g., a PUT operation) is supported through a regular expression or an XPath expression depending on the payload data format (Line 21). In Bellido et al. [2011], links and control semantics for choreographies are embedded in Link Headers following the Web Linking standard, under a simple control flow (sequential). For simplicity, in the examples presented in this article, we will use Web Linking for serving the hypermedia controls to user agents.

3. COMPOSING LONG-RUNNING RESTFUL BUSINESS PROCESSES

Synchronous, blocking service invocation implies that clients must wait for a response in order to continue with the interaction. This solution works fine if the client-server connection remains open until the response becomes available; however, for the case of high-latency services, such as those involved in long-running business processes, this is not the case. Long-running business processes are typically executed over a long period of time, involving loosely coupled services that may cross various organizational contexts and are typically coordinated by a third party [Dayal et al. 2001]. Consider the business process shown in Figure 3. In this scenario, the business process of a

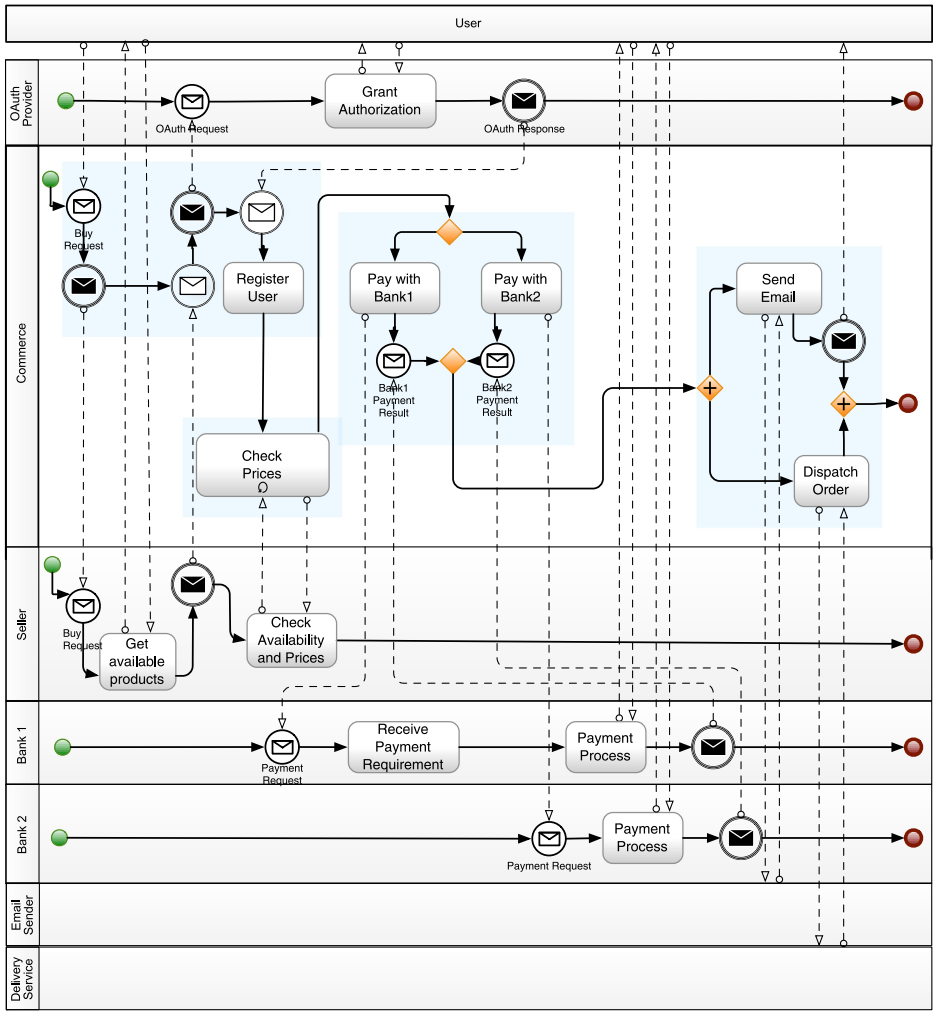

Fig. 3. The `Commerce` REST service composes the REST services Seller, two Banks, a Delivery, Delivery, Email, and Authentication.

certain `Commerce` composes various other processes, `Sellers`, `Banks`, and Delivery organizations, as well as utility services, such as `Authentication` and `Email` services.

The business process begins when a user places a request to get the available products via the commerce service. The user selects the products to buy from the list of available products provided by a Seller service. Once the products are chosen, the user is required to be registered through an OAuth authentication service in two sequential steps. First, the user authorizes the Commerce service to access a subset of his or her data, and in the second step, the authorized Commerce service gets the user's data and generates a Purchase Order, including the chosen products. Once the purchase order has been generated, the total amount is calculated by iteratively adding the value of each item. Following this, the user must choose either one of two payment options offered by the Commerce service, that is, Bank1 or Bank2. Each payment option requires user intervention and occurs out of the conversation band between the Commerce service and the user (i.e., between the Bank service and the user) to

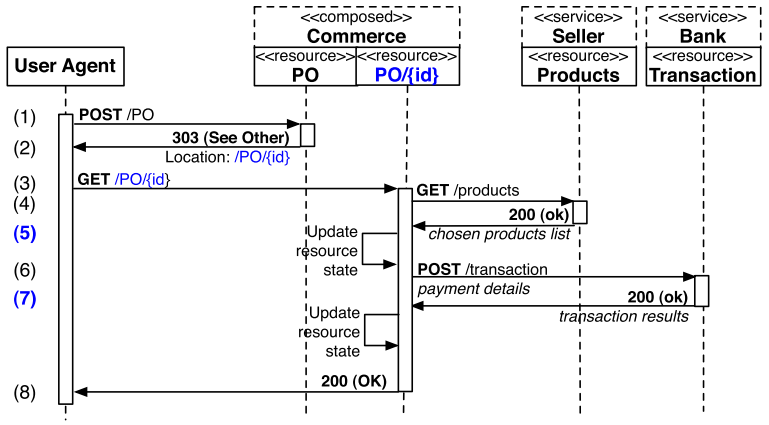

Fig. 4. Centralized implementation of the composed `Commerce` REST service.

guarantee a safe process. Finally, Commerce service user sends an email to support the result of the buying process and simultaneously generates a dispatch order of the purchased products.

In the described process model, activities that require user participation (e.g., products selection, authentication, payment, dispatch confirmation) can also involve a few days for the case of complex transactions (e.g., international shopping) or just because the users availability. In addition, various control-flow patterns are required to coordinate services interaction, such as sequential invocation for OAuth and the business process main activities, iterative invocation to check product prices, conditional invocation depending on the Bank, and parallel invocation of the final process step, delivery and confirmation email.

Traditionally, WSDL/SOAP-based and RESTful services composition is implemented following an orchestration-based centralized approach. This style presents not only limitations for the scalability, performance, and availability of the composed service; it also limits the interaction of the service components, that is, it requires designing components in a way that fits every possible future usage, compromising its evolvability and REST principles. Even though a decentralized approach may favor scalability and loose coupling, there still remains the problem of state handling for the composed service, that is, if control is going to be distributed among the service components, there is still the need to coordinate their behavior so it can comply a determined business process. Sections 3.1 and 3.2 discuss the current approaches when dealing with centralized versus decentralized design and state-handling, respectively. They also explain the trade-offs we face when pursuing a fully decentralized, RESTful compliant service composition. The conclusion of this discussion is presented in Section 3.3.

3.1. Centralized versus Decentralized Control Flow

In Figure 4, an implementation of the proposed scenario relying on a centralized interaction between the parties is shown (the base URI of participants is omitted for readability). We will focus on the Commerce, Seller, and Bank interaction, since they provide enough complexity to illustrate our design choices. The interaction starts when the user places a purchase order by sending a `POST` message to the `PO` resource (1), which is part of the `Commerce` REST service. The server generates a subordinated resource, `PO/{id}` the purchase order, and redirects the user agent to fetch (`GET`) such resource (2). The `PO/{id}` resource represents a business process instance for a

particular client that will perform the composition and will also allow the user to inspect the composition state at any time (with GET) [Pautasso 2009a].

The first action of the composed service is to retrieve (GET) the Products resource (4), which is part of the Seller service. Notice that since the composed resource behaves as a client of the Products resource, it cannot simply move the response as-is to the user agent so that an end user can choose (through a Web browser) the products he or she wants to buy. If that were to happen, the composed resource would lose control of the interaction (e.g., the user may submit the products lists directly to the Products resource). Hence, the composed resource must implement various strategies to deliver the products list to the end user without losing control of the interaction (e.g., through javascript, modifying the resource representation, etc.). Once the composed resource obtains the final products list (5), it updates its internal state and invokes the next resource, Transaction, which is part of the Bank service, through a POST message (6). The resource response (7) indicates the state of the transaction (either succeeded or failed) and causes the composed resource to update its internal state. The composed service then finishes its execution responding with the purchase order details.

Since the component resources (Products or Transaction) cannot interact directly with the end-user through the user agent, they have no autonomous control over their participation. Furthermore, any component service may require a long time to complete its execution and provide a response (messages 5 and 7). For instance, a bank may require one or more business days to complete the transaction due to business logic (e.g., the amount is too high, the payment target resides in a foreign country, the end-user receives an email for authorizing the transaction that shall be answered within three business days, etc.) or to internal processes (e.g., a failure).

A centralized approach requires the server to keep track of the interaction state (to move the interaction forward later), between the composed resource and its components on the server side, which implies a stateful style that violates REST constraints and has a negative impact on server scalability. One solution is that the composed service includes a complex logic for deallocating server resources, that is, it may close active connections with component services and constantly polling later to find if the response is available, it may execute compensatory transactions that restore the services to their previous state in case the response is a failure. Or it may close the connection with the client and trust it to remember the business process instance address (PO/{id}) and to contact the server again later. This approach is generally implemented in WSDL/SOAP service composition (orchestrations) and JOpera for REST.

An alternative is to rely on a fully decentralized style based on asynchronous messages and callback connectors, as shown in Figure 5. Similarly to the centralized strategy, once a PO resource receives a POST message (1), it creates a subordinated resource (PO/{id}) which will behave as a callback connector. Through the callback, the composed resource will coordinate the invocation of components and will keep track of the composition state through a new subordinated resource (/PO/{id}/stateN). This approach has the advantage that the callback resource's states become visible for further inspection and monitoring (state0, state1, state2). The initial state (state0) is created through a POST message (3) as a side-effect of the redirection (303 See Other) indicated by the composed resource (2, 3, 4). Messages include the callback connector address in the Location header (/PO/{id}) indicating the step of the composition (2).

When the user agent fetches the state resource (5), it is redirected to invoke the component resource (/products), including an extra header indicating the address of the Callback connector (/PO/{id}/state1) (6). The connection with the composed resource is closed, and the component is requested (7). The component resource responds with the representation of the Products and the end-user selects the products he or she needs. From that step forward, the conversation occurs between the user-agent and

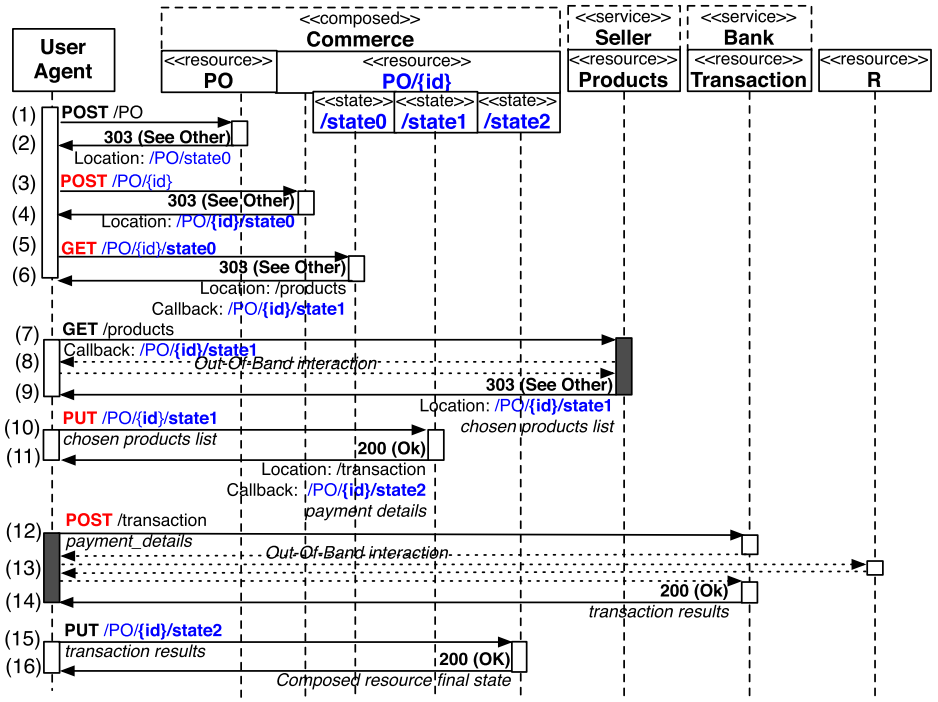

Fig. 5. Decentralized implementation of the composed `Commerce` REST service.

the component, that is, the conversation occurs out-of-band regarding the flow controlled by the composed resource (i.e., the composed resource does not intervene in such conversation and has no access to the information interchanged between the parties). The component service logic may require direct communication with the user possibly through alternative protocols (e.g., email) or transport channels (8).

Seconds, minutes, or days later, when the response is available, the component resource invokes the callback and delivers the response. For instance, if the interaction occurred through HTTP, the component resource could respond with a redirect message indicating the address of the callback component (9). The callback address refers to the expected state of the subordinated resource once the component finishes its job. In that case a `PUT` message is sent to the callback carrying on the component response (e.g., the chosen product list) (10).

This approach has a significant disadvantage. The component resource needs to know that it is part of a composition and needs to store the callback address in order to provide a final response, which increases the coupling between component and composed resources. An alternative strategy that reduces coupling is applied when interacting with the second component (11 to 16). In this case, we exploit user agent's capability to store state information. That is, instead of issuing a redirection, the composed service issues a `200` message, including the location of the component-service to invoke (e.g., a Bank transaction) and the callback address. It also includes the necessary parameters to proceed with the payment process (e.g., the amount to pay) (11), however in this case, the user agent will store the callback address and proceed with the payment transaction by performing a `POST` request (12).

Since the resource ignores that it is part of a composition, it eventually issues a `200` message with the final response (14). Notice that in this case, the user agent has no

way to differentiate the final answer among the various 200 messages it could receive during a rich interaction with the Transaction resource, so that if additional interaction with the end user is required, it shall occur as an out-of-band conversation (13) with additional resources (e.g., resource R). Once the final response is received (14) and using the locally stored callback, the user agent moves the received final response to the composed resource through a PUT message (15). Unlike message 10, there is no need for an end-user confirmation in this step, since the request is not issued as a consequence of a 303 redirection. Once the whole process is finished, the composed service (PO/{id}) marks its process as finished and responds with a representation of its final state (16).

3.2. State Handling for Composed RESTful Services

Key architectural properties of REST are high scalability and performance, and one way to achieve these properties is through the statelessness constraint, that is, requests from clients should contain all the information necessary for the server to process the request; hence, session state (also known as application state) is stored entirely at the client side instead of being shared also on the server side. This constraint makes it possible for the server to avoid keeping session state in memory after a response has been sent back to the client, which for the case of long-running business processes may imply days, months, or years.

Concerning RESTful service composition, there are various ways in which application state has been handled. The simplest way is for the user agent to request instructions from the composed resource in order to execute a business process specification entirely on the client side, considering also the locally stored application state (Figures 6(a1) and 6(a2)). Instructions can be provided through generic scripting languages, such as javascript, or even through specialized languages on service composition, such as BPEL. This approach guarantees statelessness and it is fully decentralized; however, there is no way to guarantee that the expected processing has taken place, and there is no way to enforce that the business process is correctly executed with untrusted user agents.

The symmetric alternative is to run the process completely on the server side and return the results only once the process has completed. The client remains blocked with an open connection to the server for the whole duration of the process, making this solution not suitable for long-running processes. An alternative is to expose the state of the running process instance (Figures 6(b1) and 6(b2)) through a specialized resource on the server side (e.g., /PO/{id}) that stores the state of the business process for each user agent that executes a business process. This way, it is also possible to inspect the state of the business process at any time (Figures 6(b3) and 6(b4)). This approach has been successfully used in JOpera [Pautasso 2009a]. It guarantees the enforcement of business processes and favors reusability of composed resources; however, the server must manage the state of each running process instance, compromising the scalability of the composed resource.

We can improve this approach by breaking down the composed resource into fragments representing the various steps or intermediary states (e.g., /PO/{id}/stateN) of a business process (Figure 6(c3)). This strategy could facilitate inspecting the business process (e.g., determining at which step some instances failed or stopped working) and to balance the load (e.g., some responses could be cached or moved to specialized platform, etc.) and improve performance. This approach, however, is still stateful, since a resource (e.g., PO/{id}) is created for each instance of the business process.

A fully stateless solution can be achieved by mixing the first and previous approaches (a), (c). That is, a business process is broken down into fragments representing intermediary states of a business process and control logic is centralized in

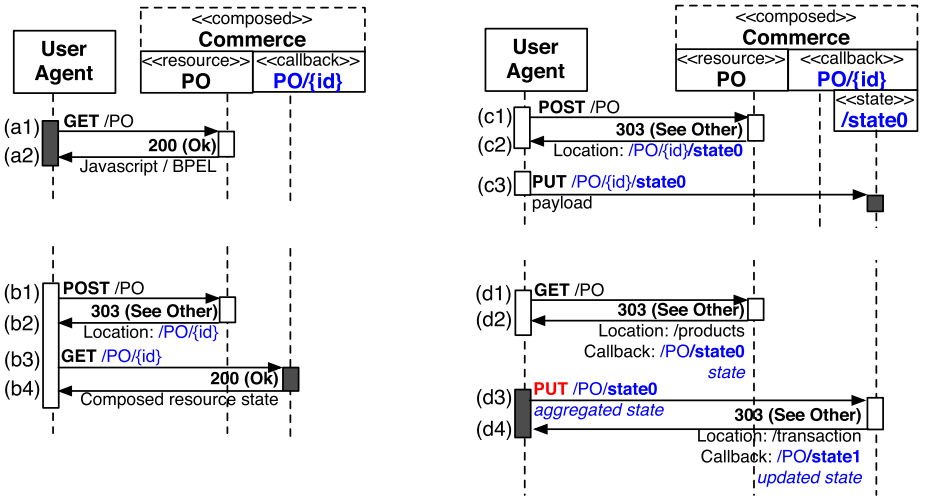

Fig. 6. Four strategies for handling application state on the client (a), (d) and on the server (b), (c) side.

the composed resource. Servers, however, do not store the state of the process instance (i.e., there is no need for a dedicated resource such as `PO/{id}`) but redirect user agents to obtain state information from third parties (Figure 6(d2)) in order to implement a step in the business process. The redirection message includes the address of the service component (Location header), the address of the next step of the business process (Callback header), information related to service component interface (i.e., parameters, HTTP method, payload format, URI scheme, etc.), and additional state (e.g., a hash number), or simple control patterns (e.g., a condition to be evaluated on the client-side).

Following redirection instructions, the user agent fetches the component service's state and aggregates such information to any other state information previously obtained from the composed service. The user agent then sends back the aggregated state to the callback as instructed (Figure 6(d3)). Servers, on the other hand, process the aggregated state according to the business rules corresponding to each specific state. Naturally, state information shall be encoded in a way that servers can process it, and content negotiation headers can be exploited to indicate encoding preferences. More redirection instruction could be issued by the servers if required (Figure 6(d4)).

Under this approach, responsibilities are shared by user agents and origin servers, state information is kept entirely on the client side so that servers remain stateless and highly scalable, whereas the logic of the business process is kept on the server, making it possible to enforce its correct execution. Naturally, we assume that user agents are well-behaved and will follow redirections as instructed. For the case of malicious user agents, mechanisms such as a digested signature of state information can be sent to the client side in order to verify whether user agents are behaving correctly, but not all properties can be protected with this technique. We discuss such limitations for each of the control-flow patterns presented in Section 4.

3.3. Rationale for Our Approach

The redirection code `303` was designed to inform the user agent it must fetch another resource, and it is widely used for services to interact with other services and accomplish business goals. For example, OAuth and OpenID are popular authorization and identity protocols implemented using redirection codes; payment entities which offer

online transactions are also implemented using redirections codes in order to allow e-commerce applications to sell products online in a security context under their control.

Due to constraints on the 303 redirection code, it cannot support complex interaction successfully. For instance, parameters should be serialized in a text format and concatenated to the URI (application/x-www-form-urlencoded), and information that cannot be serialized as plain text cannot be passed between applications in the URI parameters (e.g., images, pdf documents, etc). The resulting URI must not exceed the limit established by the server, otherwise the server should return a 414 Request-URI Too Long status code message. In order to send large quantities of data, the media type multipart/form-data and the POST method shall be used for submitting forms that contain files, non-ASCII data, and binary data. In addition, only the GET HTTP method can be used to automatically fetch the redirected URI, but as seen in our example, applications may be required to interact with each other using additional methods without requiring end-user confirmation (e.g., POST and PUT messages 3 and 10 in Figure 5).

More importantly, control flow may be more complex than sequential invocation of REST resources. Business processes also require parallel or alternative invocation as well as determining the conditions for choosing the right response; more complex control flows consider the invocation of two services in non-established order but only one at a time (unordered sequence), or service invocation for a determined number of times iterator.

Finally, notice that the composed REST service (PO and /stateN resources) encapsulates the knowledge about which services to invoke (URI), which parameters or state information should be sent and be expected to be received, which methods shall be used (e.g., GET (7) or POST (12) in Figure 5), as well as the order of the invocation, that is, they must know the service interface of the resource, which in our case is accomplished through ReLL.

4. CONTROL-FLOW PATTERNS

In the context of stateless, decentralized compositions of services described with ReLL and with the assumption that clients can process the Callback link header, in this section, we model control-flow patterns for RESTful service composition and the HTTP protocol. The set of patterns includes sequence, unordered sequence invocation, alternative, iteration, parallel, discriminator, and selection [Russell et al. 2006; van der Aalst et al. 2003].

4.1. Sequence, Unordered Sequence

The sequence pattern is a basic control-flow structure used to define consecutive invocations of services which are executed one after the other without any guard condition associated. As seen in Figure 5, this pattern could be implemented using the 303 redirection code; however, only automatic redirection of GET messages are allowed by the standard, making it difficult to update the composed resource state (i.e., PUT message of Lines 5, 9). In addition, there is no clear indication on how to handle the payload of the message. We extend the status codes with a set of codes identified with a two-digit prefix: 31x. The sequence pattern is implemented with a new code: 311 (Sequence) indicating the invocation of a service without any guard condition (see Figure 7).

The server responds with a 311 message including the component resource address (2, 6) in a Link header as well as the HTTP method in a link target attribute, and a Callback address in an additional header indicating the state of the composition. Additional information, such as state (e.g., a digested value) and, depending on the service interface, data formats or URI schemes to create the request, can be included in the payload. Actual data values for such templates shall be provided by the user agent

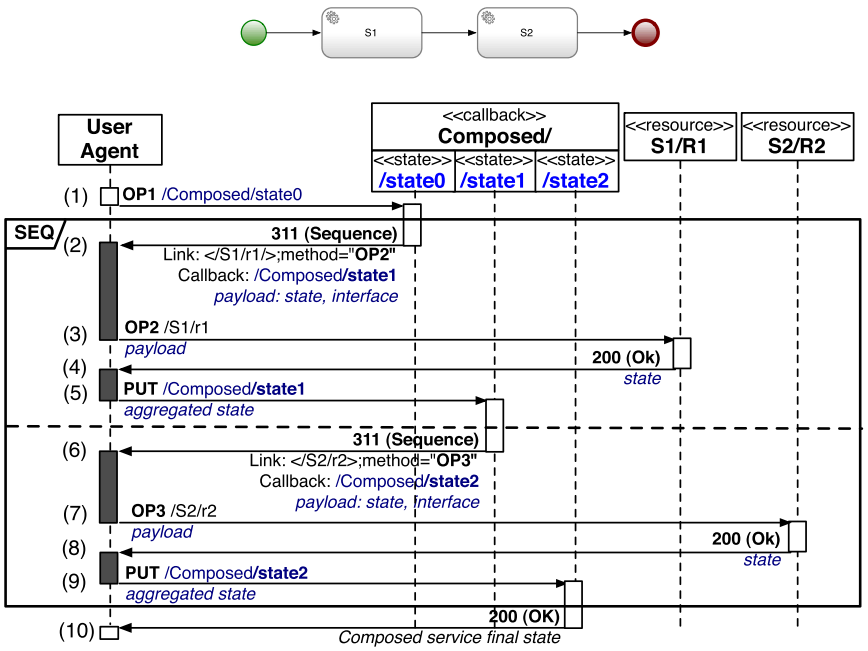

Fig. 7. A sequence control-flow pattern implemented for REST and HTTP.

either by requesting them to the user through an appropriate interface or retrieving them from the local storage. Such process is out of the scope of this proposal. The server may close the connection with the client after issuing a `311` message unless metadata indicating otherwise is included. When a user agent receives this code, it must store locally the callback address and automatically request the component resource using the indicated method (3), (7). Similarly to Figure 5, if additional communication shall occur between the component resource and the user agent, it must be modeled as out-of-band communication and is omitted for readability. When the response is available, the component replies with a `200` status code. The composed service shall not issue another request until the response has been passed by the user agent through a `PUT` message (5), then the composed service can proceed with the next component (6 to 9).

When all the components have been fetched (i.e., the final state of the sequence has been reached), the response is provided with a `200` status code and the composed service representation (10). Notice that the actual HTTP methods to be used when invoking component services must be determined by the composed resource. In order to know how to handle the resources, the composed service pre-fetches the component services descriptors which detail the interface of a set of resources at domain-level; the descriptors are themselves REST resources (Figure 8). This phase is omitted in the figures detailing the remaining patterns for readability, although it is assumed it takes place before invoking a component resource.

For the case of the unordered sequence pattern, it specifies the execution of two or more services in any order but not concurrently (Figure 9). That is, given two services `S1` and `S2`, the services execution can result as `S1` followed by `S2` or `S2` followed by `S1`. Since the list of services to be invoked is known by the composed resource and the order is irrelevant, the composed resource (server) has the information to decide which service to invoke as part of its own logic. For the user agent, all that matters is

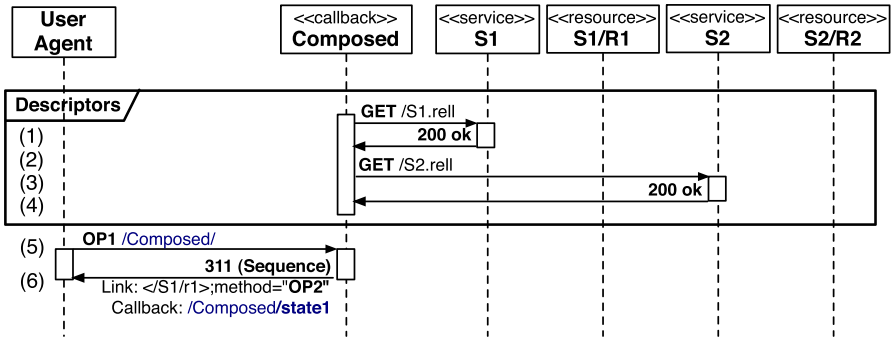

Fig. 8. ReLL descriptors are fetched considering the root resource of a service.

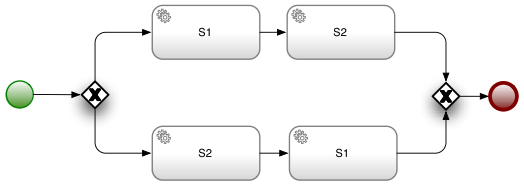

Fig. 9. Unordered sequence.

the address of a particular component resource to be invoked as well as the method; that is, this case is not different than a sequential invocation.

4.2. Alternative, Exclusive Choice

The alternative pattern is a basic control-flow structure that allows the execution of only one of two possible services. The service to be executed could depend on the outcome of preceding service execution, values of the parameters, or a user decision. The `312 (Alternative)` status code is proposed for this pattern. When a composed service requires executing one of two services, it responds to the client request with a `312` coded message, indicating the list of services to choose as `Link` headers, including the HTTP `method` as an attribute, and a `Callback` header indicating the connector state to resume interaction. The message payload is a conditional expression to be evaluated by the user agent, as well as information required to build proper request messages (i.e., data formats or URI schemes).

The composed resource closes the connection after issuing the response unless otherwise indicated by additional headers. Link services may differ on the resources (URIs) or the methods to be used (Figure 10, message 2). Since in REST, user agents keep application-state information [Zuzak et al. 2011], they shall have enough information to perform the evaluation. A good practice is to express the condition in languages well known to the Web, such as XPath, although its format escapes the scope of this proposal. Once the user agent has evaluated the condition it determines which link to follow (4) or (6). Again, additional communication may occur between a user agent and an origin server. Eventually when the component has a final response, it issues a `200` coded response, including its state in the payload (5) or (7). This causes the user agent to send an update message (`PUT`) to the composed resource carrying on the received payload (8). Once the interaction finishes, the composed resource replies with a `200` message including the representation of its final state (9).

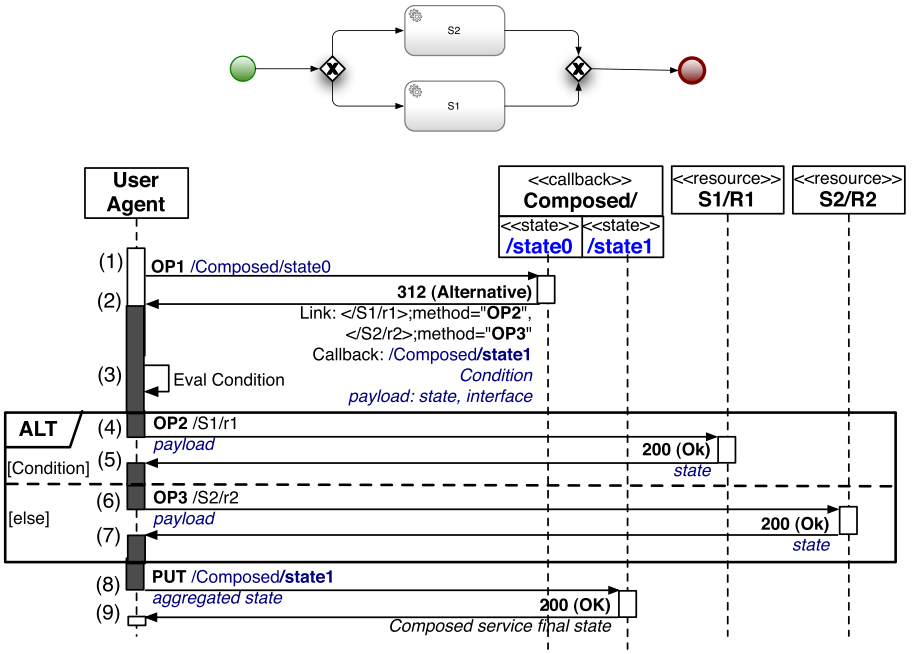

Fig. 10. An alternate control-flow pattern implemented for REST and HTTP.

4.3. Iteration, Structured Loop - while Variant

This pattern is an advanced control-flow structure that allows the execution of a service repeatedly, the number of times depending on a fixed number, a time frame, etc. which can be modeled by a conditional expression. We propose the `313 (Iteration)` status code for representing iterations.

This interaction begins when the composed resource issues a `313` message (Figure 11, message 2), including a `Link` header with the address of the component resource, a `Callback` header indicating the callback connector state address, a conditional expression, and additional information to create the message request as part of the payload. After evaluating the conditional expression (3) and obtaining positive results, the message is redirected to the component resource using the indicated operation and payload (4). Communication between client and server may include several messages interchanged. When a response is available, the component resource will issue a `200` message (5). The condition will then be evaluated again. If it still holds, the component is invoked again (4); or a `PUT` message is sent to the callback address carrying along the response content served by the component service aggregated with previous state information (6). Finally, at the end of the interaction, the component replies with a `200` message and the representation of the composed resource final state (7).

4.4. Parallel Split - Synchronization, Structured Discriminator, Structured Partial Join, Local Synchronization Merge (Selection)

The Parallel Split is a simple pattern that allows a single thread of execution to be split into two or more branches invoking services concurrently. The parallel split pattern can be paired with either one of four advanced control-flow structures. Under the paradigm of a composed service-component services, it is the former which determines whether it waits for all the responses (Synchronization, Figure 12(a)), just one of them

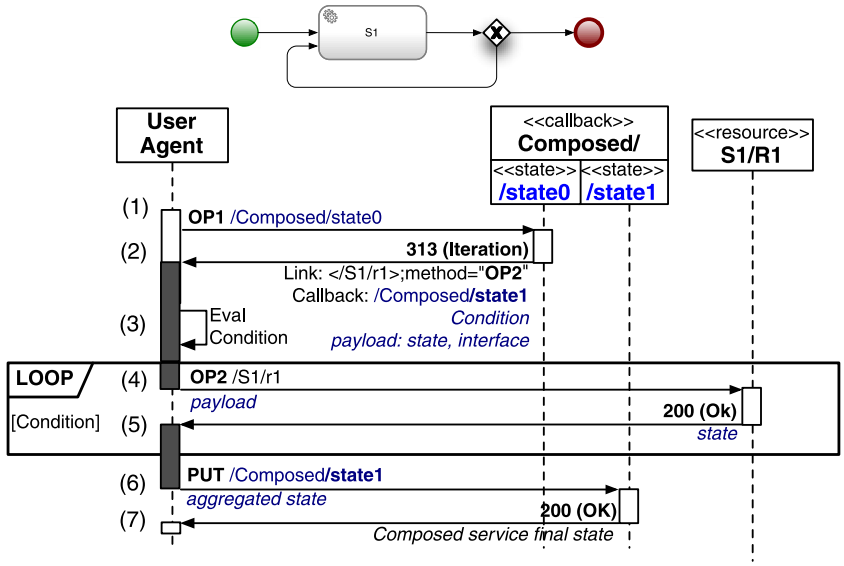

Fig. 11. An iterator control-flow pattern implemented for REST and HTTP.

(Structured Discriminator, Figure 12(b)), or a fixed number (Structured Partial Join, Figure 12(c)). Finally, for the case of Local Synchronization Merge (also called Selection, Figure 12(d)), the composed service shall wait for a number of responses that cannot be determined with local information.

In order to avoid violating the REST stateless principle, servers do not store information about how many answers are expected per client but make explicit server's expectancies through the pattern status codes, `314 Synch` (Synchronization), `315 Discriminate` (Structured discriminator), `316 PartialJoin` (Structured Partial Join), and `317 Selection` (Local Synchronization Merge). The four patterns follow the same conversational structure; however, the client's decision to inform the server about the availability of a final response is affected by the corresponding pattern.

Figure 12 shows the details for the pattern. Interaction starts when the composed resource issues either a `314`, `315`, `316`, or `317` message (Figure 12, message 2). The message includes a list of `Link` headers annotated with a `method` attribute, a `Callback` header indicating the callback connector state address, and a payload with instructions to format input data for the operations according service interface. It may also include state information, such as the number of expected service components to be addressed by the client for the case of the `316 Partial Join` pattern.

For the case of a `317 Selection` message, a conditional expression must be included. The condition must be evaluated considering application-state information stored locally at the client side (3), and the result shall be the number of request messages the client must issue to service components.

Once the user agent determines how many responses to provide to the composed resource (all, one, n out of m, or n), it invokes all the service components indicated in the list with the appropriate methods concurrently (4, 6). In practice, the number of links to be fetched is limited by the number of concurrent connections the client is able to maintain with the servers involved. Again, there may occur several messages interchanged between clients and origin servers as an out-of-band conversation, but once the final response is available, it is aggregated until the number of responses expected to be sent to the composed service is reached. The aggregated state is sent as

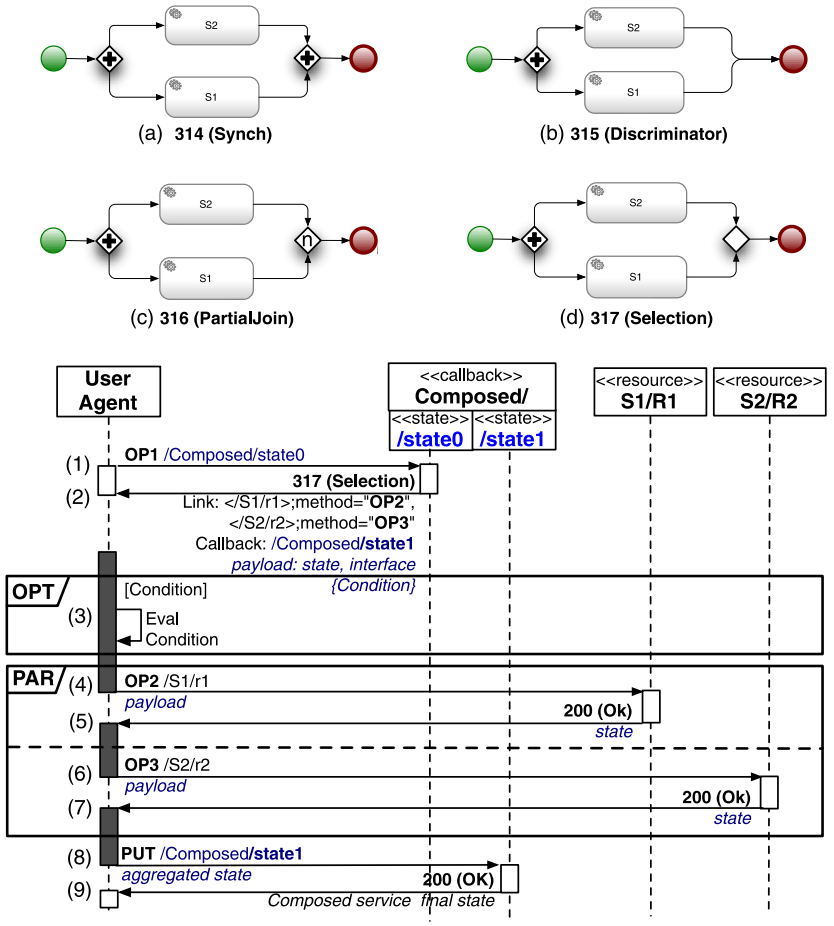

Fig. 12. A parallel control-flow pattern implemented for REST and HTTP.

a 200 coded request (8). The composed resource processes the aggregated state (e.g., it could merge the results) and issues a 200 coded response with the final state.

5. IMPLEMENTATION

The e-commerce scenario depicted in Figure 3 was implemented on a Node.JS server extended to make use of the HTTP status codes we have previously described. That is, the five RESTful Web services components were developed corresponding to the OAuth Provider, Seller, two Banks, an Email Sender, and a Delivery Service, as well as the composed service, which is the Commerce service.

The Commerce service execution comprises six steps. During the first two steps, the user chooses products to buy and gets authenticated. Both steps are executed one after the other, implementing the sequence control-flow pattern. Message interactions between user agent and composite service are shown in the following snippet log. The services were developed using the RESTify framework,[1] and the client was

[1]http://mcavage.github.com/node-restify/

implemented using the basic HTTP client library of Node.JS. We now present snippets of the messages interchanged between client and server at each step.

```
Step 1
Request:
  POST /commerce  HTTP/1.1
  Content-type: json

Response:
  311 Sequence
  Link: </seller/products>; method="GET"
  Callback: /commerce/state1

  <Out of band interaction between the end user and the Seller
     service>
-------------------------------------------------------
Step 2.
Request:
  PUT /commerce/state1  HTTP/1.1
  Content-type: json
  [Chosen products]

Response:
  311 Sequence
  Link: </OAuthProvider>; method="POST"
  Callback: /commerce/state2
  [API key]

  <Out of band interaction between the end user and the OAuth
     Provider, as a result, the client will obtain a request token>
```

To calculate the total due on the purchase order, it is necessary to execute the task of consulting for the price of each item repeatedly. The third step of the business process execution is implemented using the iterator control flow pattern. Interaction messages showing implementation of iteration control flow pattern are shown in the following fragment.

```
Step 3.
Request:
  PUT /commerce/state2  HTTP/1.1
  Content-type: json
  [Request token, chosen products]

Response:
  313 Iteration
  Link: </seller/prices>; method="GET"
  Callback: /commerce/state3
  [Conditional expression written in JavaScript]

  <User agent asks prices while the conditional expression evaluates
     positively (i.e., for each product)>

Request:
  GET /seller/prices  HTTP/1.1
  Content-type: json
  [Chosen products]
```

```
Response:
  200 OK
  [Price, seller details]
```

Once the prices of all products to buy are calculated, the Commerce Service offers users a set of supported payment options. The alternative control-flow pattern allows the user to choose only one of the payment options presented as shown in the following snippet.

```
Step 4.
Request:
  PUT /commerce/state3  HTTP/1.1
  Content-type: json
 [Final products list price, seller details]

Response:
  312 Alternative
  Link: </bank1>;method="GET",
           </bank2>;method="GET"
  Callback: /commerce/state4
  [Conditional expression written in JavaScript]
  [Payment details]

  <Local evaluation of the condition in order to select the bank
     (e.g. bank1) and out of band interaction between the end user
     and Bank services>
```

If payment for products to buy was successful, the next step is to send an email back to the buyer and generate a delivery order simultaneously. The parallel control flow pattern allows the execution of many tasks concurrently. Implementation details of this pattern are shown in the following snippet.

```
Step 5.
Request:
  PUT /commerce/state4  HTTP/1.1
  Content-type: json
  [Payment results, Chosen products]

Response:
  317 Selection
  Link: </emailSender>;method="POST",
           </deliveryService>;method="POST"
  Callback: /commerce/state5
  [Conditional expression written in JavaScript]
  [Purchase order, email details, delivery details]

  <Two request messages are send concurrently to the Email and
     Delivery services, as instructed>

Request:
  POST /emailSender  HTTP/1.1
  Content-type: json
  [Purchase order, email details]

Request:
  POST /deliveryService  HTTP/1.1
  Content-type: json
```

```
  [Purchase order, delivery details]

Response:
  200 OK
  [Mailing confirmation]

Response:
  200 OK
  [Delivery dispatch confirmation]

<Conditional expression is evaluated locally, on the client-side,
    until the result is positive (e.g., the confirmation of the
    Delivery dispatch is received)>
```

Finally, when completing the execution of the composite service process, the composed service returns the representation of its current state. For example, it could be an identification of the transaction for future reference (e.g., customer service). This content depends on the composed service logic.

```
Step 6.
Request:
  PUT /commerce/state5  HTTP/1.1
  Content-type: json
  [Purchase Order, mailing details and confirmation, delivery
      details and confirmation]
Response:
  200 OK
  [Tracking number]
```

6. EVALUATION

The quality of service attributes (QoS) of the overall composition depend not only on the component services QoS, but also on the control-flow pattern involved in the composition [Canfora et al. 2005]. Each control-flow pattern can be modeled with a finite state machine representing the execution paths (Figure 13).

For the case of the sequence pattern, a single path of execution is followed (i.e., the sequential invocation of component services). Similarly, the path of execution for an iteration control-flow pattern consists on the sequential invocation of a service as many times as required. The alternative control pattern requires to evaluate its condition (which takes a relatively insignificant time when compared to the time it takes to invoke a service). Finally, the parallel control-flow pattern causes multiple concurrent service invocations [Alrifai and Risse 2009; Zeng et al. 2004].

In addition, QoS has been determined mainly by evaluating variables such as *price*, *response time* or *duration*, *reputation*, *performance* or *success rate*, and *availability*, among others. Composed services QoS is measured using aggregation functions on the QoS of the component services [Alrifai and Risse 2009]. In Zeng et al. [2004] aggregation functions for each control-flow pattern are defined based on the critical execution path of the composed service, that is, for the worst possible cases. Control-flow patterns such as iteration, alternative, and parallelism can be reduced or transformed to a sequence model [Alrifai and Risse 2009; Cardosoet al. 2004].

In the rest of this section, we will provide an evaluation of the proposed approach (decentralized vs. centralized service composition) considering only the sequential control-flow pattern (i.e., the worst case). We implemented nine composed services invoking two to ten service components. Each service component takes 1,000 ms (1 second) execution time and supports up to 600 clients concurrently connected.

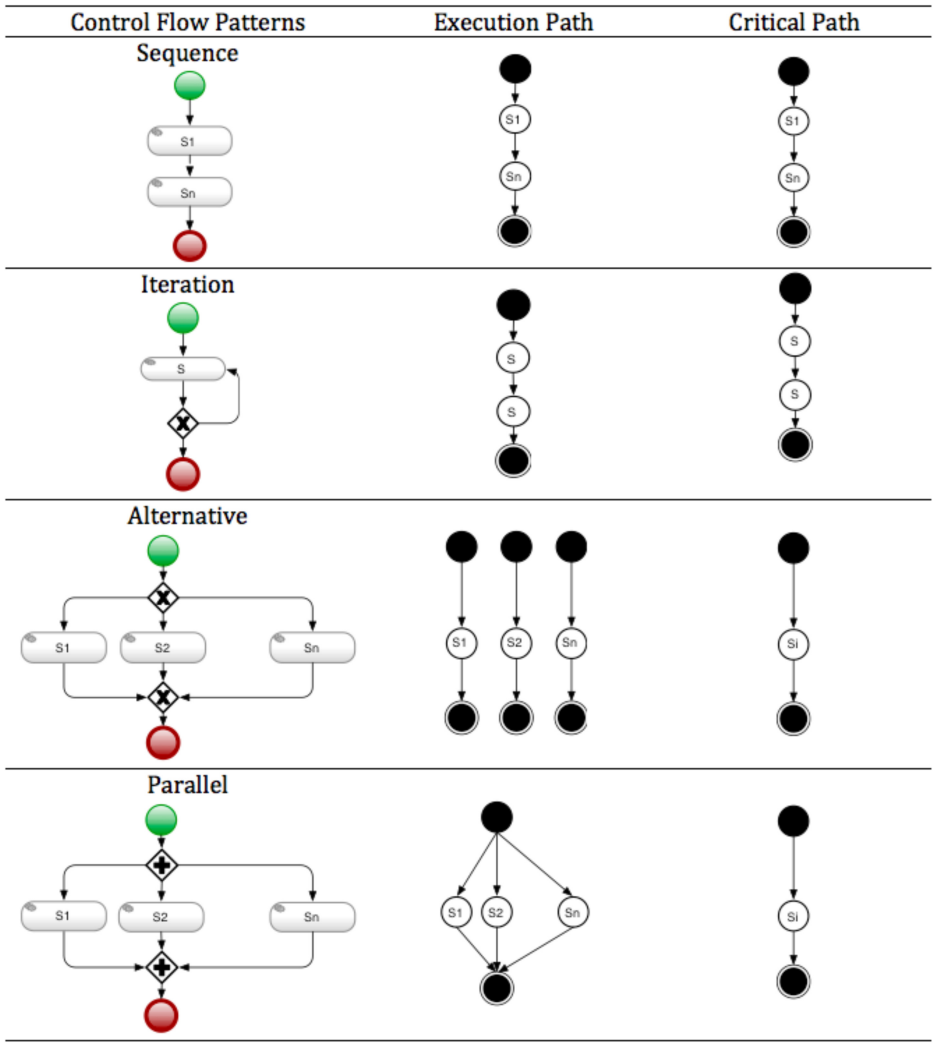

Fig. 13. Execution path and critical path for each control-flow pattern.

The service component has a processing capacity (throughput) of 600req/sec, an average response time of 1,000ms, and 100% availability when the demand or workload (workload) is less than or equal to the processing capacity of the service component (600req/sec).

Both decentralized and centralized scenarios are compared in terms of availability, response time, and throughput. For the purpose of analyzing the scalability of composite services, we assigned the resources so that each composite service can serve up to 100 clients at a time and can keep requests waiting in a queue of 100 clients (maximum size), that is, when the server is at is maximum capacity (limit), it can receive up to 200 clients. The clients arriving when the server is at it limit will cause a service denied response (e.g., 429 - too many requests). As described before, we implemented services with Node.js and performed the workload test using JMeter [Halili 2008].

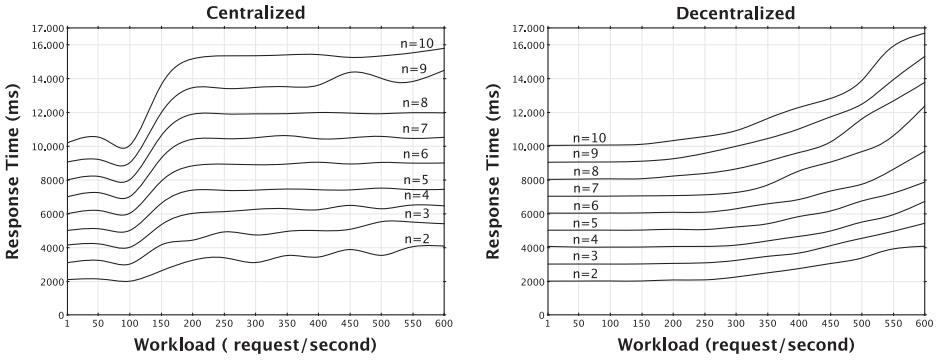

Fig. 14. Response time.

6.1. Response Time

The response time is the time it takes to send a request from the client and receiving a response from the service. Figure 14 shows a comparison of the composed service response time under a centralized versus a decentralized approach. In both cases, if the composed service workload (w) is equal to or less than the service processing capacity (in our example $w \leq 100 req/sec$), then the response time corresponds to the sum of each component service response time. The figure shows the results where a composition includes $n = 2...10$ component services. As can be observed, when the workload exceeds the composed service capacity, the composed service response time increases exponentially up to the limit of the service (in our example $L = 200 req/sec$) and then the service fail requests (e.g. responds with a `404 Not found` message). However, due to the stateless nature of the decentralized approach (i.e., no sessions are kept alive on the server side), the increase in the response time after reaching the service limit is much lower for the decentralized approach compared to the centralized scenario. The difference may seem marginal, however, the availability analysis demonstrates that the centralized approach presents a significantly higher rate of failed requests so that the decentralized approach maintains a reasonable response time while processing (not failing) a significantly higher number of requests.

6.2. Availability

Service availability is a proportion between the number of successful service invocation versus the total service invocation received by a server. A service invocation is successful if a response annotated with a 2xx or 3xx status code is produced on the service side and received by a client. Figure 15 analyzes the composed service availability for both approaches in various scenarios, where the composed service includes $n = 2...10$ component services. Under the centralized approach, availability decreases regardless of the number of component services (again the service limit $L = 200\ req/sec$). Under the decentralized approach, however, the number of failed requests is significantly less in comparison considering the same service limit and number of components.

6.3. Throughput

Service throughput is the amount of workload (request in this case) that the service can process in a unit of time. Figure 16 shows the throughput for both approaches with a configuration similar to the previous cases ($n = 2...10$ service components). In both approaches, the composed service throughput decreases if the number of service

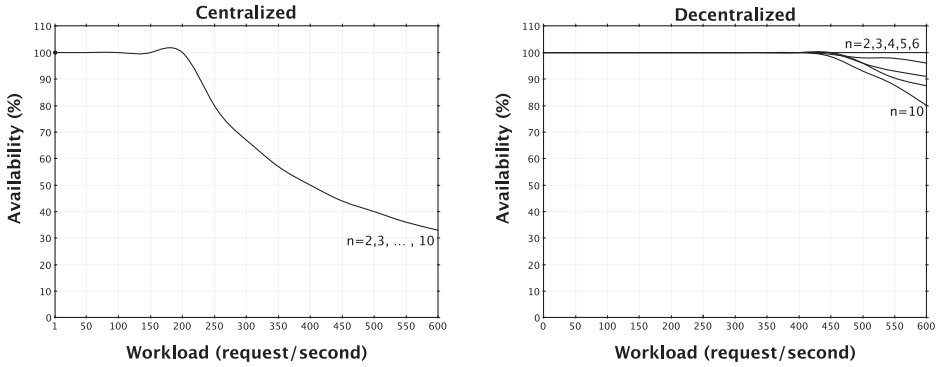

Fig. 15. Availability.

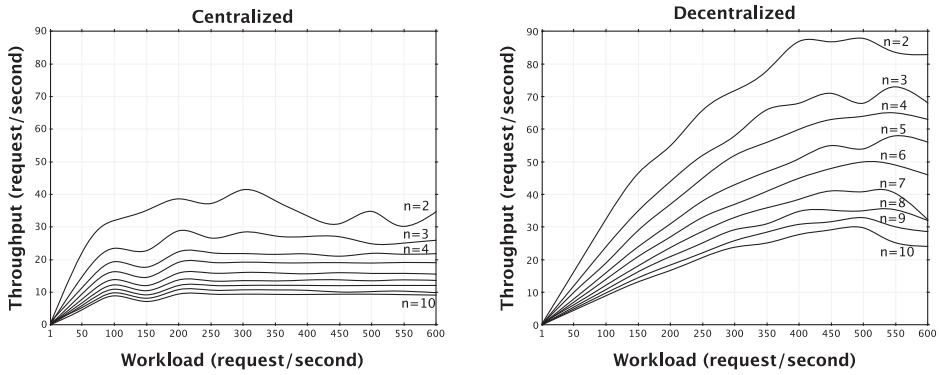

Fig. 16. Throughput.

components increases. However, a significant difference can be seen in the decentralized approach that is able to process more requests per second, whereas the centralized approach reaches a limit (failed requests) when the workload exceeds the service limit. The decentralized throughput rate overcomes the centralized approach for each scenario.

7. DISCUSSION

7.1. Impact on REST Architectural Properties

The REST architecture style yields to applications (e.g., the Web), nonfunctional properties like openness, extensibility, high scalability [Taylor et al. 2009], performance, simplicity, modifiability, visibility, portability, and reliability [Fielding 2000].

7.1.1. Openness, Modifiability, and Extensibility. An open software architecture is characterized as having a stable abstract representation (system model) as a core while allowing third-party developers to evolve independently the application. This may cause changes on the system model and the application itself; these must be validated against the semantics of the system model [Oreizy 2000]. Architectural components have different mechanisms of extensibility and modifiability, and by satisfying the layered system constraint as well as the uniform interface, they are required to bound responsibilities, changes and complexity to the corresponding layers and components.

For instance, methods and control data on HTTP messages must allow architectural components (e.g., intermediaries such as proxies) to handle messages properly without requiring additional semantics, information, or parsing the message content. Our approach relies on HTTP extensibility mechanisms (status codes) and the definition of a new media-type, the ReLL descriptor, without requiring special handling of messages from intermediaries or old clients, and allowing both mechanisms to be further extended.

In addition, user agents can receive code on demand, extending clients' functionality dynamically (e.g., a script or an applet). In a centralized implementation of service composition (Figure 4), this extensibility is lost as resources are fetched by an intermediary that has no use for scripts, css, or applets. In a decentralized implementation (Figure 5), out-of-band interaction between the end user and the component service may occur, taking advantage of extensible components (scripts, applets, css).

7.1.2. High Scalability and Performance. Statelessness, understood in REST as the lack of records of client-server interaction or sessions on the server side (application state), is an important property that allows the server to deallocate resources (memory, connections) used in responding to a client's request. It fosters the server's high scalability of concurrent requests, provided that the messages contain all the necessary information for servers to respond properly.

In a centralized implementation, the composed service behaves as a client of component resources and hence keeps the record of interactions on its side; however, it behaves as a server for end users consuming the composed resource and hence violates the statelessness principle of REST. In a decentralized implementation, the composed service does not store state information but pushes it forward it to the end-user client, keeping the composed resource stateless.

A common practice to ensure higher levels of scalability for servers is to balance the load of requests among a cluster of servers. For a centralized implementation, the composed service is responsible for performing the load balancing task to the local cluster. For the case of a decentralized implementation, the request load is naturally distributed among the various response providers (components).

In addition, for the case of REST, intermediaries may also perform partial processing of requests only if requests are self-contained or context-free. Hence, despite messages that can be longer in size, the information can be replicated or cached across a set of intermediaries (proxies and gateways), increasing the system performance and robustness. Caching is one of the key features of REST that allows applications to be highly scalable.

7.2. Backwards Compatibility

In this article, we have proposed an extension of HTTP status codes to support basic and complex control-flow patterns that are widely known in the traditional Web services composition field. Our decision does not introduce additional components to the Web but exploits existent extension mechanisms of HTTP, making it possible for existent components (clients, servers, and intermediaries) to graciously fail when receiving one of the proposed messages (i.e., treat them as a standard `300` status code) and for compliant components to enable long-running business processes on the Web.

Current definition of the `303` status code does not allow for the specification of complex control flow. Furthermore, since the semantics of such status code differs from the intention of the patterns presented in Section 4, we believe that the definition of new codes for each pattern allows advanced architectural components more control in the way they handle the messages. Thus, basic control-flow patterns can be implemented using new HTTP redirection codes that give enough information to advanced

user agents to interpret and perform the patterns. This approach allows the implementation of complex workflows. Currently, protocols such as OAuth[2] and OpenID[3] are implemented using the 303 redirection code, which forces developers to define cumbersome APIs hard to code and understand, where information is mashed in the parameters in an ad-hoc fashion (e.g., callback URIs, chains of callback URIs, security keys, signatures, etc.), causing leakage of information and introducing security risks.

In our approach, servers know explicitly that they are part of a flow that potentially involves other origin servers outside their domain so they must be careful with their responses. Since flows may cross various domains, our proposal violates the cross-origin policy in those cases so that the rules for processing each message shall be implemented in specialized browsers (otherwise, it will be treated as a simple redirection).

In addition, malicious user agents or clients may ignore information, such as conditional expressions or component addresses, and attempt to move tampered states to the composed service. To avoid such situations, component services descriptions (i.e., ReLL) shall include ways to validate the response provided by the components (e.g., an XML schema, message signature, digested values, etc.), but ultimately, it is the responsibility of the service components and the composed service to implement the required business rules and validation mechanisms to accept or not accept a response.

8. CONCLUSION

In this article, we have implemented control flows through callbacks and redirections. Our approach allows composed resources to delegate control flow to various services so that they become available to respond to newer messages. When the response to the delegated message is available, services will wake up the composed resource at the corresponding state in the execution flow.[4] In the evaluation section, we can observe that our decentralized approach allows composed services significant improvements in availability and throughput, whereas response time remains stable, demonstrating that decentralized service composition favors scalability. In addition, by exploiting REST nature, it is possible to implement long-running business processes that may take days, months, or years for completion without consuming resources on the server side (i.e., stateful), which favors scalability and evolvability.

Current development on service composition follows a centralized, stateful approach. RESTful service composition is a recent area of research that basically follows a similar tendency, violating REST principles. The consequences are a loss on nonfunctional properties, such as scalability, among others, which has been fundamental for the richness that the Web platform currently offers. A fully RESTful approach has proven elusive. Naturally, there are various options and trade-offs for designing such a kind of composed services. Our approach is fully RESTful compliant and focuses on extending the uniform interface, hence evolving HTTP in a way that requires minimal changes to current standards. Our approach balances composition responsibilities between clients and servers fostering the massive scalability that is akin to the Web and also allows the presented control-flow patterns to be regulated by current standards bodies so that thick client implementation could be normed and certified.

This article does not discuss how the composed service knows which component to invoke or which control-flow pattern shall be used to perform the invocation. Naturally, this information can be hardcoded in the composed service logic, and

[2]http://OAuth.net/

[3]http://OpenID.net/

[4]http://composedservice/state

probably some assistance could be provided for developers to create the service. Our approach also relies on a middle ground, where a service descriptor provides enough information for a client to make assumptions, discover new services, and choose which path to follow. There are several ways to describe REST services (e.g., WSDL 2.0, WADL, ReLL), but ReLL is the only descriptor that allows the extraction of data in resource representation dynamically; such data can be used to evaluate conditions and routing some control-flow patterns so that it is not required that the user agent know before hand the service descriptors, as depicted in Figure 8; they could be fetched on a need basis.

In order to perform workflows on the Web, it is necessary to allow the execution of services in different ways, for example, parallel, alternative, iteration, etc., and not only in sequence. There are other control-flow patterns commonly used in business process modeling, which we believe could be successfully addressed by implementing the patterns presented in this article. Further research is needed to model more dynamic aspects of control-flow patterns, which include features like dynamic routing, events, and a dynamic discovering of REST services.

In Alrifai et al. [2012] complex workflow patterns for WSDL/SOAP-based services (sequential, iteration, parallel, and conditional) are analyzed in order to implement centralized, QoS-aware service composition. It is clear that each workflow pattern has different consequences on the QoS attributes analyzed (e.g., performance, availability, and scalability), being the centralized composite service a bottleneck for the provision of such QoS attributes. As for future work, we are focusing on developing analytical models to perform a similar analysis in order to determine the impact of our fully decentralized approach on such QoS attributes for each control-flow pattern.

REFERENCES

Alarcón, R. and Wilde, E. 2010. RESTler: Crawling RESTful services. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, 1051–1052.

Alarcón, R., Wilde, E., and Bellido, J. 2010. Hypermedia-driven RESTful service composition. In *Proceedings of the 6th Workshop on Engineering Service-Oriented Applications (WESOA'10)*. Lecture Notes in Computer Science, vol. 6568, Springer, Berlin, Heidelberg, 111–120.

Alrifai, M. and Risse, T. 2009. Combining global optimization with local selection for efficient QoS-aware service composition. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*. ACM, New York, NY, 881–890.

Alrifai, M., Risse, T., and Nejdl, W. 2012. A hybrid approach for efficient Web service composition with end-to-end QoS constraints. *Trans. Web 6*, 2, 7:1–7:31.

Beckett, D. and McBride, B. 2004. RDF/XML syntax specification (revised). http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

Bellido, J., Alarcon, R., and Sepulveda, C. 2011. Web linking-based protocols for guiding RESTful M2M interaction. In *Proceedings of the 3rd International Workshop on Lightweight Composition on the Web (ComposableWeb'11)*. Lecture Notes in Computer Science, vol. 7059, Springer, Berlin, Heidelberg, 74–85.

Benatallah, B., Sheng, Q. Z., and Dumas, M. 2003. The self-serv environment for Web services composition. *IEEE Int. Comput. 7*, 40–48.

Canfora, G., Di Penta, M., Esposito, R., and Villani, M. L. 2005. QoS-aware replanning of composite Web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*. IEEE Computer Society, 121–129.

Cardoso, J., Sheth, A., Miller, J., Arnold, J., and Kochut, K. 2004. Quality of service for workflows and Web service processes. *Web Semantics: Sci. Serv. Agents WWW 1*, 3, 281–308.

Chinnici, R., Moreau, J., Ryman, A., and Weerawarana, S. 2009. Web services description language (WSDL) version 2.0, part 1: Core language, W3C recommendation, 16 June 2007.

Dayal, U., Hsu, M., and Ladin, R. 2001. Business process coordination: State of the art, trends, and open issues. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass Eds., Morgan Kaufmann, San Francisco, CA, 3–13.

Decker, G., Lüders, A., Overdick, H., Schlichting, K., and Weske, M. 2009. RESTful petri net execution. In *Proceedings of the International Workshop on Web Services and Formal Methods*, R. Bruni and K. Wolf Eds., Lecture Notes in Computer Science, vol. 5387, Springer-Verlag, Berlin, Heidelberg, 73–87.

Fielding, R. T. 2000. Architectural styles and the design of network-based software architectures. Ph.D. dissertation, University of California, Irvine.

Fielding, R. T., Gettys, J., Mogul, J. C., Frystyk Nielsen, H., Masinter, L., Leach, P. J., and Berners-Lee, T. 1999. Hypertext transfer protocol — HTTP/1.1. Internet RFC 2616. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Hadley, M. 2009. Web application description language (WADL). World Wide Web Consortium, Member Submission SUBM-wadl-20090831.

Halili, E. H. 2008. *Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites*. Packt Publishing Ltd, Birmingham, U.K.

Hamadi, R. and Benatallah, B. 2003. A petri net-based model for Web service composition. In *Proceedings of the 14th Australasian Database Conference (ADC'03)*, K.-D. Schewe and X. Zhou Eds., Conferences in Research and Practice in Information Technology, vol. 17. ACS, Adelaide, Australia, 191–200.

He, J., Zhang, Y., Huang, G., and Cao, J. 2012. A smart Web service based on the context of things. *ACM Trans. Internet Technol. (TOIT) 11*, 3, 13:1–13:23.

Hernández, A. G. and García, M. N. M. 2010. A formal definition of restful semantic Web services. In *Proceedings of the 1st International Workshop on RESTful Design (WS-REST'10)*, C. Pautasso, E. Wilde, and A. Marinos Eds., ACM, New York, NY, 39–45.

Mitra, N. and Lafon, Y. 2010. SOAP version 1.2 part 0: Primer. W3C Recommendation. http://www.W3C.org/TR/soap1/part0/. 27 April 2007.

Narayanan, S., Marucheck, A. S., and Handfield, R. B. 2009. Electronic data interchange: Research review and future directions. *Decision Sci. 40*, 1, 121–163.

Nierstrasz, O. and Meijler, T. D. 1995. Requirements for a composition language. In *Object-Based Models and Languages for Concurrent Systems*. Lecture Notes in Computer Science, vol. 924, Springer, Berlin Heidelberg, 147–161.

Nottingham, M. 2010. Web linking. Internet RFC 5988. http://www.ietf.org/rfc/rfc5988.txt.

Oreizy, P. 2000. Open architecture software: A flexible approach to decentralized software evolution. Ph.D. dissertation, University of California, Irvine.

Pautasso, C. 2009a. Composing RESTful services with JOpera. In *Proceedings of the 8th International Symposium on Software Composition*, A. Bergel and J. Fabry Eds., Lecture Notes in Computer Science, vol. 5634. Springer-Verlag, Berlin, Heidelberg, 142–159.

Pautasso, C. 2009b. On composing RESTful services. In *Software Service Engineering*, F. Leymann, T. Shan, W.-J. van den Heuvel, and O. Zimmermann Eds., Number 09021 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Pautasso, C. 2009c. RESTful Web service composition with BPEL for REST. *Data Knowl. Eng. 68*, 9, 851–866.

Pautasso, C. and Wilde, E. 2009. Why is the Web loosely coupled?: A multi-faceted metric for service design. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*. ACM, New York, NY, 911–920.

Peltz, C. 2003. Web services orchestration and choreography. *Computer 36*, 10, 46–52.

Richardson, L. and Ruby, S. 2007. *RESTful Web Services*. O'Reilly & Associates, Sebastopol, CA.

Rosenberg, F., Curbera, F., Duftler, M. J., and Khalaf, R. 2008. Composing RESTful services and collaborative workflows: A lightweight approach. *IEEE Internet Comput. 12*, 5, 24–31.

Russell, N., ter Hofstede, A. H. M., van der Aalst, W. M. P., and Mulyar, N. 2006. Workflow control-flow patterns: A revised view. Tech. rep. BPM-06-22. BPMcenter.org.

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. 2009. *Software Architecture: Foundations, Theory, and Practice* 1st Ed. John Wiley & Sons, Hoboken, NJ.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. 2003. Workflow patterns. *Distrib. Parallel Datab. 14*, 1, 5–51.

Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Vallés, J. G., and Van de Walle, R. 2012. Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. In *Proceedings of the 3rd International Workshop on RESTful Design (WS-REST'12)*. ACM, New York, NY, 33–40.

Zeng, L., Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnanam, J., and Chang, H. 2004. QoS-aware middleware for Web services composition. *IEEE Trans. Softw. Eng. 30*, 5, 311–327.

Zhao, X., Liu, E., and Clapworthy, G. 2011. A two-stage restful Web service composition method based on linear logic. In *Proceedings of the 9th IEEE European Conference on Web Services (ECOWS)*. IEEE Computer Society, 39–46.

Zuzak, I., Budiselic, I., and Delac, G. 2011. A finite-state machine approach for modeling and analyzing RESTful systems. *Web Eng. 10*, 4, 353–390.

Received September 2012; revised March 2013, August 2013; accepted October 2013