DYNAMIC COMPOSITION OF REST SERVICES

JESÚS EDWIN BELLIDO ANGULO

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Doctor in Engineering Sciences

Advisor:
ROSA ALARCÓN

Santiago de Chile, December 2014

© MMXIV, JESÚS EDWIN BELLIDO ANGULO
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
SCHOOL OF ENGINEERING

DYNAMIC COMPOSITION OF REST SERVICES

JESÚS EDWIN BELLIDO ANGULO

Members of the Committee:
ROSA ALARCÓN
YADRAN ETEROVIC
CECILIA BASTARRICA
ANDRÉS NEYEM
CESARE PAUTASSO
CRISTIÁN VIAL

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Doctor in Engineering Sciences

Santiago de Chile, December 2014

© MMXIV, JESÚS EDWIN BELLIDO ANGULO
To my family and friends
ACKNOWLEDGEMENTS

This thesis would not have been finished without the support and encouragement of several special people. Therefore, I would like to take this opportunity to show my gratitude to those who have assisted me in countless ways.

I would first like to express my greatest thanks to my supervisor Rosa Alarcon. There was many times when I was lost and each time Rosa was there to steer me towards the right path. Her predisposition to offer me so much of her time, trust and intellect is the major reason this thesis is completed.

I also would like to thank my friends for their appreciated help and cooperation. I have had the good fortune to study with some wonderful and authentic people: Lisney, for all his teachings and nice moments shared. Paul, who gave me the gift of the freshness of his imagination. Jorge, who gave me moments of interesting talks. Soledad, who gave me valuable advices. Debbie, who offered me her support. Thanks to Teresa, Juan and Mauricio. I apologize if I forget some name.

I specially would like to thank my family to whom I owe a great deal. To my father Justo, thank you for showing me the way of endeavour and nobleness. My own curiosity probably comes from his recursive inquiries about the nature. To my brothers Luz, Moisés and María Soledad, they are my guide and are always in my heart. To Gregoria, Valery and Doris, thank you for teaching by example the importance of values and the true meaning of wealth. To my mother Cleofé, thank you for showing me the way of fortitude. She is a great example of courage and love. Finally, to all those who with their help made it possible for me to continue on the way: Nora, Guillermima, Antonio and Fernando.

Finally, I would like to thank the institution that funded my studies and travels: DCC, DIPEI and VRAID.
### TABLE OF CONTENTS

ACKNOWLEDGEMENTS .................................................. iv
LIST OF FIGURES ..................................................... ix
LIST OF TABLES ....................................................... xi
ABSTRACT ............................................................... xii
RESUMEN ............................................................... xiv

1. INTRODUCTION ...................................................... 1
   1.1. Background ..................................................... 2
       1.1.1. Service Oriented Architecture ......................... 2
       1.1.2. Web services .............................................. 4
       1.1.3. Service Composition .................................... 7
       1.1.4. REST services composition .............................. 10
       1.1.5. QoS ......................................................... 12
   1.2. Limitations .................................................... 13
   1.3. General Goals ................................................ 14
   1.4. Hypothesis .................................................... 15
   1.5. Thesis Work and Main Contributions ........................ 15
   1.6. Document Structure .......................................... 17

2. Control Flow Patterns for Decentralized RESTful Service Composition 19
   2.1. Abstract ...................................................... 19
   2.2. Introduction .................................................. 19
   2.3. Background .................................................... 22
       2.3.1. Service composition in REST ............................ 23
       2.3.2. Links Processing .......................................... 25
       2.3.3. HTTP based interaction ................................. 26
### 3.5.1. Security Ontology ........................................ 74
### 3.5.2. ReLL: Resource Linking Language .......................... 76
### 3.5.3. Security Constraints Model: ReLL-S ......................... 78
### 3.6. ReLL-S security description .................................... 80
  - 3.6.1. API Keys .................................................. 80
  - 3.6.2. Username and Password ................................... 81
  - 3.6.3. HTTP Basic Authentication ................................ 81
  - 3.6.4. HTTP Digest Authentication ............................... 83
  - 3.6.5. OpenID ................................................... 86
  - 3.6.6. OAuth ..................................................... 88
### 3.7. Implementation and Case of Study: RESTler and Flickr ............. 91
  - 3.7.1. ReLL-S extensions for RESTler ............................ 92
  - 3.7.2. OAuth for Flickr ......................................... 95
  - 3.7.3. Analysis of our approach in light of the REST principles .. 101
### 3.8. Conclusions .................................................... 102

### 4. SAW-Q: A dynamic composition approach of REST services based on queue model ................................................. 105
  - 4.1. Abstract ...................................................... 105
  - 4.2. Introduction .................................................. 105
    - 4.2.1. Dynamic Composition ..................................... 106
    - 4.2.2. QoS-aware composition .................................. 107
    - 4.2.3. Contribution ............................................... 108
  - 4.3. Related Work ................................................ 109
  - 4.4. Towards a Queue Model for RESTful services .................... 110
    - 4.4.1. REST architectural constraints ........................ 110
    - 4.4.2. Understanding REST and Queue Models .................. 113
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Links processing levels</td>
<td>25</td>
</tr>
<tr>
<td>2.2</td>
<td>ReLL description for the resources catalog of products and shoppingCart.</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>The Commerce REST service composes the REST services Seller, two Banks,</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>a Delivery, Delivery, Email and Authentication.</td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Centralized implementation of the composed Commerce REST service.</td>
<td>33</td>
</tr>
<tr>
<td>2.5</td>
<td>Decentralized implementation of the composed Commerce REST service.</td>
<td>34</td>
</tr>
<tr>
<td>2.6</td>
<td>Four strategies for handling application state on the client (a, d) and on the server (b, c) side.</td>
<td>38</td>
</tr>
<tr>
<td>2.7</td>
<td>A sequence control flow pattern implemented for REST and HTTP</td>
<td>41</td>
</tr>
<tr>
<td>2.8</td>
<td>ReLL descriptors are fetched considering the root resource of a service.</td>
<td>42</td>
</tr>
<tr>
<td>2.9</td>
<td>Unordered Sequence</td>
<td>42</td>
</tr>
<tr>
<td>2.10</td>
<td>An alternate control flow pattern implemented for REST and HTTP</td>
<td>43</td>
</tr>
<tr>
<td>2.11</td>
<td>An iteror control flow pattern implemented for REST and HTTP</td>
<td>44</td>
</tr>
<tr>
<td>2.12</td>
<td>A parallel control flow pattern implemented for REST and HTTP</td>
<td>46</td>
</tr>
<tr>
<td>2.13</td>
<td>Execution path and critical path for each control-flow pattern</td>
<td>53</td>
</tr>
<tr>
<td>2.14</td>
<td>Response Time</td>
<td>55</td>
</tr>
<tr>
<td>2.15</td>
<td>Availability</td>
<td>56</td>
</tr>
<tr>
<td>2.16</td>
<td>Throughput</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Security ontology: Oval shapes represent concepts while rectangles represent instances (blue arrows). Green arrows represent specialization relationships and black arrows correspond to relationships with specific domain semantics.</td>
<td>74</td>
</tr>
<tr>
<td>3.2</td>
<td>ReLL-S a security extension for security aware service composition</td>
<td>77</td>
</tr>
<tr>
<td>3.3</td>
<td>Abstract OAuth interaction between four parties</td>
<td>89</td>
</tr>
<tr>
<td>3.4</td>
<td>RESTler architectural components</td>
<td>93</td>
</tr>
</tbody>
</table>
LIST OF TABLES

1.1 Service composition strategies ............................................. 8

4.1 Architecture constraints that determine the scalability of RESTful services. . 111

4.2 Quality attributes aggregation functions depending on control-flow patterns 124

4.3 Characteristics of the candidate services to implement the Travel Planner abstract process. .................................................. 131
Service composition is one of the principles of service-oriented architecture; it enables reuse and allows developers to combine existing services in order to create new services that in turn can be part of another composition. Dynamic composition requires that service components are chosen from a set of services with equal or similar functionality at runtime, and possibly automatically. The adoption of the REST services in the industry has led to a growing number of services of this type, many with similar functionality. The existing dynamic composition techniques are method-oriented whereas REST is resource-oriented, and considers only traditional (WSDL/SOAP) services. The REST architectural style has attracted a lot of interest from the industry due to the non-functional properties it contributes to Web-based solutions. In this thesis, we contribute to the area of web service composition in REST by proposing three techniques oriented to improve static and dynamic composition of this type of service. First we introduce a technique for static composition proposing a set of fundamental control flow patterns in the context of decentralized compositions of REST services. In contrast to current approaches, our proposal is implemented using the HTTP protocol and takes into account REST architectural principles. Afterwards, we present a technique to improve the dynamic composition in security domain extending ReLL to ReLL-S and allowing a machine-client to interact with secured resources, where security conditions may change dynamically. Finally, we propose SAW-Q, an extension of
Simple Additive Weighting (SAW), as a novel dynamic composition technique that follows the principles of the REST style. SAW-Q models quality attributes, in terms of response time, availability and throughput, as a function of the actual service demand instead of the traditional constant values. We tested recommendations of both techniques by applying common stress tests. Our results validate our main hypotheses indicating improvements with respect to alternative state-of-the-art methods. This also shows that the ideas presented in this thesis represent a relevant contribution to the state-of-the-art of REST service compositions.

Members of the Doctoral Thesis Committee:

ROSA ALARCON
YADRAN ETEROVIC
CECILIA BASTARRICA
ANDRES NEYEM
CESARE PAUTASSO
CRISTIAN VIAL

Santiago, december, 2014
La composición de servicios es uno de los principios de la arquitectura orientada al servicio; esto hace posible el reúso y permite a los programadores combinar los servicios existentes para crear nuevos servicios que podrían ser parte de otra composición. La composición dinámica requiere que los servicios componentes sean elegidos en tiempo de ejecución de un conjunto de servicios con igual o similar funcionalidad. La adopción de los servicios REST en la industria ha ocasionado un creciente número de servicios de este tipo, muchos con similar funcionalidad. Las técnicas existentes de composición dinámica son orientadas a métodos, mientras que el estilo REST está orientado al recurso; y solo considera servicios web tradicionales (WSDL/SOAP). El estilo de arquitectura REST ha atraído un gran interés de la industria debido a las propiedades no-funcionales que este estilo favorece.

En esta tesis, contribuimos al área de composición de servicios REST proponiendo tres técnicas orientadas a mejorar la composición estática y dinámica de este tipo de servicios. Primero, introducimos una técnica de composición estática proponiendo un conjunto de patrones de control de flujo fundamentales en el contexto de composiciones descentralizadas de servicios REST. En contraste con los enfoques actuales, nuestra propuesta es implementada usando HTTP y tomando en cuenta los principios arquitecturales de REST. Después, presentamos una técnica que mejora la composición dinámica dentro del dominio
de la seguridad extendiendo ReLL to ReLL-S y permitiendo a clientes de máquinas interactuar con recursos seguros, donde las condiciones de seguridad podrían cambiar de forma dinámica. Finalmente, proponemos SAW-Q, una extensión de Simple Additive Weighting (SAW), como una técnica novedosa de composición dinámica que sigue los principios del estilo REST. Adicionalmente, SAW-Q modela los atributos de calidad como una función de la demanda de solicitudes que espera atender el usuario en lugar de los tradicionales valores constantes. Nosotros hemos probado estas técnicas aplicando a los servicios compuestos resultantes pruebas de estrés. Nuestro resultado valida nuestra hipótesis principal indicando mejoría respecto a los técnicas existentes. Esto también muestra que las ideas presentadas en esta tesis representan una contribución relevante del estado del arte de la composición de servicios REST.

**Miembros de la Comisión de Tesis Doctoral**

ROSA ALARCON
YADRAN ETEROVIC
CECILIA BASTARRICA
ANDRES NEYEM
CESARE PAUTASSO
CRISTIAN VIAL

Santiago, Diciembre, 2014
1. INTRODUCTION

Web services are software entities that provide a determined functionality, they are platform-independent and can be described, published and consumed following a defined set of standards such as WSDL, UDDI and SOAP. These characteristics emphasize loose coupling between components and allow designers to develop interoperable, and evolving applications that can be massively distributed. Web services also promote the reuse of software, which reduces development costs, increases maintainability, and enables organizations to create composed services by combining basic and other composed services resulting in new software with aggregated value (Erl, 2008).

Service composition is performed statically if it occurs at design-time or dynamically if it occurs at runtime. Also, depending on how the composition is made, it can be manual or automatic. The dynamic and automatic service composition is an active research area due to the benefits of reduced need for pre-installation and configuration of the service composition, adaptability to change and contexts, high decoupling and personalization, smaller development time, and reuse (Erl, 2008). However, static and manual techniques dominate in the industry due to the complexity of discovering services dynamically (WSDL descriptions lack domain semantics) and automatic composition. In addition, virtually all research on service composition focuses on the classic Web services (i.e. based on the WSDL, UDDI and SOAP standards). Currently, emerging services technology such as REST \(^1\) is becoming an alternative to conventional services in the industry.

Unlike classical Web services which are centralized and operation-centric, the REST architectural style is resource-centric (e.g. www.puc.cl/course/12) that are manipulated through a limited set of standard and known operations (e.g. HTTP.GET, POST, PUT, DELETE). REST services are popular because they allow massive scalability, decoupling, and fault tolerance (e.g. Amazon API, Facebook API, etc.). REST has attracted the interest of the scientific community to serve as a supporting framework for defining business processes and service composition. A REST resource should be able to be discovered by:

---

\(^1\)Representational State Transfer.
a machine at runtime, and it should be able to understand the mechanism of interaction with the resource (e.g. to use **GET** to read, and to use **DELETE** to delete it). This property would make possible to determine at runtime (dynamically) if the resource serves the consumer purposes, as well as to determine the order in which the operations, that change the resource state, should be invoked. Thus, it would be possible to generate workflows that implement B2B processes dynamically and automatically.

Currently, this is not possible because the REST resources lack a description that encapsulate domain semantics and can be interpreted by a machine. Moreover, current techniques of service composition are oriented to classic Web services that invoke an unlimited set of operations, rather than resources whose state are transformed by invoking a limited set of operations. The overall objective of this thesis is to contribute to the state of the art research on static and dynamic composition of REST services through the design and implementation of a dynamic composition model. In particular, we propose a RESTful decentralized, stateless composition model, a QoS model focused on security in order to determine the feasibility of service composition at runtime, and a QoS model focused on scalability, throughput and response time in order to dynamically select the best service component automatically.

### 1.1. Background

#### 1.1.1. Service Oriented Architecture

Service orientation has existed for some time and has been used in various contexts with different purposes. The most used form of this term has been to identify approaches that focus on the separation of concerns, which means that the logic required to solve a large problem that can be developed and managed, if such logic is decomposed into a collection of related pieces, and each piece gives a solution to a specific part of the problem (Erl, 2005).

This approach transcends technology but when combined with software architecture, service orientation acquires a technical connotation. Service-oriented architecture (SOA)
is a model in which an application’s logic is decomposed into several smaller units that together implement a larger piece of business logic. SOA promotes that these individual units exist independently but not isolated from each other, in fact they could be distributed. This requires that these units operate on the basis of certain principles that allow them to evolve independently while maintaining uniformity and standardization among them. In SOA these logical units are known as services.

Web services encapsulate business logic and can provide solutions to problems of various sizes. The service logic may include the logic provided by other services, in this case, the service is called a “composed” service, and the logic providers are called “component” services. A component service may be responsible for a single or a complex task. Web services interoperate among them due to a common understanding provided by services’ descriptions. The description of a service determines the service endpoint, the data received and the expected data returned by the service through message passing. In this way, programs or services use a description and a message channel independent from a protocol to interact and create a loosely coupled relationship.

In summary, SOA services maintain a relationship that minimizes dependencies (loose coupling), adhere to communication agreements (service contract), independently manage the logic they encapsulate (autonomy), hide logic that is not relevant to a determined context (abstraction), separate responsibilities in order to promote reuse (reusability), can be coordinated and assembled to form new services (composability), avoid to retain or store information specific to an activity (interaction stateless), and are designed to be described so they can be found and executed through discovery mechanisms (discoverability). The collection of services raises an inventory of services that can be managed independently (Erl, 2008).

SOA establishes an architectural model that aims to improve the efficiency, agility and productivity when developing software. It places services as first-class entities through which the business logics are implemented in alignment with service-oriented computing goals.
1.1.2. Web services

According to the context in which they are used, Web services may assume different roles: providers, clients and intermediaries. A Web service plays a provider role if it is invoked from an external source, and a client role if it invokes a provider service. Client services look for and evaluate which is the appropriate service to invoke based on the provider service description. Intermediary services are those which play a role of routing and processing messages sent between client and provider services until they reach their destination (Erl, 2008).

1.1.2.1. Traditional Web Services

The platform to implement Web services has been traditionally defined by a set of industry standards. This platform is divided into two generations, each associated with a set of standards and specifications (Erl, 2008).

- The first Web services generation is comprised of the following technologies and specifications: Web Service Description Language (WSDL), XML Schema Definition Language (XSD), Simple Object Access Protocol (SOAP) and Universal Description Discovery and Integration (UDDI). These specifications have been widely adopted in the industry; however, they lack information about service quality, which is required to address mission critical functionality at production level.

- The second generation adds extensions (WS-* extensions) to fill the gap, related to service quality, left by the first generation. The main issues addressed by these extensions are service security, transactions, reliable messaging services, among others.

A traditional Web service is comprised of (Erl, 2008):

- A service contract that is, a WSDL document describing the service interface (endpoint, operations and parameters) and an XML schema definition defining data types.
• The program logic. This logic may be implemented by the service itself, or inherited and wrapped by the service so that the legacy functionality can be consumed.

• Message processing logic consists of a set of parsers, processors and service agents. The runtime environment generally provides this logic.

• Software components that implement non-functional requirements defined by the WS-* standards extension.

1.1.2.2. REST Services

REpresentational State Transfer (REST) (Fielding, 2000; Erl, Carlyle, Pautasso, & Balasubramanian, 2012) is an architectural style that underlies the Web. A RESTful service approach is another way of implementing a service that follows the design constraints and requirements specified by this architectural style. Each constraint can have positive and negative impact on various non-functional attributes. The non-functional goals of REST are: performance, scalability, simplicity, modifiability, visibility, portability and reliability. An architecture that lacks or eliminates one of the REST constraints is usually not considered a REST architecture.

The REST architectural style constraints are:

• Client-Server: Enforces separation of responsibilities between two components of the architecture, the client and the server. This establishes a distributed architecture where each component evolves independently. This restriction requires the server to process the requests sent by the client.

• Stateless: Determines that the past communication between the client and the service are not kept stored on the server-side (service), that is, the interaction state is not remembered by the service. This implies that each client request must contain all the information necessary for the service to process the request and respond accordingly, without the use of session information.
Cache: Services responses may be temporarily stored in a Web intermediary component (e.g. routers, gateways, proxies, etc.) and thus avoid the service to process a similar request again, diminishing the workload on the service-side.

Uniform Interface: This constraint states that the architectural components must share a single interface to communicate, which is detailed below.

Layered System. A REST-based solution can contain multiple architectural layers. These layers may be added, modified and rearranged according to the evolvability need of the solution.

Code On Demand: This is an optional restriction that allows that some logic is dispatched from the server to the client, to be executed on the client-side. It allows to customize Web applications.

The REST uniform interface is a set of architectural constraints that differentiates REST from any other style:

- Resources must be uniquely identified (e.g. through a URI), and the identifiers must be opaque to prevent coupling, that is, the structure of a URI shall not include any particular meaning that can be guessed by a customer.

- Resources’ state must be manipulated through operations defined by standard (or ad-hoc) network protocols. For example, for the case of HTTP, the operations are POST that initialize the state of a resource whose identifier is unknown and could possibly create a subordinate resource, GET which obtains a representation including the current state of a resource, PUT that modifies the state of an existing resource, DELETE that indicates a request to eliminate a resource (but could be ignored by the server). GET, PUT and DELETE operations are idempotent and reliable. The POST operation allows an unsafe interaction since the invocation of this operation may cause changes to the server (Fielding, 2000).

- A resource can support multiple representations that encode the state of the resource in a particular format (e.g. XML denoted by application + xml).
The format can be negotiated through HTTP headers to facilitate interoperability between client and server.

- The HATEOAS (Hypermedia as the Engine of Application State) property indicates that the state transitions modeled in a Web application (e.g. buy a book, pay the bill, provide the deliver address, etc.) are served as hyperlinks that indicate the user the set of actions available to him or her at a given time (representation).

1.1.3. Service Composition

Software integration involves connecting two or more applications that may or may not be compatible, even when they are not built on the same platform or were not designed to interact with one another. The growing need for reliable data exchange is one of the strategic objectives of integration. Web services are inherently designed to be interoperable and are built with the knowledge that they will have to interact with a long-range of potential service consumers.

Web service composition is a technique that coordinates the combination of service components operating within an agnostic business process context. A composed service is comprised of component services that have been assembled in order to provide the functionality required to automate a task or a specific business process. These service components may be part of other compositions. The ability of a service to be naturally and repeatedly composable is essential to achieve the strategic goals of service-oriented computing.

When Web services are composed, the business logic of the composed service is implemented by several services, which allows the creation of complex applications by progressively adding components.

Service composition is associated with business processes automation. When a business process workflow is defined, several decision points are created in order to define the dataflow and actions according to variables and conditions evaluated at runtime. A business process definition can be done manually if a person defines the sequence of invocations, or automatically if an algorithm defines such sequence (see Table 1.1). In addition, service
components can be chosen at design time (static composition) or at runtime (dynamic composition).

### 1.1.3.1. Orchestration and Choreography

In service composition, the coordination of service components invocation can follow two styles: orchestration or choreography. The orchestration is a centralized control of a business processes execution; it defines the business logic and the order of service invocation and execution. Unlike the orchestration, the choreography has a decentralized approach; services collaborate and determine its role in the interaction.

Orchestration is the process by which various resources can be coordinated to bring out the logic of a business process. Orchestration technology is commonly associated with a centralized platform for activities management (Mendling & Hafner, 2008). Service orchestration encapsulates a service business process and other services invocations. The most widely used technology in the industry to implement orchestrations is the Web Service Business Process Execution Language (WS-BPEL) (Jordan et al., 2007).

REST’s navigational style naturally supports a workflow style of interaction between components. However, interaction is decentralized, components are loosely coupled and can mutate at any time. This characteristic, called evolvability, poses a challenge to service composition since components (resources) may change unexpectedly. Hence, clients must make few assumptions about resources and must delay the binding with the actual resources up to the invocation-time (dynamic late binding) (Pautasso, 2009a).
REST composition research focuses on orchestration, with JOpera (Pautasso, 2009a) being the most mature framework. In JOpera, control and data flow are visually modeled while an engine executes the resulting composed service. In (Alarcón, Wilde, & Bellido, 2010), control and data flow is modeled and implemented using a Petri Net whereas interaction and communication with the resources themselves is mediated by a service description called ReLL (Alarcon & Wilde, 2010). In (Krummenacher, Norton, & Marte, 2010), control flow is specified in SPARQL and invoked services could be WSDL/SOAP-based endpoints or RESTful services (i.e., resources); from the orchestrator perspective, services are described as graph patterns. In (Stadtmüller & Harth, 2012) resource’s graph descriptions are publicly available (can be discovered using HTTP OPTIONS). A set of constraints regulates when certain controls can be executed on resources (e.g., a required state), so that an orchestrator engine could perform a composition, but no indication is given about how to express such constraints. The two former approaches support dynamic late binding and the hypermedia constraint. An RDF-based approach for describing RESTful services where descriptions themselves are hyperlinked is proposed in (Verborgh, Steiner, Deursen, Van de Walle, & Valles, 2011). The approach is promising for service discovery at a high level of abstraction; however, no support for dynamic late binding is provided and the composition strategy is not detailed.

Choreographies can be described from a global and local (one party) perspective. WS-CDL (Kavantzas et al., 2005) is a W3C candidate recommendation that describes global collaboration between participants in terms of types of roles, relationships, and channels. WS-CDL defines reactive rules, used by each participant to compute the state of the choreography and determine which message exchange will or can happen next. Stakeholders in a choreography need a global picture of the service interaction, but none of them sees all the messages being exchanged even though such interaction has an impact on related parties (Decker, 2006).

In (zur Muehlen, Nickerson, & Swenson, 2005), REST-based service choreography standards evolution is presented. Business processes, modeled as choreographies, are implemented as single resources at a higher level. The lower level comprises the resources
themselves (process instances). Although the approach provides process state visibility, it is not clear whether the higher level corresponds to a centralized orchestration or to a partial view of choreography.

1.1.4. REST services composition

Resources and resource collections are the components in a RESTful scenario (Pautasso, 2009a, 2009c). Unlike WSDL-based services, REST resources have standardized, few, and well-known assumptions at the application level (i.e. the Web) instead of the domain level. Resources must be identified with a URI (Universal Resource Identifier) that binds the proper semantics (at the application level) to the resource ((Fielding, 2000) section 6.2.4), and must be manipulated through links and controls (i.e. an HTML form) embedded in a resource’s representations (e.g., HTML page). Representations dynamically determine the set of resource’s state transitions available to consumers (Hypermedia as the engine of application state (Fielding, 2000)).

Composition requirements that are specific to REST are dynamic late binding, that is, the URI of the resource to be consumed can be known only at runtime; the composed service must also support the REST uniform interface; data types are known at runtime; content negotiation must be allowed; and clients must be able to inspect the composed service (Pautasso, 2009c; Nierstrasz & Meijler, 1995).

Currently there are no proposals for automatic and dynamic composition of REST services. REST services composition research is mainly focused on static service composition (Pautasso, 2009c; Nierstrasz & Meijler, 1995). The most comprehensive work in the area is the JOpera project (Pautasso & Alonso, 2005) that aims to provide a similar implementation of BPEL for REST. JOpera allows the static composition of Web services by means of two graphs. The first, Flow Control Dependency Graph (CFDG) describes the sequence of invocation of services, and the second, the Data Transfer Flow Graph (DFTG) defines the relationship between the data inputs and outputs when invoking the REST components. Unlike BPEL, service composition in JOpera allows that the service URI to be invoked is
known at runtime (dynamic late binding), it also supports the uniform interface, and content negotiation at runtime. However, JOpera does not consider the HATEOAS constraint, and the CFDG is performed by a centralized process engine, in a stateful fashion which negatively impacts on service scalability.

Similarly, Bite (Rosenberg, Curbera, Duftler, & Khalaf, 2008b) proposed a BPEL-inspired composition language describing data and control flow. Bite partially supports a uniform interface based on HTTP, dynamic data types, and state inspection (only GET and POST). Regarding dynamic late binding, Bite can generate URIs to created resources, but cannot inspect the service’s responses and find out the links provided by the service (HATEOAS).

Decker (Decker, Lüders, Overdick, Schlichting, & Weske, 2008) presents a formal model for implementing REST processes based on Petri nets, they uses PNML (Petri Net Markup Language) as a language for specifying the states, transitions and an execution engine. Decker considers only partial support for dynamic late binding since it can generate URIs for resources created but cannot inspect the responses and retrieve the embedded links. In this model, control flow is driven by the decisions of a human user. Guard conditions, such as authentication are not supported and like others (Xu, Zhu, Liu, & Staples, 2008) it assumes XML as the content type for representations leaving out REST content negotiation constraint.

Zhao (H. Zhao & Doshi, 2009) presents an automatic service composition approach for REST typifying and semantically describing the services in three categories according to the operations that the service can perform (GET, PUT, POST, DELETE). Service composition is automated by a first-order logic situation calculus representing changes and actions. This approach does not consider several of the REST principles such as HATEOAS, content negotiation, opaque URIs, nor dynamic late binding.
1.1.5. QoS

The quality of service (QoS) is a combination of several qualities or properties of a service (Menasce, 2002). Dynamic composition of Web services requires the consumer to choose the best component services that satisfy the functional and non-functional requirements of the composition. Non-functional requirements involve attributes of quality of service as response time, availability, security and performance (Menasce, 2002).

The definition and measurement of these attributes of service quality vary by approach. For example, the response time can be measured as the average of the results obtained of the last 15 minutes, or an average vector of 15-minute intervals during the day.

The quality attributes of a service depend on the load current at which the service is submitted, however the technical description of these quality attributes of a service are focused on the description of a discrete value associated with each property.

1.1.5.1. QoS en REST

For the case of REST, research initiatives on service composition are fairly recent, and interest on QoS properties have focused mainly on security. For instance, in (Kübert, Katsaros, & Wang, 2011) a RESTful service API is defined to support service-level agreements based on the WS-Agreement standard. Agreements (and templates) are REST resources encoded in the application/xml media-type, whose life cycle and state is handled by means of HTTP verbs. Graf et al. present a server-side authorization framework based on rules that limit access to the served resources according to HTTP operations (Graf, Zholudev, Lewandowski, & Waldvogel, 2011). In (Field, Graham, & Maguire, 2011a) a server-side obligation framework allows designers to extended existent policies with rules regulating users’ information access. Rules may trigger additional transactions (e.g., sending a confirmation e-mail, register the information access attempt in a log) or even modify the response content or the communication protocol (e.g., require HTTPS). Allam (Allam, 2012) aims to homologate WSDL-based and RESTful services by considering them black boxes, where interaction occurs as simple message passing between clients and servers. Security policies can be placed when receiving or sending a message as well as locally (e.g., at the server
or client side). This vision does not consider complex interaction involving third parties (e.g., OAuth), or service’s interface heterogeneity (Hongbin, Fengyu, & Tao, 2012), where industry standards are implemented in various ways. We assume a workflow perspective where data is transformed locally in successive steps until the message constraints required by the service provider are achieved. In addition, clients, servers, and third party services may engage in an interaction that implements sub-workflows.

1.2. Limitations

The REST architectural style offers several advantages when compared to traditional Web services in terms of non-functional attributes, namely, low latency (small response time), massive scalability (due mainly to statelessness, cacheability, and Web intermediaries), modifiability, and simplicity, among others. However, REST considers humans as its principal consumer and they are expected to drive service discovery and state transition by understanding the representation content. The lack of a REST machine-readable description forces service providers to describe their APIs in natural language, which makes difficult to properly design machine-clients that can perform discovery and service composition in an automated way.

Furthermore, REST has been used in the industry as an infrastructure layer for supporting massive service provisioning in the form of Web APIs which have given rise to the evolution of application ecosystems that constitutes simple service composition basically consuming services in a sequential control-flow pattern. A massively used example of complex control-flow is the OAuth protocol that constitutes an orchestration where various parties cooperate through redirections in order to implement a workflow. The control-flow patterns involved in the OAuth are sequence and conditional execution. Researchers also recognize the lack of a complex service behavior model in REST as one of the difficult issues to be addressed in order for REST to support rich SOA (Issarny, 2009). Research proposals for REST service composition focus either on operation-centric (Rosenberg, Curbera, Duftler, & Khalaf, 2008a), centralized (Pautasso, 2009a), stateful and static service composition, violating REST architectural constraints.
Traditional Web services dynamic and automatic composition focuses on diverse tech-
niques (e.g. planning, graph models, semantic models, QoS constraints, etc. in order to
make possible the automatic or dynamic generation of a composition plan (i.e. a workflow)
and the selection of the services components (D’Mello, Ananthanarayana, & Salian, 2011).
QoS has been a focus of extensive research and is being recently addressed in REST but
not in its capacity for determining service composition on runtime. Furthermore, the very
nature of the non-functional attributes makes it hard to reduce them to uniform representa-
tions such as numbers or Boolean values. Security for instance requires not only a complex
combination of algorithms but also protocols (in the case of OAuth a choreography) in
order to determine whether a service can be composed or not. For the case of response
time, availability and throughput, research in Web sites provisioning, also known as capac-
ity planning, have demonstrated the variables such as the user demand and cache policies
are relevant to determine the quality of a Web site (equivalent to a single REST service).
Hence such quality attributes cannot be reduced to simple numeric variables as is the case
of traditional QoS-aware compositions in SOA.

1.3. General Goals

The overall objective of this thesis is to facilitate the dynamic composition of REST
services. In particular, we propose:

(i) A decentralized RESTful, stateless composition model that places an emphasis
on service behavior (control-flow),
(ii) A QoS model focused on security in order to determine the feasibility of com-
position at runtime, and
(iii) A QoS model focused on scalability, throughput and response time in order to
select the service component automatically at runtime.
1.4. Hypothesis

The main hypothesis of this thesis is related to the relevance of decentralized service composition, in that "a stateless and decentralized composition technique follows the REST architectural style constraints and generates a composite service with the same REST properties whereas a centralized composition does not". Second, "QoS-based dynamic and automatic RESTful service composition must take into account the characteristics of the non-functional attributes in order to preserve REST architectural constraints in the composite service".

1.5. Thesis Work and Main Contributions

The first part of this thesis presents a technique for decentralized, hypermedia-centric and stateless REST services composition. The proposed technique models services behavior through a set of well-known, simple and complex control-flow patterns and focuses on the following REST constraints, namely, client-server interaction, layered client-server, client-stateless-server, and the uniform interface. The uniform interface (messages between server and client are self-descriptive) was extended through 300 HTTP redirection codes in order to include control-flow information in the message; hypermedia (Web linking) is used as the application state machine. We present the advantages of this approach (centralized Vs. decentralized) in terms of response time, availability and throughput, which are non-functional goals in REST. By definition a stateful approach is not RESTful so that such comparison is left out.

The second part of this thesis describes automatic and dynamic REST services composition based on non-functional attributes. In this part two QoS domains are analyzed, the first corresponds to security and the proposal is a hybrid between static and dynamic service composition in the sense that a service description (created at design-time) is used to determine the feasibility of service composition (at runtime) and actually enforce the composition (at runtime). The environment is decentralized, stateless and promotes the use of hypermedia as a state machine.
The third part of the thesis focuses on a second QoS domain considering scalability. That is, the non-functional attributes: response-time, throughput and availability were used to support automatic and dynamic service composition. In this case, a queuing theory-based model was used to identify response-time, throughput and availability. These models were later used in a technique called SAW-Q to identify the compositions with the highest quality. SAW-Q was compared with a well-known technique, SAW (from which SAW-Q was derived) with good results.

Accordingly, the main contributions of this thesis are:

(i) Part 1
   (a) A decentralized, stateless, hypermedia-centric technique for designing composed service behavior in REST.
   (b) A set of control-flow patterns that implement decentralized, stateless, hypermedia-centric REST service composition.

These contributions were published in the following journal:

(ii) Part 2
   (a) A centralized, hybrid (design-time and runtime) and manual REST service composition based on the security QoS attribute.
   (b) A security domain model.
   (c) An extension to ReLL, a hypermedia REST service description, that considers the security domain model in order to determine the feasibility of consuming a protected service and actually execute the OAuth choreography:

These contributions were published in the following journal:

(iii) Part 3
(a) A decentralized, dynamic REST service composition technique based on the response-time, throughput and availability QoS attributes.

(b) A queuing theory-based REST model.

(c) A response-time, throughput and availability models based on the proposed queuing model.

(d) SAW-Q, a refinement of SAW, a technique for scoring service compositions in terms of various variables such as response-time, throughput and availability that is sensible to user demand and service capacity.

These contributions were submitted to the following journal:


Other articles were also produced during this investigation:


### 1.6. Document Structure

The remainder this thesis is organized as follows: Chapter 2 presents the journal that summarizes the results of the first part of the thesis. Chapter 3 explores the attributes of service quality in the RESTful services composition, focused on security. Chapter 4 presents the dynamic composition approach based on SAW-Q and scalability (response-time, throughput, availability). Chapter 5 presents the main conclusion of this thesis and future research. The chapters of this thesis are self-contained, that is, that can be read independently.
Chapter 2 consists of the paper: Control-Flow Patterns for Decentralized RESTful Service Composition: The paper describes a technique for decentralized REST services composition that takes into account the constraints of REST architectural style in the composition process. The proposed technique involves the creation of control-flow patterns that allow seamless interaction between the client and the composite service and uses hypermedia as a state machine. The main idea is to implement control-flow patterns through callbacks and redirections. Finally, we discuss the impact of our design decisions according to the architectural principles of REST.

Chapter 3 consists of the paper: QoS aware descriptions for RESTful service composition: security domain: We explore QoS aware RESTful services composition, which is characterized by a decentralized, stateless and hypermedia-driven environment. We focus particularly on the security domain since current security practices on the Web illustrate the differences between both the centralized, function-based approach and the decentralized, hypermedia and resource-based approach. We rely on ReLL (a REST service description) that can be processed by machine-clients in order to interact with RESTful services. Our approach identifies key security domain elements as an ontology. Elements serve to model hypermedia-based, decentralized security descriptions supporting simple and complex interaction such as protocols and callbacks. In this paper, we propose an extension to ReLL that considers security constraints (ReLL-S) and allows a machine-client to interact with secured resources, where security conditions may change dynamically. A case study illustrates our approach.

Chapter 4 consists of the paper: SAW-Q: An approach for dynamic composition of REST services: We propose SAW-Q an extension of SAW, as a technique of dynamic composition according to the principles of the REST style and considering quality attributes modeled as a demand function instead of the traditional constant values. Our model is much more accurate when compared to real implementation, positively improving dynamic composition.
2. CONTROL FLOW PATTERNS FOR DECENTRALIZED RESTFUL SERVICE COMPOSITION

2.1. Abstract

The REST architectural style has attracted a lot of interest from industry due to the non-functional 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 and 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 paper 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.

2.2. 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 & Ruby, 2008). 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 document and use the SOAP (Mitra, Lafon, et al., 2003) 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 & Benatallah, 2003) (Russell, Arthur, van der Aalst, & Mulyar, 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 paper, we present a subset of the control flow patterns well known in the business process management community (van der Aalst, ter Hofstede, Kiepuszewski, & Barros, 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 & Wilde, 2010b) 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 paper is organized as follows: Section 2.3 introduces basic concepts regarding REST services composition; Section 2.4 presents a motivating scenario and the rationale of our approach. Section 2.5 presents our proposal for a set of basic and advanced control flow patterns for REST services. Our reference implementation is described in Section 2.6, while Section 2.7 presents a comparative evaluation of different QoS attributes. Section
2.8 discusses the impact of our approach on key REST architectural properties. Finally, Section 2.9 presents our conclusions and proposals for future work.

2.3. 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 & Wilde, 2009). In this paper 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 2.3.2) and
HTTP interactions (see Section 2.3.3). Our approach is both stateless and decentralized (as exemplified in Section 2.4), 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 paper we rely on ReLL (Section 2.3.4), which is a hypermedia-centric description of RESTful services.

2.3.1. 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 (Peltz, 2003) (Benatallah, Sheng, & Dumas, 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 run-time, 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 (Pautasso, 2009a; Nierstrasz & Meijler, 1995). Visual editors supports the design of manual, centralized, stateful service composition that can be executed by an orchestration engine; it also addresses control and data flow, 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.,
2008a) proposes a BPEL-inspired workflow composition language describing both control and data flow. 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 (Decker et al., 2008), presents a formal model for REST process enactment based on Petri Nets, they use 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 & 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 the 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 (X. Zhao, Liu, & Clapworthy, 2011) also proposes 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 RESTdesc 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, Zhang, Huang, & Cao, 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.

### 2.3.2. Links Processing

Clients process the links embedded in resource representations in the following order: protocol, hypermedia, and application (Ivan Zuzak, 2011) (Figure 2.1). 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 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).

![Figure 2.1. Links processing levels](image)

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 non-structured 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 & 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, Marucheck, & Handfield, 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.

2.3.3. 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).

2.3.4. 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 run-time, 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, Moreau, Ryman, & Weerawarana, 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 automation 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, Alarcon, & Sepulveda, 2011).

2.3.4.1. ReLL

. The Resource Linking Language describes a set of assumptions that clients expect
from servers such as resources identifiers, representations, links to other resources, con-
trols 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, par-
tial 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., follow-
ing 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 grace-
fully in the sense, that they must stop their behavior and notice the cause of failure (i.e., an
assumption has failed). Partial knowledge in ReLL prevents the automatic generation of
client code, and hence reduces coupling between clients and servers.

Figure 2.2 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 run-time 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 controls 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 paper we will use Web Linking for serving the hypermedia controls to user agents.

### 2.4. 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 the ones 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, Hsu, & Ladin, 2001). Consider the business process shown in Figure 2.3. In this scenario, the business process of a certain Commerce composes various other processes, Sellers, Banks, and Delivery organizations, as well as utility services such as Authentication and Email services.

**Figure 2.3.** The Commerce REST service composes the REST services Seller, two Banks, a Delivery, Delivery, Email and Authentication.
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 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 are 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. Subsections 2.4.1 and 2.4.2 discuss the current approaches when dealing with centralized versus decentralized design and state-handling, respectively. They also explain the tradeoffs we face when pursuing a fully decentralized, RESTful compliant service composition. The conclusion of this discussion is presented in section 2.4.3.

2.4.1. Centralized versus Decentralized control flow

In Figure 2.4 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.

![Diagram of REST service interaction](image)

**Figure 2.4.** Centralized implementation of the composed Commerce REST service.

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.

![Diagram of decentralized implementation of the composed Commerce REST service.](image)

**Figure 2.5.** Decentralized implementation of the composed Commerce REST service.

An alternative is to rely on a fully decentralized style based on asynchronous messages and callback connectors as shown in Figure 2.5. 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 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).

2.4.2. 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 (Figure 2.6.a1, 2.6.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 (Figure 2.6.b1, 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. In this way it is also possible to inspect the state of the business process at any time (Figure 2.6.b3, 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 2.6.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 a 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 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 2.6.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 2.6.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 2.6.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 2.5.

2.4.3. 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 redirection 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 Figure2.5).

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 2.5), 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.

### 2.5. 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 (van der Aalst et al., 2003), (Russell et al., 2006).

#### 2.5.1. 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 2.5, 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 2.7).

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 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 2.5, 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
FIGURE 2.8. ReLL descriptors are fetched considering the root resource of a 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 2.8). 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 2.9). 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 the address of a particular component resource to be invoked as well as the method; that is, this case is not different from a sequential invocation.
2.5.2. 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).

![Diagram of alternative control flow pattern](image)

**Figure 2.10.** An alternate control flow pattern implemented for REST and HTTP

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 2.10, message 2). Since in REST user agents keep application-state information (Ivan Zuzak, 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 thesis. 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).

2.5.3. 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.

![Diagram of Iteration, Structured Loop - while variant](image)

**Figure 2.11.** An iterator control flow pattern implemented for REST and HTTP

This interaction begins when the composed resource issues a 313 message (Figure 2.11, 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).

2.5.4. 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 2.12.a), just one of them (Structured Discriminator, Figure 2.12.b), or a fixed number (Structured Partial Join, Figure 2.12.c). Finally, for the case of Local Synchronization Merge (also called Selection, Figure 2.12.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...
Figure 2.12. A parallel control flow pattern implemented for REST and HTTP

(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 2.12 shows the details for the pattern. Interaction starts when the composed resource issues either a 314, 315, 316 or 317 message (Figure 2.12, 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 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.

2.6. Implementation

The e-commerce scenario depicted in Figure 2.3 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 snippet log below. The services were developed using the RESTify framework \(^1\) and the client was implemented using the basic

\(^1\)mcavage.github.com/node-restify/
HTTP client library of Node.JS. Below we 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 fragment below.

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:
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 snippet below.

### Step 4.

**Request:**

```
PUT /commerce/state3 HTTP/1.1
Content-type: json
```

**Response:**

```
312 Alternative
Link: </bank1>;method="GET",
     </bank2>;method="GET"
Callback: /commerce/state4
```

### Step 5.

**Request:**

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 snippet below.

```
Step 5.
Request:
```
<table>
<thead>
<tr>
<th>PUT /commerce/state4  HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-type: json</td>
</tr>
<tr>
<td>[Payment results, Chosen products]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response:</th>
</tr>
</thead>
<tbody>
<tr>
<td>317 Selection</td>
</tr>
<tr>
<td>Link: &lt;/emailSender&gt;;method=&quot;POST&quot;,</td>
</tr>
<tr>
<td>&lt;/deliveryService&gt;;method=&quot;POST&quot;</td>
</tr>
<tr>
<td>Callback: /commerce/state5</td>
</tr>
<tr>
<td>[Conditional expression written in JavaScript]</td>
</tr>
<tr>
<td>[Purchase order, email details, delivery details]</td>
</tr>
</tbody>
</table>

-Two request messages are sent concurrently to the Email and Delivery services, as instructed-

<table>
<thead>
<tr>
<th>Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /emailSender  HTTP/1.1</td>
</tr>
<tr>
<td>Content-type: json</td>
</tr>
<tr>
<td>[Purchase order, email details]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /deliveryService  HTTP/1.1</td>
</tr>
<tr>
<td>Content-type: json</td>
</tr>
<tr>
<td>[Purchase order, delivery details]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response:</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 OK</td>
</tr>
<tr>
<td>[Mailing confirmation]</td>
</tr>
</tbody>
</table>
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]
```

## 2.7. 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, Di Penta, Esposito, & Villani, 2005). Each control-flow pattern can be modelled with a finite state machine representing the execution paths (Figure 2.13).
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 & 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 & 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 i.e. for the worst possible cases. Control-flow patterns such as iteration, alternative, and parallelism can be reduced or transformed to a sequence model (Alrifai & Risse, 2009; Cardoso, Sheth, Miller, Arnold, & Kochut, 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 1000 ms (1 second) execution time and supports up to 600 clients concurrently connected. The service component has a processing capacity (throughput) of 600req/sec, an average response time of 1000ms, 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 its 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).
2.7.1. 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 2.14 shows a comparison of the composed service response time under a centralized vs. 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 \( w \leq 100 \text{ req/sec} \)), then the response time corresponds to the sum of each component service response time. The figure shows the results where a composition includes \( n = 2 \ldots 10 \) 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 \( L = 200 \text{ req/sec} \)) 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.
2.7.2. 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 2.15 analyzes the composed service availability for both approaches in various scenarios where the composed service includes \( n = 2 \ldots 10 \) component services. Under the centralized approach, availability decreases regardless 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.

2.7.3. Throughput

Service throughput is the amount of workload (request in this case) that the service can process in a unit of time. Figure 2.16 shows the throughput for both approaches with a configuration similar to the previous cases (\( n = 2 \ldots 10 \) service components). In both approaches the composed service throughput decreases if the number of service 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.

2.8. Discussion

2.8.1. Impact on REST architectural properties

The REST architecture style yields to applications (e.g., the Web) non-functional properties like openness, extensibility, high scalability (Taylor, Medvidovic, & Dashofy, 2009), performance, simplicity, modifiability, visibility, portability, and reliability (Fielding, 2000).

2.8.1.1. 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 mediatype, 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 2.4), this extensibility is lost since resources are fetched by an intermediary that has no use for scripts, css or applets. In a decentralized implementation (figure 2.5), out-of-band interaction between the end-user and the component service may occur, taking advantage of extensible components (scripts, applets, css).

2.8.1.2. 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 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.

2.8.2. Backwards compatibility

In this paper, 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 2.5, we believe that the definition of new codes for each pattern allows advanced architectural components to have 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.

\(^2\)“http://OAuth.net/”
\(^3\)“http://OpenID.net/”
callback URIs, chains of callback URIs, security keys, signatures, etc), causing the leaking 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.

2.9. Conclusion

We 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 (http://composedservice/state). In the evaluation section we can observe that our decentralized approach allows composed services significantly 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 non-functional 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 tradeoffs for designing such 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 paper 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 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 2.8; 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, e.g., 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 paper. 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, Risse, & Nejdl, 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.
3. QOS AWARE DESCRIPTIONS FOR RESTFUL SERVICE COMPOSITION: SECURITY DOMAIN

3.1. Abstract

Current research on QoS aware service composition focuses on a WSDL/RPC service paradigm, characterized by a centralized, synchronous, and stateful approach. In this paper, we explore QoS aware RESTful services composition, which is characterized by a decentralized, stateless and hypermedia-driven environment. We focus particularly on the security domain since current security practices on the Web illustrate the differences between both the centralized, function-based approach and the decentralized, hypermedia and resource-based approach. We rely on ReLL (a REST service description) that can be processed by machine-clients in order to interact with RESTful services. Our approach identifies key security domain elements as an ontology. Elements serve to model hypermedia-based, decentralized security descriptions supporting simple and complex interaction such as protocols and callbacks. In this paper, we propose an extension to ReLL that considers security constraints (ReLL-S) and allows a machine-client to interact with secured resources, where security conditions may change dynamically. A case study illustrates our approach.

3.2. Introduction

A Web service exposes data and functionality that can be consumed by humans or other services. Traditionally, Web services provide a WSDL (Web Service Description Language) document (Chinnici et al., 2009) describing its interface and the conditions to be consumed (e.g. with WS-PolicyAttachment annotations (A. S. Vedamuthu et al., 2007)). The process of combining the functionality of two or more services (components) into a new one (composite) that provides aggregated value is called service composition. The selection of suitable components may be manual or automatic (i.e., determined by an
algorithm); the composite behavior (i.e., the services invocation order and the data transformation) can be determined also in a manual or automatic fashion, either at design-time (static) or run-time (dynamic) (Alonso, Casati, Kuno, & Machiraju, 2004).

To determine the suitability of a service to be part of a composite, additional information, such as non-functional properties, usually called QoS (Quality of Service), can be considered (Kritikos & Plexousakis, 2009). Most research efforts on QoS aware service composition are focused on WSDL-based services characterized by a centralized model (orchestration), whereas RESTful services (Richardson & Ruby, 2008) are characterized by a decentralized, stateless and hypermedia-based approach (Fielding, 2000). Representational State Transfer (REST) is the architectural style that underlies the Web and provides it with high scalability, performance, and evolvability - properties that are also desirable for Web services. RESTful services are gaining momentum in the industry, but the architectural differences between WSDL-based and RESTful services (i.e., centralized vs. distributed, operation-centric vs. resource-centric, tightly coupled vs. loosely coupled (Pautasso & Wilde, 2009)), make it unclear how current research on WSDL-based QoS aware service composition can be applied to RESTful services.

Security is a QoS property that is highly relevant for Web services composition (Maamar, Narendra, & Sattanathan, 2006). The security domain can be hardly modeled by a single metric since it addresses various dimensions (e.g., confidentiality, authorization, etc.) that can be combined. Security requirements can be determined statically (e.g., data must be encrypted), but also may imply a dynamic behavior (e.g., authentication protocols) provided through choreographies on the Web (e.g., the OAuth authorization protocol (Hammer-Lahav, 2010)). Due to its relevance and complexity, we have chosen the security domain to illustrate RESTful security aware service composition.

In Section 3.3, we provide a summary of related work on RESTful services composition, and Section 3.4 focuses on QoS aware approaches, particularly for the security domain. Section 3.5 presents our approach, called ReLL-S, as an extension of ReLL, which is a RESTful service description (Alarcon & Wilde, 2010; Alarcón & Wilde, 2010b). ReLL-S
allows machine-clients (e.g., other Web services, intelligent user agents, etc.) to process and understand a set of security constraints in order to interact with a secured service. The security domain is modeled as an ontology (Section 3.5.1) in which fundamental concepts serve as elements of the ReLL-S constraints-set document. We illustrate and validate ReLL-S expressiveness with the most widely used security approaches for the Web (Maleshkova, Pedrinaci, Domingue, Alvaro, & Martinez, 2010) (Section 3.6), and in Section 3.7, we present implementation details for a machine-client (RESTler) and a case study based on Flickr’s authentication mechanism. Finally, Section 3.8 presents our conclusions. Unlike WSDL-based security approaches, we can support security tasks requiring simple and complex interaction (e.g., protocols implemented as choreographies with asynchronous communication). ReLL-S is minimal but expressive enough to be executed; it is also a loose contract that cannot guarantee accuracy but includes the mechanisms for identifying failures and changes while allowing evolvability and keeping components loosely coupled.

3.3. RESTful Service Composition

Service composition includes six dimensions: a component model, a data access model, a service selection model, an orchestration model, a transactions model, and an exception handling model (Alonso et al., 2004). The component model defines the assumptions about components; the fewer the assumptions the more flexible and heterogeneous components are allowed. Resources and resource collections are the components in a RESTful scenario (Pautasso, 2009a, 2009c). Unlike WSDL-based services, REST resources have standardized, few, and well-known assumptions at the application level (i.e., the Web) instead of the domain level. Resources must be identified with a URI (Universal Resource Identifier) that binds the proper semantics (at the application level) to the resource ((Fielding, 2000) section 6.2.4), and must be manipulated through links and controls (e.g., an HTML form) embedded in a resource’s representations (e.g., HTML page). Representations dynamically determine the set of resource’s state transitions available to consumers (Hypermedia as the engine of application state (Fielding, 2000)).
The data access model defines how data is specified and exchanged between components (Dustdar & Schreiner, 2005). Unlike WSDL-based services, representations’ standardized data models do not refer to domain-specific information (e.g., XML document types) but to generic models, or media types (e.g., text/html, etc.), to be interpreted by generic clients (e.g., Web browsers). State information interchanged between clients and servers follows the rules of a network protocol (e.g., HTTP). Messages must include all the information required by the servers to process them (stateless principle). Application state (i.e., the collection of state information exchanged during client-server interaction) is stored on the client-side.

The service selection model determines how a service component is chosen to be part of a composition, either in a static (at design-time) or dynamic (at run-time) way. Based on the hypermedia constraint, servers explicitly steer clients across a path of resources and state transitions. Clients must inspect representations for links and controls in order to discover service components’ URIs. URIs can be also generated from the information conveyed in the representation and ad-hoc rules, or can be chosen from a registry. However, hypermedia decouples clients from servers and allows servers to evolve independently (i.e., change the host name or the URI structure at run-time) without breaking the clients. For media types with limited support (e.g., binary), controls can be embedded in HTTP headers (Web Linking (Nottingham, 2010)).

The orchestration model defines the order and conditions of service components invocation. When invocation control is centralized, the style is called orchestration (Mendling & Hafner, 2008), and choreography otherwise. Orchestrations can be specified in languages such as WS-BPEL (Jordan et al., 2007) and executed by an orchestration engine. REST’s navigational style naturally supports a workflow style of interaction between components. However, interaction is decentralized, components are loosely coupled and can mutate at any time. This characteristic, called evolvability, poses a challenge to service composition since components (resources) may change unexpectedly. Hence, clients must have few assumptions about resources and must delay the binding with the actual resources up to the invocation-time (dynamic late binding) (Pautasso, 2009a).
REST composition research focuses on orchestration, with JOpera (Pautasso, 2009a) being the most mature framework. In JOpera, control and data flow are visually modeled while an engine executes the resulting composed service. In (Alarcón et al., 2010), control and data flow is modeled and implemented using a Petri Net whereas interaction and communication with the resources themselves is mediated by a service description called ReLL (Alarcon & Wilde, 2010). In (Krummenacher et al., 2010), control flow is specified in SPARQL and invoked services could be WSDL/SOAP-based endpoints or RESTful services (i.e., resources); from the orchestrator perspective, services are described as graph patterns. In (Stadtmüller & Harth, 2012) resource’s graph descriptions are publicly available (can be discovered using HTTP OPTIONS). A set of constraints regulates when certain controls can be executed on resources (e.g., a required state), so that an orchestrator engine could perform a composition, but no indication is given about how to express such constraints. The two former approaches support dynamic late binding and the hypermedia constraint. An RDF-based approach for describing RESTful services where descriptions itself are hyperlinked is proposed in (Verborgh et al., 2011). The approach is promising for service discovery at a high level of abstraction; however, no support for dynamic late binding is provided and the composition strategy is not detailed.

Choreographies can be described from a global and local (one party) perspective. WS-CDL (Kavantzas et al., 2005) is a W3C candidate recommendation that describes global collaboration between participants in terms of types of roles, relationships, and channels. WS-CDL defines reactive rules, used by each participant to compute the state of the choreography and determine which message exchange will or can happen next. Stakeholders in a choreography need a global picture of the service interaction, but none of them sees all the messages being exchanged even though such interaction has an impact on related parties (Decker, 2006). In (zur Muehlen et al., 2005), REST-based service choreography standards evolution is presented. Business processes, modeled as choreographies, are implemented as single resources at a higher level. The lower level comprises the resources themselves (process instances). Although the approach provides process state visibility, it is not clear
whether the higher level corresponds to a centralized orchestration or to a partial view of a choreography.

In addition, due to the synchronous messaging nature of the HTTP protocol, clients must wait for server’s responses in order to continue their work. A response may be delayed or lost if the service invokes third party services, either in a choreography or an orchestration fashion. Asynchronous interaction, where servers notify clients about available responses may alleviate this problem, but implementing the approach is far from trivial (Bellido et al., 2011). Although REST requires clients’ requests to be self-contained, and servers’ responses include links to steer clients, current practices (e.g., Web APIs) do not provide such guidance and force developers to hardcode URLs into the clients, increasing coupling. Furthermore, the lack of a machine-readable service description forces developers to hardcode their understanding of the service interface. These characteristics limit automatic service composition support, either under the choreography or orchestration paradigm.

3.4. Refining service composition with QoS

QoS properties are used to refine the suitability of a candidate service as a component. They include domain-independent categories, such as performance, dependability, security, transactions integrity, network, infrastructure, and costs. In WSDL/SOAP-based services, QoS properties are represented by a set of standards collectively known as WS-*, which have been criticized due to its complexity, and requires a set of middleware tools and infrastructure. **WS-Policy Attachment** makes it possible to annotate WSDL descriptions with QoS properties so that the service interface can explicitly declare its requirements to be consumed. WSDL descriptions can be also extended to include **semantic annotations** (SAWSDL), by means of URIs that relate various aspects of the description (e.g., data types, operations, etc.) to a semantic model (e.g., an ontology) that provides additional information at the semantic level. Both semantic and QoS annotations enrich the service interface so that tasks such as automatic discovery and service composition can be facilitated. For instance, in (Medjahed & Atif, 2007) the WS-Policy standard is extended to
include semantic capabilities so that composite services that consider QoS properties can be automatically generated from high-level user specifications. QoS properties are treated in broader terms; for instance, security is modeled as a simple Boolean value indicating whether certain security support is provided (e.g., the service supports a specific encryption algorithm). A similar approach is presented in (Kritikos & Plexousakis, 2009), where QoS properties are narrowed down to specific concepts. For instance, response time is a performance attribute measured in time units (e.g., seconds); flexibility is a dependability attribute measured as inflexible, flexible, or very-flexible values, whereas security attributes are mostly defined as Boolean values indicating the presence of some security vulnerability.

There is no such standardization effort in the RESTful services domain; instead, Web security has evolved organically to cope with technological and user requirements, resulting in lightweight and decentralized approaches that are discussed in the following section.

The need to keep interactions secure is fundamental for composed resources in the Web and so has the need of describing services’ security capabilities (Movahednejad, Ibrahim, Sharifi, Selamat, & Tabatabaei, 2011). Unlike other QoS properties, security can be defined in three dimensions; confidentiality, integrity and identity. When two services interchange messages where the content is secured in such a way that a third actor intercepting the message cannot read and understand it, then the interaction guarantees confidentiality. If a third party did not alter the received message, the interaction guarantees message integrity. If the receiver can check that the sender is really who the message says it is, the interaction guarantees identity. WSDL-based service security capabilities are described by a family of WS-* industry standards (e.g., WS-Security (Nadalin, Kaler, Monzillo, & Hallam-Baker, 2004), WS-Policy (A. Vedamuthu, Orchard, Hondo, Boubez, & Yendluri, 2006), etc.) along with well-known techniques sometimes implemented in an ad-hoc fashion (e.g., signature) (Allam, 2012).
In (Movahednejad et al., 2011), security-aware service composition approaches are classified as semantic or syntactic. The former rely on a formal model (typically ontologies), and mechanisms for determining whether two or more services can interact (matching) towards achieving some goal. The latter is based on models (typically XML) that specify how security is supported. That is, security is either explicitly considered when designing the composition control-flow, or it is focused only on access control, or it is defined through policies, or it is defined in a formal model executed by a system. Our approach falls under the syntactic control-flow category since we use security descriptions that aim at explicitly regulating the consumer-provider interaction. Unlike previous approaches, we consider that such interaction can be as complex as a workflow itself.

In (Dell’Amico, Serme, Idrees, Santana de Olivera, & Roudier, 2012) the need of distributed enforcement of security policies and policy management is acknowledged, and a solution based on hierarchical domains and attributes that are transversal to various entities is proposed. Policies are enforced by means of distributed monitors that filter and control the information flowing between services. A more limited approach is presented by (Rouached, 2012), where an access control mechanism is implemented based on an event calculus model and compatible monitors that guarantee the enforcement of a security policy. Similarly, we identify a scope specifically related to Web resources and we also consider attributes (tokens) in order to apply security policies that are transversal to a set of resource categories. Unlike (Dell’Amico et al., 2012), we do not consider monitors but we distribute the security policy enforcement responsibility to both the client and the server. That is, the server specifies how it expects to be addressed and the client must fulfill such expectations (that may unexpectedly change in time). There are no intermediaries that regulate such interaction, since in an open and heterogeneous environment, such as the Web, this approach will introduce coupling and will limit service evolvability.

Regarding semantic approaches, security constraints have been modeled in various ways. In (Maamar et al., 2006), a security model limited to encryption and decryption (confidentiality) of messages is proposed. Authors do not consider authentication, authorization, or integrity, but they identify various granularity levels, such as a single Web
service, an instance of such service, and a composed service. The model does not consider existing industry standards, such as WSDL or WS-* and is limited to single endpoints, so it is not clear how the approach could be applied to a RESTful service, where a security policy may involve several resources. Others (Carminati, Ferrari, Bishop, & Hung, 2007; Carminati, Ferrari, & Hung, 2006) propose OWL (Web Ontology Language) vocabularies with SAML assertions (included in WSDL documents) specifying services’ security capabilities combined in Boolean formulas. Most security ontologies address security concepts at a very high level of abstraction (Maleshkova et al., 2010; Maamar et al., 2006) requiring machine-clients to encapsulate knowledge regarding the implementation of each security strategy (e.g., encoding schemes, cryptographic techniques, server URIs, etc.). This approach hides information (assumptions) that may be necessary to determine whether an API can be used or not (i.e., whether the service is suitable for a composition). Others consider specific security domains in detail (Blanco et al., 2008) or provide a vocabulary of unrelated concepts, where responsibilities of future components are unclear (Carminati et al., 2006).

The tension between abstract descriptions and concrete and executable implementations is common. When hiding technical details, service designers can document, reason, maintain, and modify security policies at a business and global level more easily. Although it is tempting, technical detail is required in practice at various levels. For instance, using X.509 vs. SHA1 for providing integrity affects the security (e.g., it is questionable to use X.509 in such a way), business, performance, and usability levels. Furthermore, fine-grained actions, such as string manipulation, and the implementation of the security mechanisms themselves, are necessary in order to effectively enforce security policies. However, the more detailed the description, the more it becomes cluttered and complex, and also the stronger is the coupling between the consumer (client) and the provider. That is, low-level descriptions require clients to know in advance the semantics and assumptions of the description. The client specializes in interacting with such kinds of services, but as technology and business goals change on the server-side, changes to the description will cause all clients to fail.
We aim at a middle-way approach where low-level constructs are limited to state handling for HTTP and actual implementations of security mechanisms are referred to through URIs. When services change their description, clients can identify at which point and why they failed but components remain loosely coupled. A fully automatic recovery may be only possible if alternative implementations are available (e.g., through URLs) and can be dynamically bound but such strategy is out of the scope of our current work. In order to lessen the coupling between clients and servers, security mechanisms and concepts are defined in a vocabulary that extends the work proposed by Garcia et al. (D. Z. G. Garcia & de Toledo, 2008; D. Garcia & de Toledo, 2008). They associate security concepts at a high level of abstraction with security mechanisms and later use their ontology to enrich WS-Policy descriptions used by WSDL-based service. We extend their proposal with additional security concerns and include protocols as a security mechanism.

For the case of REST, research initiatives on service composition are fairly recent, and interest on QoS properties have focused mainly on security. For instance, in (Kübert et al., 2011) a RESTful service API is defined to support service-level agreements based on the WS-Agreement standard. Agreements (and templates) are REST resources encoded in the application/xml media-type, whose life cycle and state is handled by means of HTTP verbs. Graf et al. present a server-side authorization framework based on rules that limit access to the served resources according to HTTP operations (Graf et al., 2011). In (Field, Graham, & Maguire, 2011b) a server-side obligation framework allows designers to extended existent policies with rules regulating users’ information access. Rules may trigger additional transactions (e.g., sending a confirmation e-mail, register the information access attempt in a log) or even modify the response content or the communication protocol (e.g., require HTTPS).

Allam (Allam, 2012) aims to homologate WSDL-based and RESTful services by considering them black boxes, where interaction occurs as simple message passing between clients and servers. Security policies can be placed when receiving or sending a message as well as locally (e.g., at the server or client side). This vision does not consider
complex interaction involving third parties (e.g., OAuth), or service’s interface heterogeneity (Hongbin et al., 2012), where industry standards are implemented in various ways. We assume a workflow perspective where data are transformed locally in successive steps until the message constraints required by the service provider are achieved. In addition, clients, servers, and third party services may engage in an interaction that implements sub-workflows.

These initiatives contribute to bringing the traditional research and techniques of WSDL-based services closer to the REST perspective. However, they focus on hiding on the server-side the logic related with QoS properties, in a way that clients cannot make informed decisions regarding how to proceed, which contravenes the REST principle of visibility. By making relevant information visible to Web components (e.g., such as proxies or clients) through metadata, status codes, etc., it is possible for them to adjust their behavior and achieve some desired properties (e.g., scalability). It is possible also to recover from a failed interaction (e.g., following a retry link in case of a failed authentication). For the case of service composition this introduces constraints that are not explicitly declared, resulting in unstable composed resources. This principle, known as serendipity (Vinoski, 2008), allows unanticipated interoperability without requiring rigid and complex service interfaces (or contracts), but depends on human intelligence to repair a failed interaction.

3.5. Security aware RESTful Service Composition: ReLL-S

Based on the work of (D. Z. G. Garcia & de Toledo, 2008) and the WS-Security standard, we provide an ontology comprising around 30 main concepts, classified into three core concepts: security goals, security tokens, and protocols (Figure 3.1). The ontology is later used to define a security constraints description called ReLL-S.
3.5.1. Security Ontology

As discussed before, we identify three security goals: confidentiality, integrity, and authentication (identity). We add authorization, understood as the mechanism that assigns to users certain specific access rights to resources. Encryption mechanisms provide cryptographic transformations that make messages unreadable by a third party (enabling confidentiality).

Figure 3.1. Security ontology: Oval shapes represent concepts while rectangles represent instances (blue arrows). Green arrows represent specialization relationships and black arrows correspond to relationships with specific domain semantics.
Integrity is usually enabled by Encryption mechanisms that produce a digital Signature from the message content so that the recipient can verify that the message has not been modified. The identity of a service Consumer, which could be a User or an Application, is proved through Authentication mechanisms such as Authentication Protocols. The Consumer access rights over functionality and content is verified through Authorization Protocols.

In order to present the appropriate keys to these protocols, Consumers play the role of Key bearers of Security Tokens. The token type depends on the Consumer purpose and the token format. The most common use is to represent Credentials for authentication, such as an Identifier and a Password. In other cases, tokens are used to grant access rights during an authorization process (Grant Token). Binary tokens are specific and complex formats that are used in particular security strategies.

Tokens are interpreted and exchanged in the context of security Protocols. For instance, in a SOAP service context, Kerberos and X.509 are examples of common authentication protocols. OAuth, on the other hand, is a popular authorization protocol widely used in REST implementations. Unlike former protocols, OAuth requires the interaction of three parties: Providers, resource Consumers, and Users. A User that owns a resource, hosted by a service provider (ResourceServer), grants access rights to a third party Application (a Consumer proxy). In order to do so, the Application must present the proper credentials, such as a ClientId (API key (Farrell, 2009)), to the ResourceServer and engage in an authorization protocol that produces the proper grant tokens (AuthToken). Other protocols may behave differently, but the basic concepts remain - a resource server must serve private resources to authenticated consumers that have the proper credentials and access rights.

The security ontology serves as a basis to determine fundamental concepts in the realm of security but is insufficient to fully support a secured interaction between REST services. It is necessary to have a similar high-level understanding of the service interface in order to allow a machine-client to interact in an automatic fashion. The lack of relevant
information (i.e., configuration, initialization, assumptions, asynchronous operations, legal invocation, state or operation mode, or side effects) in Web APIs for REST services increases the complexity of the client application (Taylor et al., 2009). Furthermore, for the case of REST, interfaces are documented in an ad-hoc way, generally in natural language, so that engineers derive the consumption rules and hardcode them in the clients. In order to facilitate to machine-clients the consumption of RESTful services, in (Alarcon & Wilde, 2010), ReLL (Resource Linking Language) is proposed. In the remainder of this section, ReLL and ReLL-S are introduced.

3.5.2. ReLL: Resource Linking Language

ReLL (Alarcon & Wilde, 2010) is a lightweight description for RESTful services that provides a declarative meta-model for RESTful services, including mechanisms for URI generation, extraction, parsing, and dynamic late binding. A machine client using ReLL can traverse the graph of resources that underlies a RESTful service or change the state of some of the resources. ReLL offers a simple way to describe RESTful services considering resource identification, linking, and a uniform interface through which linked resources can be accessed (see upper box in Figure 3.2). ReLL design is based on the hypermedia constraint, which means that service interactions that in non-REST approaches result in server state are actually implemented as clients following links to resources representing that state, resulting in services that are resource- and link-centric.

The ReLL metamodel (Figure 3.2, upper box) is the basis of an XML Schema (Alarcon & Wilde, 2010) used to write ReLL XML descriptions. A resource may have multiple representations, which are the serialization of the resource in some syntax or media type. Representations can be associated to schemas for possible input data validation. A representation can contain any number of links, which can be retrieved through selectors expressed in a language (selector type) that suits the representation media type (e.g., XPath expressions for XML documents). A selector refers to a location, which can be the representation content or its metadata (e.g., HTTP headers) as the expression scope. Links relate a resource’s representation to a resource type (instead of a resource URI) as indicated by
the target in order to avoid coupling with the resources’ naming scheme (since the actual resource URI can be discovered only at run-time).

A link has a link type representing the link semantics, but its semantics are outside of the scope of the description language. A link can also contain a protocol description, which specifies the rules that govern the interaction with the linked resource. This is important because links in RESTful services not only have application-specific semantics, but following the links also may require different ways of using the uniform interface provided by a certain protocol. It is possible to compute links by executing an expression such as a concatenation, written in some language.
3.5.3. Security Constraints Model: ReLL-S

ReLL descriptions focus on the services’ underlying hypermedia as well as the rules to traverse it. QoS is not part of such responsibility, and in order to foster modularity, it must be kept as a separate extension component where relationships between basic descriptions and QoS constraints remain as less intrusive as possible. In addition, the risk of over engineering the description (and hence tighten up the contract between servers and clients) must be also considered.

The lower box in Figure 3.2 (ReLL-S) depicts our proposal. A Constraint-Set comprehends a set of Scopes that refer to a single or many Resources or Constraints. Constraints can be reused by different scopes, and related constraint-sets can define a rich, specific interaction Protocol that may require Encryption or Signature mechanisms, or Security Tokens. Protocols must provide enough detail for machine-clients to resolve the interaction but must not include specific implementation detail in order to foster flexibility and reusability. Each element in the extension module must be identified. Protocols may require invoking specific resources concerning the security domain under certain preconditions in order to encapsulate security responsibilities within the security module.

Based on the security goals described in Section 3.5.1, a security constraint can be one of confidentiality, integrity, authentication, or authorization, briefly detailed here.

- **Confidentiality**: This is the simplest security constraint and refers to the mechanisms that enable confidential communication, such as encryption (e.g., MD5, SHA1, SHA256). This constraint implicitly refers to the mechanism (standard or ad-hoc) to be used, but does not explicitly bind the description with a message encryption implementation; it is the client responsibility to provide such binding.
- **Integrity**: Similarly to confidentiality, it is used to provide message integrity through digital signature mechanisms. Many services implement their own signature mechanism that is applied to a string in order to generate a signature. As with confidentiality, the specific steps about how to build the signed string (e.g.,
sort the data, apply a digest algorithm, and concatenate the resulting key to the original content) are not specified by the constraint, but an implicit reference to the implementation must be provided. The constraint must also provide the mechanisms to refer to information across a constraint set (i.e., variables and request pattern) that may be required during a signature process.

- **Authentication**: This refers to the protocols used to prove client’s identity. These include HTTP Basic and Digest authentication (Franks et al., 1999), OpenID (Recordon & Reed, 2006), username and password, application token, etc. All of them rely on tokens representing the client identity. The constraint must describe the tokens needed during the protocol, as well as the mechanisms required to identify and provide required information (i.e., variables and request pattern).

- **Authorization**: This refers to the mechanisms used to grant access rights over restricted access resources, such as the OAuth protocol. It defines the mechanism name and the authorization grant (token) as well as the information to be retrieved and passed along during the interaction (i.e., variables and request pattern). If authorization algorithms require a sequence of steps to be followed, these steps must be represented as intermediary resources in order to cope with the stateless REST architectural constraint. Hence, the protocol must dynamically bind such resources by using hypermedia constraint (i.e., the URLs must be provided by the servers in the representations). Current implementations of the OAuth protocol sadly ignore this architectural constraint and force clients to hardcode relevant URIs, hampering server-side evolution and fostering brittle clients.

These constraints are defined in an XML schema as simply as possible, adding just the minimal elements to be readable by a client machine but avoiding the complexity of WS-Security. When possible, we use the same nomenclature in order to clarify the element intended use (e.g., security tokens, encryption, and digital signature mechanisms). Element identifiers are used to match constraints in the model. The following section provides examples in which we describe wide-usage security strategies.
3.6. ReLL-S security description

Compared to traditional services, security is addressed in a different way on the Web. In Maleshkova et al. (Maleshkova et al., 2010) a review of the self-declared REST Web APIs corresponding to the ProgrammableWeb site \(^1\) was analyzed regarding their support of security mechanisms. The Maleshkova et al. study analyzes a Web site that is a well-known repository for services (including WSDL, REST, XML-based, Web APIs, etc.). They picked 222 services from the RESTful service category (18%) covering the 51 service categories (e.g., search, geolocalization, etc.) available. The security mechanisms found range from very simple practices (the majority), to the W3C standards on security on the Web. For instance, 38% uses API Keys whilst the OAuth protocol is used by 6% of the reported service. Notice that mainstream so-called REST services (e.g., Facebook, Twitter, Google, etc.) support and require OAuth authentication. Therefore, we believe the study is representative from a practical (and informal) and a theoretical (standards) point of view. In this section we present ReLL-S descriptions supporting each of the security mechanisms identified in the Maleshkova et al. survey.

3.6.1. API Keys

This is an instance of a simple Authentication protocol where the consumer credentials for retrieving a restricted resource are directly passed to the provider within the request. In this case, the credential is the API key, an instance of a clientId. Given the key doesn’t need to be encrypted or signed, neither confidentiality nor integrity constraints are involved. The key value is passed as a query parameter (named client_id in the example) and its value must be provided to the client at run-time through a local variable (e.g., $client_id).

Listing 3.1. A simple authentication constraint specification (Api Key)

```xml
<constraint id="apiKeyAuth" type="Authentication">
```

\(^1\)http://www.programmableweb.com
3.6.2. Username and Password

This is another instance of a simple Authentication protocol where the credentials are directly sent from consumer to provider, but in this case both Identifier and Password credentials are required. The username and the password don’t need to be encrypted or signed and the message content is specified similarly to the Listing 3.1.

**LISTING 3.2. A simple authentication constraint specification (Username/Password)**

```xml
<constraint id="userPwdAuth" type="Authentication">
  <usernamePassword>
    <query_param select="$username" name="username"/>
    <query_param select="$password" name="password"/>
  </usernamePassword>
</constraint>
```

3.6.3. HTTP Basic Authentication

This is a more complex protocol for Authentication. HTTP Basic Authentication (Franks et al., 1999) is a standard supported by most web browsers; it aims to provide a little more security than sending authentication information in plain text. This protocol requires sharing the consumer credentials in an opaque string sent in the HTTP request header, following a well-known encryption mechanism such as Base64 encoding and a simple signature algorithm. The signed string is a concatenation of username, colon, and
password. In this authentication protocol, the confidentiality goal is also supported; however, the confidentiality constraint has a different scope because it applies on a specific piece of information and not to the entire signed string.

**LISTING 3.3. An authentication and confidentiality constraint specification (HTTP Basic Auth)**

```xml
<scope resource="type1">
  <constraint id="httpBasicAuth" type="Authentication">
    <var name="encoded" type="credentials"/>
    <header_param name="Authorization" select="concat('Basic ', $encoded)"/>
  </constraint>
</scope>

<scope token="credentials">
  <constraint id="httpBasicCon" type="Confidentiality">
    <value-of select="BASIC_AUTH"/>
  </constraint>
</scope>

<encryption id="BASIC_AUTH" name="wl:base64encode">
  <description>Concat userId, a semicolon, and a password to generate the base string, which must be encrypted with base64encode.
  </description>
  <value-of select="concat($userid,':',$password)"/>
</encryption>
```
This flow starts with the authentication constraint that specifies how to build the required HTTP header request using a credentials variable. Notice that it affects (scope) a set of resources corresponding to a resource type as described in the ReLL description, at a domain level. The variable used to generate the HTTP header in the request message is affected by another scope corresponding to a security token that provides confidentiality using a BASIC_AUTH encryption mechanism. In order to foster modularity, the encryption mechanism is defined as a separate element.

The encryption module implements a Base64 encoding whose specification can be located at a specific URL; a description is also provided, and the mechanism concatenates the `userId` and the `password`. The values of such variables must be provided at runtime to the machine-client consuming the secured resource. The resulting encoded value is passed to the credentials scope and then to the `encoded` variable to form the HTTP request header.

### 3.6.4. HTTP Digest Authentication

HTTP Basic Auth is not a very secure protocol for user authentication because the encoded entity is transmitted as clear text that can be easily decoded. The HTTP Digest Web standard provides a stronger and more complex, support for confidentiality and integrity. Username and password credentials need to be signed with `realm` and `nonce` attributes (in the simplest case), provided by the secured server. The signature is usually generated with the MD5 encryption mechanism.

This protocol is implemented in two phases. In the first phase, the server replies with a 401 (Unauthorized) message when a client attempts to retrieve a resource that is protected with HTTP Digest, including sensible information in the HTTP header. This representation can be interpreted by a machine-client through a ReLL description as shown in Listing 3.4. A `WWW-Authenticate` response header includes the variables `realm` and `nonce`. If the client does not have this information, it will not be able to build
the authentication parameters correctly for the second request. Since this information is part of the client state, it is the RESTful client responsibility to save such information.

After storing the nonce and realm values locally, the client is able to build the request message for the second interaction. Such message includes a set of variables to be included
as headers. The format and value for each of these variables are specified using concatenation functions (XPath), where some values must be passed to the client at run-time. That is, the `userId`, the `nonce` and `realm` variables previously obtained, the URI of the protected resource (`$REQUESTED_URI`), and the HTTP method (`$METHOD`) to be used.

The user’s password must be provided within the scope of credentials supporting a confidentiality constraint. This constraint describes how to build the credentials value using encryption mechanisms defined within the constraints set document, which in this case are the checksum algorithm and the digest algorithm. Unlike the case of HTTP Basic Auth, confidentiality provision is more complex since it requires concatenating some values (stored in variables A1 and A2) that are later encrypted using the checksum algorithm. A concatenation of the resulting values must be encrypted again using the checksum algorithm.

**Listing 3.5. An authentication and confidentiality constraint specification (HTTP Digest Authentication)**

```xml
<scope resource="type1">
  <constraint id="digestAuth" type="Authentication">
    <var name="user" select="concat('username=', $userid)"/>
    <var name="_realm" select="concat('realm=', $realm)"/>
    <var name="_nonce" select="concat('nonce=', $nonce)"/>
    <var name="_uri" select="concat('uri=', $REQUESTED_URI)"/>
    <var name="request-digest" type="credentials"/>
    <var name="response" select="concat('response=', $request-digest)"/>
    <header_param name="Authorization" select="concat('Digest',
        concat((($user, $_realm, $_nonce, $_uri, $response),',
        '))" />
  </constraint>
</scope>
```
3.6.5. OpenID

This authentication protocol is significantly different from the others because it requires an interaction with a third party that validates the identity proof. This party is the OpenID provider; a consumer (user) can authenticate itself against a service by presenting a
token generated by an identity provider where the consumer has been previously registered and which is trusted by the service.

LISTING 3.6. An authentication constraint specification (OpenID)

```xml
<scope resource="restrictedResources">
  <constraint id="openIdAuth" type="Authentication">
    <appToken>
      <query_param name="userLoginToken"
        select="userLoginToken|wl:OpenID"/>
    </appToken>
  </constraint>
</scope>

<protocol name="wl:OpenID">
  <invoke url="http://www.identityprovider.com/open_id"
    pre="not($userLoginToken)">
    <query_param select="$callback" name="callback"/>
  </invoke>
</protocol>
```

The interaction with the third party is defined as a protocol element specifying the identity provider location that will be invoked if a precondition is satisfied. The precondition evaluates the client state asking whether certain information is not available, in this case, and if satisfied, a message to the identity provider will be issued including probably some query parameters. Since OpenID is a third party protocol, interaction between the user and the identity provider occurs out of the interaction band between the user and the machine-client to guarantee the privacy of user sensitive information. In order to regain control of the conversation, the machine-client must provide a way to receive the validation information from the third party. This feature is accomplished in this case through a
callback connector whose address could be accorded between service and client at setup
time or passed as a parameter to the provider under the callback query_param element,
as shown in the example. The interaction implies that the client will not have the required
token until it is received through the callback, which is not described by the constraints set,
and it can be implemented in any way the client prefers. Once the client has the token, it
can be passed as a query_param to satisfy the authentication constraint.

3.6.6. OAuth

OAuth (Hammer-Lahav, 2010) is an Open Authorization protocol that allows a third-
party application - a client application - to access resources provided by a service - resource
server - and owned by a user. The user has to authorize the third-party application to ac-
cess the resources stored by the resource server, without exposing his or her authentication
credentials, to the third-party application. The authorization grant is represented as a token.

OAuth defines four grant types: authorization code, implicit, resource owner password
credentials, and client credentials; and provides an extension mechanism for defining ad-
ditional grant types. The protocol flow is flexible and depends on the type of authorization
that is going to be granted. So in the flow shown in Figure 3.3, up to four parties could be
involved: the resource owner, the resource server, the authorization server, and the client.
The result of the protocol is an access token that represents the authorization granted by
the resource owner and that is sent later by the client to the resource server to access the
restricted resources.

What is interesting about this protocol is that it involves more than two entities that
can communicate using a protocol that has been designed for client-server communication,
as it is HTTP, in a stateless one-to-one conversation. However, this characteristic presents
particular issues to deal with, which makes it really interesting to describe in more detail.

Once the client has required the access grant, the authorization server starts a conver-
sation with the resource owner that, from the client’s perspective, occurs completely out
of its control. This conversation’s goal is asking the user to give authorization to access
the resources. It could be implemented many ways; the authorization server could send an email to the user, an SMS, or any other kind of conversation. In most implementations the conversation occurs synchronously by redirecting the user-agent from the client domain to the authorization server domain. So that, to restore the interaction between user and client, the authorization server must also redirect the user-agent to the client. If the user grants access rights to the client, such redirection message will contain an authorization grant. This conversation implements an asynchronous conversation through a callback connector provided by the client, which becomes the target of the redirection.

Once the client has the authorization grant, it can use it to retrieve an access token from the server. The client sends its credentials and the authorization grant, representing the permissions granted by the user, to the server. The server verifies the permission and generates a final token (access token, or oath token) to be used in future calls, so that the
resource server allows access to restricted resources. It could have an expiration time and some authorization servers provide the feature to refresh or renew the token.

**Listing 3.7. An authorization constraint specification (OAuth)**

```xml
<scope resource="restrictedResources">
  <constraint id="oauthAuth" type="Authorization">
    <accessToken>
      <query_param name="auth_token" select="$auth_token|wl:OAuth"/>
    </accessToken>
  </constraint>
</scope>

<protocol name="wl:OAuth">
  <invoke url="http://www.authorizationserver.com/oauth" pre="not($auth_code)">
    <query_param select="$state" name="extra"/>
    <query_param select="$callback" name="callback"/>
  </invoke>
  <invoke url="http://api.service.com/getToken" pre="$auth_code">
    <query_param select="$auth_code" name="auth_code"/>
    <store selector="token" persist="auth_token"/>
  </invoke>
</protocol>
```

In this case the client makes two calls to get authorization to the user’s restricted resources. The first one could include the callback address and also a parameter, named `state`, that must be sent back by the authorization server when issuing the callback request to the client. The client can use the parameter to keep the flow state. The callback
call will include also the authorization code used by the client to get the final authorization token during a second call. The order of both calls is defined by their preconditions (i.e. they play the roles of guard conditions). That is, when the client runs the protocol, it will invoke the first call initially because it doesn’t have the auth_code stored locally. Once the request is issued, the client blocks itself waiting for an answer. When the callback arrives, the client is restarted and since its current condition has changed (i.e., it has the code) then it will make the second call (the condition is met) to finally resolve the authorization constraint.

3.7. Implementation and Case of Study: RESTler and Flickr

RESTler (Alarcon & Wilde, 2010) is a machine-client that can consume services and follow the links embedded in the served representations (hypermedia). Originally, RESTler was conceived as a proof of concept for ReLL. That is, if a RESTful description was hypermedia-centric, then a crawler may use it to make sense of services interfaces and navigate the associated graph of resources. Later, RESTler was extended to follow a GRDDL approach\(^2\) and transform the retrieved resources into semantic entities (instances). This was possible because ReLL descriptions provide a named high-level vision of a Web graph; in that sense, it became a domain vocabulary for a service (Alarcon & Wilde, 2010).

In a third experiment, RESTler was provided with instructions to follow a business process where simple control-flow (sequential invocation) and data processing (merging) was dictated by a Petri-Net (encoded in a PNML file) (Bellido et al., 2011). Along these experiences RESTler has evolved from an in-depth, recursive crawler to a constrained navigator, similar to a headless browser (we are not yet addressing code-on-demand such as Javascript). In this paper, we are experimenting with enriching the description of resources RESTler is attempting to consume. RESTler is a robot that can react and adapt to changes in its environment if the server provides it with meaningful information: links and guarded conditions to model interaction steps. That is, in addition to implementing security tasks

\(^2\)http://www.w3.org/TR/grddl-primer/
(e.g., encryption, string manipulation, etc.), RESTler can execute workflows and sub workflows. Further work is needed to model services in terms of states, for instance, so that machine-clients like RESTler could be provided with proactive behavior. But such a goal is out of the scope of this paper, although some research is being developed in those areas (Ivan Zuzak, 2011; Ghezzi & Gall, 2013; Verborgh, Mannens, & Van de Walle, 2013).

### 3.7.1. ReLL-S extensions for RESTler

RELL-S does not depend on RESTler. It only requires a simple XML parser in order to read descriptions and a constraint resolver that enforces the security policy. Such a resolver needs not only to understand ReLL-S descriptions; it also must include the necessary modules to execute security mechanisms (e.g., implement an md5 algorithm). RESTler was implemented in Java and was extended with various modules (Figure 3.4). A constraints (RELL-S) reader and parser have been added to the ReLL’s reader. A constraints resolver component has been added along with improvements to the HTTP client. Since some security protocols require asynchronous interactions, a callback connector and processor has been added as a generic endpoint to receive HTTP requests. The RESTler constraint resolver is extended via plugins to support actual security mechanisms. Notice that some mechanisms are ad-hoc (e.g., FLICKR_MD5 in Listing 8), while others are well-known mechanisms as those registered in the ontology (e.g., wl:MD5 in Listing 8). Our security ontology is used as a vocabulary (D. Garcia & de Toledo, 2008) where fundamental concepts are defined and used in ReLL-S descriptions (i.e., as elements, as attributes) and later executed by the Constraint resolver, following a model-driven approach.

A detail of RESTler logic when interacting with secured resources can be seen in Figure 3.5. When the RESTler engine attempts to consume a guarded RESTful service (sd fetch protected resource sequence fragment) without the proper credentials (A), the origin server responds with an appropriate message (e.g., (B) 401 Unauthorized). If such a situation occurs, the server following a hypermedia approach must issue a message including all the information required for the user-agent to reach its goal. That is, a Link Header including the address of the security constraints set must be included in
the response. Alternatively RESTler could discover such additional information by issuing an HTTP OPTIONS message indicating the resource it attempts to consume (C). In either case, the server must include in the response the address of the security constraints and possibly the address of the service description (i.e., (D) ReLL and ReLL-S documents).

Once it has received the documents’ URLs, RESTler analyzes whether it keeps a fresh copy in its cache, and if not, it fetches the documents (E, F). Both ReLL and ReLL-S descriptions are parsed by a reader and an object model is obtained (G to J). At this point, based solely on the obtained models, it is possible to evaluate whether RESTler can satisfy the security constraints required to consume a service. For instance, it can determine whether it has the proper credentials, such as API key, username, userId, and password cookies, for the case of simple Authentication constraints. It also must determine if it has the proper capabilities or libraries for the case of Confidentiality constraints (e.g., the base64 encryption algorithm complying with RFC2617 as required by HTTP Basic Auth, or MD5 for HTTP Digest). When ad-hoc algorithms are required (see Listing

![Diagram of RESTler architectural components](image_url)
3.8, line 38), the corresponding implementation must be plugged into RESTler as a new security mechanism (see Figure 3.4) so that the capability is supported.

These constraints are simple in the sense that they require a simple evaluation to determine whether they can be fulfilled; other more complex constraints can only be determined
dynamically. For instance, OpenID and OAuth require interaction with one or more servers following a protocol in order to obtain some credentials. RESTler can only determine in advance whether it has fresh copies of the required credentials (cookies), whether it has the required capabilities (e.g., Encryption libraries), whether it has the capability to issue HTTP messages, whether it is willing to enable callback URLs if necessary, and the number of steps involved in the protocol. It cannot determine in advance additional QoS, such as the total response time (e.g., the servers could fail to provide an answer) and hence a timeout policy could be required, the availability of the servers, etc. Furthermore, along the path of the protocol a resource may be protected and an additional protocol may be launched. That is, nested protocols may take place and can only be discovered dynamically. A possible solution consists in exploiting HTTP OPTIONS messages to fetch all the ReLL and ReLL-S documents related to all the resources involved in order to determine the scope of the interaction as well as the basic requirements (cookies, cached copies, and libraries) so that RESTler can better determine whether it is feasible to authenticate and later fetch the protected resource. Again, dynamic QoS attributes (e.g., response time, availability, abrupt changes on the interface, etc.) cannot be determined in advance.

Once RESTler determines whether it is possible to engage in a secure interaction, the constraints are resolved and enforced (ref fragment in Figure 3.5). As a result, state information must be produced (e.g., credentials) so that it can be used by RESTler to retry its initial goal (sd fetch protected resource). In the following subsection we detail the protocol corresponding to such ref fragment corresponding to a specific implementation of OAuth for Flickr.

### 3.7.2. OAuth for Flickr

The OAuth protocol allows room for customization, so that implementations can vary from one service to another. For instance, there are four types of authorization grants - there are different ways to require the user authorization and authentication; the authorization server could be independent from the resource server or they could be integrated, etc.
Listing 3.8 corresponds to a ReLL-S description for the Flickr API. The specification scope includes three protected resources (lines 2, 19, 31) corresponding to the resources RESTler attempts to retrieve (userComments), the authorization server, and the token grant service. Based on the specification RESTler evaluates to interact with the protected resources based on its capability for supporting the ad-hoc signature mechanism (line 58) and whether it holds valid copies of the security tokens that cannot be obtained otherwise (lines 5 and 6, 22 and 23, 34 and 35). Our approach does not bind the constraint set description with specific implementations but requires description designers to provide pointers to such elements. For instance, the machine client uses Flickr signature specification, referred to by a URL (location sub element of DigitalSignature element), to determine whether the mechanism is supported.

The access token specified in line 14 can be obtained by following the protocol specified in lines 44 to 56. No alternatives are provided to obtain the appToken, since it requires human intervention (i.e., to manually register RESTler as a valid third-party application). Additionally, when manually registering RESTler with Flickr a callback URL is stored and later used by RESTler. The protocol consists of two steps where the first invocation follows the send-and-forget pattern (i.e., indicated by the nofollow mark, line 49) and the second one is triggered when the conditions are met. At this point, RESTler criteria are driven by user preferences (e.g., timeout, retries in case of a failed call, etc.)

**Listing 3.8. An authorization constraint specification (OAuth) for Flickr**

```xml
<constraintSet xmlns:wl="http://www.example.com/wl-security/">
  <scope resource="flickr:rest?method=flickr.activity.userComments">
    <constraint id="appAuth" type="Authentication">
      <appToken>
        <query_param select="$api_key" name="api_key"/>
        <query_param select="$api_secret" name="api_sig"/>
      </appToken>
    </constraint>
  </scope>
</constraintSet>
```
<scope resource="flickr:auth/">
  <constraint id="appAuth" type="Authentication">
    <appToken>
      <query_param select="$api_key" name="api_key"/>
      <query_param select="$api_secret" name="api_secret"/>
    </appToken>
  </constraint>
</scope>

<scope resource="flickr:rest/?method=flickr.auth.getToken">
  <constraint id="appAuth" type="Authentication">
    <appToken>
      <query_param select="$api_key" name="api_key"/>
    </appToken>
  </constraint>
</scope>
<query_param select="$api_secret" name="api_sig"/>
</appToken>
</constraint>

<constraint id="FlickrMD5Sign" type="Integrity">
    <mechanism name="FLICKR_MD5"/>
</constraint>
</constraint>
</constraint>
</scope>

<protocol id="wl:OAuth">
    <invoke name="Step1" pre="not($auth_token) and not ($frob)">
        <url="http://www.flickr.com/services/auth/"/>
        <query_param select="$perms" name="perms"/>
        <query_param select="$extra" name="extra"/>
        <nofollow/>
    </invoke>
    <invoke name="Step2" pre="$frob and not$auth_token">
        <url="http://api.flickr.com/services/rest/?method=flickr.auth.getToken"/>
        <query_param select="$frob" name="frob"/>
        <store selector="token" persist="auth_token"/>
    </invoke>
</protocol>

<digitalSignature name="FLICKR_MD5">
    <location url="http://www.flickr.com/services/api/auth.spec.html"/>
    <description>Sort parameters value alphabetically based on parameter name and concat this to the secret string
Figure 3.6 indicates the steps followed by RESTler in order to enforce the specification once the evaluation was positive. It starts by resolving the authentication constraint; that is, RESTler retrieves the appToken from a local storage (lines 5 and 6 in listing 3.8, message A, B in Figure 3.6). Then the second constraint is resolved; that is, the FLICKR_MD5 library is loaded (message C). When the third constraint is resolved and no local copy of the accessToken is found, the wl:OAuth protocol (OR operator) starts. The first step of the protocol includes an interaction with an authentication server under similar conditions. A pre-condition is specified in order to trigger the invocation; that is, neither the local variable (i.e., stored as a cookie or in a local storage) auth_token nor the frob must be held by RESTler. If the condition is satisfied, a request message to the resource URL including the appAuth parameters and the perms and extra parameters signed by the FLICKR_MD5 mechanism is issued (message D) and RESTler does not wait for a response (nofollow).

This invocation is issued to obtain a transitory token (frob) issued when the user grants permission to access the protected resources (userComments), and the second one is issued to obtain the final accessToken. RESTler attempts to perform each invocation in sequence; however, each one is guarded by a condition (i.e., the temporal permission must not exists in the former case, and it exists but the final token does not, in the latter case). That is, only the first invocation will take place if RESTler was not previously authorized to access the resource. Once the first invocation is issued, Flickr redirects the user to another authentication page requesting the end user (not RESTler) to grant authorization. At this point RESTler losses control of the interaction (it occurs between the end user and Flickr).

When the grant is issued, Flickr redirects the end user to the RESTler callback (message E). At this point, RESTler receives and stores the representations (i.e., the frob and
extra variables, messages E, F and G) and attempts to fulfill all the security constraints. That is, again authentication and integrity constraints are resolved (messages H, I, J) and since the auth_token is not yet held by RESTler, the protocol is fired. This time, preconditions have changed (the frob is available) and the second step starts. Again the invoked resource is protected by integrity and authentication constraints, so the request message is built by applying the FLICKR_MD5 to the frob, and appToken (api_key and api_secret) parameters (message K). The response message shall contain the access token, which is stored as part of the client state (e.g., as a cookie, message L, M). At
this point RESTler has resolved all the security constraints and the control is passed back to Engine that is to retrieve the protected resource as indicated by fragment sd fetch protected resource in Figure 3.5.

3.7.3. Analysis of our approach in light of the REST principles

In (Bellido et al., 2011) we analyze the impact of ReLL on one critical factor, namely, coupling. Coupling is described through 10 facets or dimensions (Pautasso & Wilde, 2009) and the approach does not contravene any of them. We briefly summarize the dimensions stressing the desirable characteristic of each facet.

ReLL and ReLL-S support the discovery of resources instead of a centralized URL repository; resources identification (URIs) is agnostic from the protocol (i.e., not only HTTP); resources binding is dynamic. Platform independency is achieved by using ReLL since it is written in XML and XPath, which have ample support. Platform independency, however, is compromised by ReLL-S since we introduce expressions to deal with strings and (HTTP) cookies, even though they follow the XPath style. Synchronous (through callbacks) and asynchronous interaction is supported; vertical interface through protocols instead of layered dependencies (ReLL is independent from ReLL-S). Messages are independent from any model; ReLL and ReLL-S add an interpretation layer for the machine-client to perform its task, but in case of failure in the models, the composed service instead of component services will be affected. Stateless interaction favors massive scalability; the ReLL-S mechanism for dealing with asynchronous messages through callbacks and guarded conditions makes possible implementing workflows in a stateless fashion. RESTler is a stateless engine, so that state persistence from one interaction to the next must be implemented through cookies and parameters.

The capability of generating code from a service description (either a client, or a resource) is seen as a negative feature. This is because it increases the dependence, and hence the coupling, of the description to interact with (or generate the resources of) the service. Since ReLL and ReLL-S do not provide guarantees about the completeness and correctness of the service model, it is not possible to use the descriptions to generate code. RESTler
implements a reflective inspection mechanism based on clients inquiring about the possible future steps of the interaction from inspecting the servers' responses (hypermedia).

3.8. Conclusions

QoS properties, particularly security, are playing a major role in the current Web due to the massive scale, performance, availability and evolvability requirements that pervade modern Web applications. In the absence of a security description, a machine client is simply not able to retrieve restricted resources. Composition of Web services in the industry domain is hardly feasible if we just consider functional properties. Even though servers may steer clients in their interaction, it may be the case of a machine-client traversing unexpected (from a server perspective) paths, and executing unexpected controls in order to implement a new business process (choreography). It requires clients not only to understand how to use the service’s interface (as supported by ReLL) or its semantics (as proposed by the Linked Data initiative), but also to be aware and support QoS constraints for effective service composition.

In addition, current practices for services security (i.e., Web APIs) consider the involvement of humans in the process of granting certain security tokens (e.g., application keys, secret codes, etc.), through mechanisms such as Captcha (von Ahn, Blum, Hopper, & Langford, 2003). Automatic composition of services requires either to consider such human involvement (e.g., composed services pre-register to a set of interesting services) or to design alternative methods in order to guarantee the good faith of the service requiring a security token.

Unlike WSDL/SOAP-based services composition, REST architectural constraints require loosely coupled components in order to guarantee their independent evolvability. Service descriptions including functional and QoS properties should not tighten such coupling. WSDL descriptions, including or not QoS requirements, are a strong contract that tightly binds clients and servers. ReLL provides the means for a client to interpret a RESTful service interface. However, many interpretations may hold and ReLL does not guarantee
the consistency or completeness of its description with the corresponding service. It may inform, however, where and why such description does not hold anymore.

Considering this scenario and the dynamic late binding requirement, a machine-client should be able to determine dynamically or statically whether it can interact with a secured service (i.e., a Boolean evaluation of its supports of required encryption mechanisms) but it also must discover on run-time whether it can be engaged in complex interactions and satisfy a constraint. The latter situation also has an impact on the availability and performance of the composed service. More latency can be induced by an unexpected interaction; unexpected complexity and unforeseen security risks may take place since interaction occurs with unexpected services (e.g., due to redirections). Security models cannot be reduced to Boolean values or integers as machine-clients may need to determine to what extent they are willing to distract its goal in order to satisfy a constraint on run-time. Under our approach, the machine-client can determine beforehand the number of services to visit, the number of interactions (steps, or calls) to expect, and the complexity of the required algorithms. The ReLL-S metamodel can serve to determine a multidimensional metric to assess the suitability of a service to become part of a composition.

RESTful services are usually described in ad-hoc HTML documents specifying the service’s API in natural language. ReLL-S sits in between such flexible service documentation and a highly structured service description such as WS-Security, which is a strong contract between clients and servers. Since the constraint sets documents are discovered on run-time (through OPTIONS), servers must reflect the current state of services in the security constraint document. The description must leave to the client the responsibility of implementing specific mechanisms for encryption, canonicalization and other security tasks that can be identified but not described to the detail by ReLL-S. RESTful service interaction greatly benefits from hypermedia since it allows servers to change their interfaces dynamically. Since the proper URLs are embedded in the response messages, clients may recover dynamically while allowing components to remain decoupled. A fundamental task then is for RESTful services’ automatic composition to focus on generating the
proper messages, which requires a series of state transformation on the client-side as well as lightweight, loosely coupled, machine-readable service interfaces.
4. SAW-Q: A DYNAMIC COMPOSITION APPROACH OF REST SERVICES BASED ON QUEUE MODEL

4.1. Abstract

Service composition is one of the principles of service-oriented architecture; it enables reuse and allows developers to combine existing services in order to create new services that in turn can be part of another composition. Dynamic composition requires that service components are chosen from a set of services with equal or similar functionality at runtime, and possibly automatically. The adoption of the REST services in the industry has led to a growing number of services of this type, many with similar functionality. The existing dynamic composition techniques are method-oriented whereas REST is resource-oriented, and consider only traditional (WSDL/SOAP) services. We propose SAW-Q, an extension of Simple Additive Weighting (SAW), as a novel dynamic composition technique that follows the principles of the REST style. Additionally, SAW-Q models quality attributes as a function of the actual service demand instead of the traditional constant values. Our model is much more accurate when compared to real implementation, positively improving the quality of dynamic service compositions.

4.2. Introduction

Some of the properties that determine the quality of a service are its performance, availability, and security (Menasce, 2002). Availability is the percentage of time that the service is operational; security properties include the existence of mechanisms for authentication, confidentiality, integrity of exchanged messages, and non-repudiation. Performance as observed by the client can be measured with the response time: how long it takes to answer service requests. Conversely, performance from the provider’s perspective can be seen as the rate at which a service can process requests from multiple clients. Other measures of quality of service can include the maximum capacity or function that describes how performance varies with request load.
The REST architectural style is defined by a set of constraints that facilitate the design and delivery of scalable, dependable and evolvable services, thanks to loose coupling, standard well-defined interfaces and safe encapsulation of legacy components (Fielding, 2000). Scalability in REST refers to the ability of the architecture to support a large number of components (both clients and services) and interactions thereof. The main REST architectural components include servers, gateways, proxy servers and user agents that are associated through connectors such as clients and server interfaces, caches, proxies, and tunnels (Zuzak, Budiselic, & Delac, 2011). The Web services that are built based on the REST architecture are also called RESTful (Richardson, Amundsen, & Ruby, 2013).

There are several approaches that consider the service quality attributes (QoS) as the foundation in finding the best service to meet certain expectations (Menasce, 2002) (Alrifai & Risse, 2009) (Ran, 2003) (Zheng, Ma, Lyu, & King, 2011). However, these approaches only take into account the quality that can be delivered at maximum capacity (a constant, optimistic bound). In reality, the quality perceived by clients will change as a function of the actual workload of the service. This distinction has significant impact on QoS modeling since in the first case, such capacity is considered infinite whereas in industrial systems that is not possible. Even considering REST caches as a replication technique in order to increase servers capacity, such limit exists and a service operating below or above such capacity limit behaves differently in terms of its QoS.

In this paper we analyze REST services scalability considering the quality of service attributes according to workload and service capacity. This analysis is used to refine a model that is then applied for service selection in dynamic REST service composition scenarios.

### 4.2.1. Dynamic Composition

Unlike static composition, in dynamic composition, service components are selected at runtime; and a composition plan based on functional and non-functional requirements is generated and executed also at runtime. Dynamic composition strategies are categorized in several ways (D’Mello et al., 2011): based on constraints and business rules (Orriëns, Yang,
& Papazoglou, 2003), based on planning techniques from artificial intelligence (Zhang, Chang, Feng, & yi Jiang, 2010), based on interaction and user customization (Agarwal & Jalote, 2010), based on contextual information (Medjahed & Atif, 2007), and based on service’s signature (input and output parameters) (Paganelli & Parlanti, 2013). From a different perspective (Alamri, Eid, & El Saddik, 2006), dynamic composition strategies can be grouped into those that use wrappers or adapters to create a new service interface at runtime; others use a language describing the composition; others are based on workflows and an abstract process model where each activity can be linked to a service. Other techniques use an ontological description of the services; and declarative techniques use the rules defined by the customer as restrictions on the composition to determine the services that will be part of the composition, these rules are used to generate composition constraints according to customer specifications.

We follow a hybrid technique considering a workflow (an abstract model of composite service) and composition rules (declarative approach) defined by the client to determine the best combination of service components to be invoked from the workflow.

A pure declarative composition technique requires too much effort from the composer to determine the way components should be invoked and their relationship with the business rules. On other hand, using only workflow driven composition leaves aside requirements such as non-functional, business-related, etc. defined by those who require the composition.

4.2.2. QoS-aware composition

The quality of a composed service (QoS) results from a combination of several qualities properties of its component services (Menasce, 2002). Dynamic composition of Web services requires choosing the best component services that satisfy the functional and non-functional composition requirements (Menasce, 2002). Non-functional requirements involve quality attributes such as response time, availability, security and performance (Menasce, 2002) among others. The definition and measurement of these attributes vary. For example, the response time can be measured as the average of the results obtained over the last 15 minutes, or a vector of averages taken over 15-minute intervals, measured
during the whole day. Most approaches consider quality as a single value associated with each attribute, however, the current service workload will affect quality attributes. We propose a technique for describing quality attributes of REST services as a function that depends on the expected service workload rather than a fixed value.

4.2.3. Contribution

A composed REST service is a special intermediary; unlike gateways and proxies, it not only directs requests to the origin servers, but also breaks them down into several invocations to diverse services (Pautasso, 2009b). Existing dynamic composition techniques focus on traditional Web services (WSDL/SOAP), which do not take into account the architecture of the Web and hence are not equivalent to REST services. To our knowledge there have been no attempts to define QoS-aware dynamic service composition of REST services that take workload into account and use predictive models of QoS as the principal focus. In this paper we present SAW-Q, a hybrid approach to REST service dynamic composition, namely a declarative workflow based composition. The service composition is derived from an abstract process definition, taking into account the composition rules established by a consumer. We focus our research on QoS attributes such as response time, availability, and throughput. We also present a queue-based service model that considers REST constraints.

This paper is organized as follows in Section 2 we review and analyze the current approaches related to dynamic composition and queue models. Next, in Section 3 we establish the equivalence between REST architectural style and queue models, to obtain functions to measure the quality attributes of atomic web services considering the client’s demand. In Section 4 we propose SAW-Q as an extension of SAW to dynamically compose web services. Finally, a theoretical and experimental evaluation is presented in Section 5 and the conclusions in Section 6.
4.3. Related Work

Major research challenges in discovering Web services include, provisioning of services across multiple or heterogeneous registries, differentiating between services that share similar functionalities, improving end-to-end Quality of Service. The capability of discovering services across accessible registries that conform certain processes is a challenge that many researches try to meet. Clients that consume services may differentiate services with similar functionality according to their non-functional or quality attributes (Ran, 2003). The quality criteria that are commonly used are, execution price, performance, reputation, successful execution rate and availability among others. In order to satisfy client expectations and to be selected as the best option over other candidates, services should scale according to client demand. For instance, Join-Idle-Queue (JIQ) (Lu et al., 2011) is an algorithm for distributed load balancing in large systems. JIQ performs the load-balancing task considering user demand in order to choose the appropriate server so that client’s requests are distributed homogeneously. To achieve this, each service is modeled as a queue and then the load balancer estimates and compares the cost to deliver a client request to each service.

QoS aware service discovery approaches vary, for instance, one approach extends the service repository and allows clients to search for services based on quality of services criteria as specified by the user in preferences vector, however, they do not consider client’s demand when recommending the services (Al-Masri & Mahmoud, 2007a). Similarly a collaborative filtering approach for predicting QoS values of Web services and making Web service recommendation is proposed. This technique takes advantages of the service users’ past usage experiences in order to rank services, however this limits user’s control when selecting the services (Zheng et al., 2011).

The main objective of dynamic composition based on quality attributes is to derive an execution plan that maximizes the quality attributes of the composite service. The solution to this problem has exponential execution costs (NP complex) that are addressed using integer programming algorithms, and heuristic techniques. One approach proposes...
a technique that optimizes the time of service component selection according to a defined abstract service; it combines local optimization and global planning based on a hybrid selection technique (Alrifai & Risse, 2009). An improvement to this technique includes a distributed algorithm to select the service candidate (Alrifai et al., 2012). However, in both cases request demand is not considered and quality attributes are modeled as scalar values.

Simple Additive Weighting (SAW) is a technique that uses multiple decision criteria to recommend candidate services for a composition and has good performance (Turskis & Zavadskas, 2011), it is being used in optimization problems of local and global planning in dynamic composition of Web services (Zeng et al., 2004). SAW consists of two phases; the first phase consists of scaling and normalizing the quality attributes of the candidate services. The next phase consists of assigning a score to each service with reference to a quality attribute-weighting factor defined by the consumer. Each service that is a candidate for a task in the local optimization is scored and the services with the highest scores are chosen, while for the overall planning the score is assigned to the combination of possible candidates for a composite service. The set of candidates with the highest score are selected for composition. AgFlow (Zeng et al., 2004) is an implementation of this approach and the reported experiments demonstrate that it is a highly efficient technique for the selection of component services in terms of computation cost. Likewise previous approaches, SAW do not consider client’s demand.

4.4. Towards a Queue Model for RESTful services

4.4.1. REST architectural constraints

Fielding (Fielding, 2000) proposes a set of architectural constraints that have diverse impact on non-functional requirements. In this paper we focus on Response Time, Availability and Throughput since we are interested in how the demand (client requests) impacts the service. More in detail, the REST hybrid architectural style is derived from others styles to combine useful architectural properties. That is, repository replication (Replication), a hierarchical decomposition of components (Hierarchical), and executable code on
TABLE 4.1. Architecture constraints that determine the scalability of RESTful services.

<table>
<thead>
<tr>
<th>Replication</th>
<th>Quality attributes</th>
<th>Response Time</th>
<th>Availability</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td></td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCS</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>CSS</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>CSSS</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>LCSCS</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Mobile Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VM</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>COD</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>LCODDCS$SS</td>
<td>-</td>
<td></td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

demand (Mobile Code). These characteristics are defined by the architectural constraints summarized in Table 1 and explained below. We are not considering the Uniform Interface constraint since the cases where it affects Response Time are particular (e.g. format transformations) or solved by the inclusion of caches, which are already included in our analysis under Replication.

Repository Replication (RR) reduces service response time by decreasing the latency of the network. Service availability is increased when the service is replicated because if an instance fails, subsequent requests will be answered by other instances. Service throughput is also improved by replicating the service since it increases the processing capacity for the service requests.

A specialization of the Repository Replication (RR) is the Cache ($) or temporary storage of the responses to RESTful services requestst. This type of temporary storage decreases the response time of a service when it has been invoked before and its representation is temporarily stored in any of the architecture components (i.e. Browser, origin server, proxies, etc.). Again, service availability and throughput are improved through the existence of buffered copies (if they exist).

The separation of concerns is the principle behind the Client-Server (CS) constraint. An adequate separation of the server component functionality simplifies the REST service deployment and hosting and hence improves service scalability. The service availability is
also improved because this constraint allows components (clients, servers and intermediaries) to evolve independently, since their interface does not change.

Layered Client Server (LCS) reduces the coupling between multiple layers, by hiding the server from clients behind intermediate layers. Although the layering improves the evolution and reusability of a service, the coordination and communication overhead between the layers increases the number and size of the messages interchanged and hence the response time of a service.

Client-Stateless-Server (CSS) derived from Client-Server (CS) with the additional restriction that each request to a server must contain all the necessary information to process the request, and not take advantage of context information (Stateless). To maintain the interaction state on the client-side increases the service response time for a request message, since messages must be larger and will take longer to transmit and process them. The service availability is improved because it is easier to recover a service from failures and reprocess the last request instead of retrieving context information that could be lost during the failure. Service performance is also improved because the server does not need to store information and therefore it can release the used resources, immediately after responding to each request.

Client-Cache-Stateless-Server (C$SS) derived from the Client-Server-Stateless (CSS) and Cache ($) restrictions. The intermediaries between the client and server elements respond to client requests with previously stored information (i.e. information is not only stored on the server, but also on the client, and on intermediaries). This restriction reduces or eliminates some of the interactions between the client and the server and improves the throughput and service availability of a REST application.

The REST architectural style adds to the LC$SS style the ability to execute on the client, the code provided by the server (COD). It also defines that the interaction between the components of the architecture must be through a uniform interface, simplifying and adding visibility to the interactions. The uniform interface requires resources to be uniquely identified; resources are manipulated through their representations; messages
between server and client must be self-descriptive, and hypermedia must be used as the application state machine. The uniform interface promotes decoupling between architectural components, which allows their independent evolution.

Code on Demand is an optional constraint that allows service consumers to support the execution or deferred service logic that can be used to extend the functionality of the consumer, or temporarily specialize it for a particular purpose (Erl et al., 2012). It improves modifiability, scalability and performance; but negatively impacts on visibility and simplicity.

4.4.2. Understanding REST and Queue Models

Being accurate in the task of choosing the best candidate service is a challenge in the composition of Web services since consumer needs may vary greatly. A key element for the prediction of service quality is client-perceived performance, since it has a significant impact on the user satisfaction (Al-Masri & Mahmoud, 2007b). In other words, it is important to provide an effective and accurate evaluation of performance before investing in the implementation of the composition (Leitner, Hummer, & Dustdar, 2013).

A Web service performance analysis is a process of monitoring and forecasting the service workload, as well as specifying the most effective environment to meet future demands. Given infinite resources, one can expect that any performance level of a service can always be provided. This is not the case with limited resources. There are multiple ways to measure the performance of a service. The most common metrics are: response time, availability and throughput. For instance, response time is the time interval from the moment a service receives a request until the response reaches the consumer and is determined by the network latency and bandwidth and the service capacity.

In queueing theory, a queue system can be specified using the Kendall’s notation (Taha, 2007) as $a/b/c : d/e/f$, where $a$ denotes the time between arrivals to the queue, $b$ is the time that service takes to process a request, $c$ is the number or servers (or replicated services), $d$ is the discipline of queue, that is, the rules followed to select the queue elements; $e$
is the maximum quantity of requests that can live in the system, and \( f \) denotes the number of possible clients which can be finite or infinite (Taha, 2007).

A Web service can be described as the queue model \( M/M/c : GD/N/INF \) (Cao, Andersson, Nyberg, & Kihl, 2003). In a Web service, requests and responses are distributed randomly and denoted by a Markovian (or Poisson) distribution \( (M) \); service instances and servers can be replicated many times \( (c) \). Incoming requests to the server that cannot be processed immediately are stored into the pending requests queue, if there is enough space, otherwise if the queue is full, the response is denied to the client (that is a 503 Server Unavailable HTTP status code is issued as response). The waiting time in the queue is variable, depending on the number of requests in the system already been processed and the processing capacity of the service. The service queue follows one of some disciplines such as First Input First Output (FIFO), Last Input First Output (LIFO) or Random Service Order. In our analysis the queue discipline is considered as a General Discipline \( (GD) \) to avoid implementation details. A Web service can attend a finite number of requests at the same time \( (N) \) and the size of the source from which customers requests arrive is extremely huge and can be considered as infinite \( (INF) \).

Here we will derive a queue model for REST by progressively including the REST architectural constraints defined in the previous section.

### 4.4.2.1. Client Server Service (CS)

The client-server architectural style is the most common way of network-based applications. A service component provides a set of services and receives requests from clients. The client invokes the execution of a service by sending a request to the server via a connector. The server processes or rejects requests and sends a response to the client. The separation of concerns is the principle behind the client-server restrictions. Proper separation of functionality could simplify the server component to improve scalability of a REST service.

Figure 4.1 shows a service that implements the client-server architecture as a queuing model where the main actors are the client and server (Taha, 2007). The client requests can
be processed and answered immediately, or wait in a queue if the server is busy, handling other requests.

The Kendall notation corresponding to a client-server architecture queue model is $(M/M/1 : GD/N/INF)$, where client requests arrive according to a probability distribution $(M)$, there is a single server manager $(1)$, the queue strategy can be $(GD)$. The maximum number of clients that may exist at a given time is $N$, that is a finite number, and there is just one copy of the service (i.e. the service is not replicated). The size of the waiting queue is $N - 1$, and the requests are infinite $(INF)$. In this architecture the server instances are not replicated, therefore service capacity is limited $(C \leq N)$.

4.4.2.2. Layered Client Service (LCS)

LCS adds to Client Server (CS) style an intermediate between the client and the origin server. In this style, we are not considering caches as Web intermediaries, but only pass-trough elements such as routers and gateways; caches are addressed in the following style. The use of intermediary systems facilitates scalability by delegating responsibility to each layer (i.e. security, load balancing, etc.), however, intermediaries increase the processing time and the delay of the interaction between the client and the server (latency). The service is not responsible nor can control such intermediary elements; hence we are not considering them in our queue model. Figure 4.2 shows the LCS architecture as a queue model where each layer’s intermediaries $(I)$ processes client requests $(I)$ until they reach the origin service $(S)$.

![Figure 4.1. A queue model for the client-server style $(M/M/1 : GD/N/INF)$.](image1)

![Figure 4.2. A queue model for the Layered Client architectural style $(M/M/1 : GD/N/INF)$.](image2)
4.4.2.3. Layered Client Cache Stateless Service (LC$SS$)

This style adds to LCS the ability to replicate the server and while keeping stateless interactions between the client and one of the server replicas. That is, there may be various instances of a service, which will find all the information needed to process the request in the message sent by the client, therefore it does not require session information stored on the server to process the request. This may result in repetitions in the content of the messages and thus additional bandwidth consumption. However, since no state information is stored on the server it is possible to free server resources once the response has been sent to the client. Likewise, each request – being independent from the previous ones – can be routed to a suitable service replica without constraints.

The REST architectural style takes also advantage of intermediaries dedicated to provide representation replicas, known as caches. Caches can be located near the client-side, or the server-side. In the latter case, specialized providers (e.g. Akamai, Amazon, Google, etc.) implement diverse cache policies that exploit time and geographic locality among other variables to provide an efficient service. Cache management is far from a trivial task (Psaras, Clegg, Landa, Chai, & Pavlou, 2011), and is influenced also by the business logic, for instance, unlike media content, when dealing with transactions caches may be unnecessary since the information is transient (highly volatile) or it is interesting for very few clients. With this in mind, we are not including cache intermediaries in our model.

The queuing model corresponding to the LC$SS$ style is shown in Figure 4.3 and defined by the Kendall’s notation as $(M/M/C : GD/N/INF)$, which differs from the CS style in that the server is replicated $C$ times to increase the ability to process a greater number of requests from clients. Notice that the system size (i.e. the maximum number of clients that could be served at a given time) must be greater (if waiting queues are supported) or equal to the number of replicas ($N \geq C$).

![Figure 4.3. A queue model for the LC$SS$ style $(M/M/C : GD/N/INF)$](image-url)
4.4.2.4. REST

REST-specific constraints do not modify the proposed queue model for LC\$SS. Therefore, a REST service can be treated as a queuing model of type \((M/M/C : D/N/\text{INF})\). In this model, requests are submitted from the client to the server at an average rate of \(\lambda\) requests per unit of time. Requests that cannot be processed immediately are stored in a queue of pending requests and when there is no more space available in the queue, requests are dismissed (e.g., again a 503 status code is sent with the response). Each instance of a REST service has a request processing capacity average \((\mu)\), and the service is replicated \(C\) times to increase the total processing capacity of the system. Similarly to the previous model, \(N \geq C\) still holds.

4.4.3. Differences between below vs. above capacity for a single service

To simulate a single REST service as a queuing model we use the vector \(W_s = (\mu, c, N)\). For example, service \(W_{s1}\) can process client requests at a rate of one request per second \((\mu = 1\text{req/s})\); it is replicated twenty times \((C = 20)\) and can support up to 50 requests which are being processed or are waiting \((N = 50)\); this service is denoted by \(W_{s1} = (1, 20, 50)\).

The behavior of a queuing system depends on the workload to which it is subjected. If the rate of requests coming into the system is less than the system processing capacity, then requests will be handled normally without placing requests in a waiting queue or even denying requests when there are no more available queue slots. For instance, when the service \(W_{s1}\) must process a workload of ten requests per second \((\lambda = 10\text{req/s})\) which is less than its maximum processing capacity \((c \times \mu = 20\text{req/s})\) then no request has to wait, which guarantees that no request is denied and the response time of the service for all requests (without considering network latency) is about the same (1s). The results of this scenario are shown in Figure 4.4.

When the rate of requests exceeds the capacity of the service, the service behaves differently. In our example, for service \(W_{s1} = (1, 20, 50)\), a scenario where requests arrive at
an average rate of forty requests per second ($\lambda = 40 req/s$), which is greater than the maximum processing capacity of the service, causes the effects shown in Figure 4.5. Requests are processed with different response time as some are addressed directly and others wait in the queue. Service availability decreases because requests arriving when there are no more queue slots available are denied, and finally the processing time of the service does not exceed its maximum capacity even when the workload is higher.

4.4.4. Modeling REST QoS service using Queue Models

There are multiple ways to measure the performance of a service. The most common metrics are response time, availability and throughput. A service response time is the time interval from the request arrival to the server until the response is received by the client and
is determined by two factors: the network latency and bandwidth and the service capacity. The way a REST service processes requests and replies can be modeled using queuing theory. The client requests follow the Poisson distribution (Tao, you Chang, qin Gu, & Yi, 2012; Taha, 2007) and are randomly distributed over a period of time. The generalized queuing model considers the long-term behavior of the queue or steady state, reached after the system has been running for a sufficiently long period of time.

Performance measures of a queue are based on the probability that a certain number of requests may exist in a system at a given time ($p_n$). For instance, $p_0$ would represent the probability of a minimal waiting time due to an empty queue (0). More in detail, useful performance measures are(Taha, 2007): the expected amount of requests in the system.
which includes the expected number of requests from customers in the queue \((L_q)\), the waiting time of a request to be processed by the service \((W_s)\) and the waiting time of the request in the queue \((W_q)\).

Some specializations of the generalized queue system that calculates performance measures have been proposed and proved. The specialized queue model corresponding to a REST service is \(M/M/C : GD/N/INF\), where \(N \geq C\) as described before. The corresponding performance measure formulas originally defined in (Taha, 2007) are described below.

Given the estimated rate of requests \((\lambda)\) that arrive at the system \((S)\), the rate of denied request depends on the rate of arrival at a given time (Formula 4.1). The higher the arrival, the higher the probability of denied requests.

\[
\lambda_{lost} = \lambda \cdot p_N \tag{4.1}
\]

The effective rate of arrival is the difference between the arrived requests minus the denied requests (Formula 4.2)

\[
\lambda_{eff} = \lambda - \lambda_{lost} = (1 - p_N) \cdot \lambda \tag{4.2}
\]

The rate between arrival \((\lambda)\) and processed requests rate of the system \((\mu)\) is defined by \(\rho = \lambda/\mu\). It is possible to calculate the expected number of requests waiting in the queue \(L_q(s)\) using Formula 4.3 (Taha, 2007), and the expected number of requests in the \(L_s(s)\) system using Formula 4.4.

\[
L_q(s) = \frac{\rho^{c+1}}{(c - 1)!(c - \rho)^2} \left\{ 1 - \left( \frac{\rho}{c} \right)^{N-c+1} - (N - c + 1) \left( 1 - \frac{\rho}{c} \right) \left( \frac{\rho}{c} \right)^{N-c} \right\} p_0 \tag{4.3}
\]

\[
L_s(s) = L_q(S) + \rho \tag{4.4}
\]
The processing time for a REST service is defined as a rate between the number of processed request and the effective arrival rate of requests (Formula 4.5).

\[ W_s(s) = \frac{L_s(s)}{\lambda_{eff}} \]  

(4.5)

The waiting time in the waiting queue for a request in a REST system is defined as a rate between the number of requests waiting in the queue and the effective arrival rate of requests (Formula 4.6).

\[ W_q(s) = \frac{L_q(s)}{\lambda_{eff}} \]  

(4.6)

The average service response time \( Q_{ResponseTime}(s) \) is calculated as the sum of the processing time of the services \( W_s(s) \) and the waiting time in the queue of requests \( W_q(s) \), as seen in Formula 4.7. We are not considering network latency since this is out of the scope of responsibility and control of the service.

\[ Q_{ResponseTime}(s) = W_s(s) + W_q(s) = \frac{L_q(s)}{\lambda_{eff}} + \frac{L_s(s)}{\lambda_{eff}} \]  

(4.7)

Replacing the factors in the formula in order to reflect the clients demand dependency, we obtain the following Formula 4.8.

\[ Q_{ResponseTime}(s) = \frac{L_q(s)}{(1 - pN)\lambda} + \frac{L_s(s)}{(1 - pN)\lambda} \]  

(4.8)

Service processing capacity is defined as the number of requests that a system (the whole number of service replicas) can process at a given time. The service availability \( Q_{Availability}(s) \) is usually measured as the percentage of time that the service is ready to be consumed at a period of time (Taha, 2007). However, the availability of a service depends on the number of customers attempting to access a service. If the number of requests \( n \) exceeds the capacity of the service \( C \leq n < N \), the availability is downgraded because many request try to access and compete for the service resources, hence the probability of being served is reduced. Otherwise, if the number of requests \( n \) falls below the service capacity \( 0 \leq n < C \), the availability of the service grows. Service availability is calculated...
as the probability that \( n \) requests exist in the system \( (p_n) \). Service availability is calculated using Formula 4.9.

\[
Q_{\text{Availability}}(S) = \begin{cases} 
\frac{\rho^n}{n!} p_0, & 0 \leq n < c \\
\frac{\rho^n}{c!(n-c)!} p_0, & c \leq n < N
\end{cases}
\]  

(4.9)

The service throughput \( Q_{\text{Throughput}}(s) \) is the number of requests that the service can process in a unit of time (i.e. seconds). If \( Q_{\text{ResponseTime}} \) is the function to calculate the time that takes a request to be processed by the service, then the quantity of requests that can be processed by the service in one second is calculated as the inverse function of \( Q_{\text{ResponseTime}} \), as seen in Formula 4.10.

\[
Q_{\text{Throughput}}(S) = \frac{(1 - p_N) \lambda}{L_q(S) + L_s(S)}
\]  

(4.10)

4.4.5. Applying the Queue Model in a real scenario

We implemented a REST service as a Python script running on an Apache Web server. The experiment was performed locally, in order to discard the effect of network latency, on an Intel PC with 4GB RAM, and 4 cores. We also modeled response time, availability and throughput using the equations proposed in section 3.4. The implemented service is characterized as \( W_s = (1, 20, 50) \) where each instance is able to process one request per second, the service is replicated 20 times and there may be up to 50 requests in the system at a given time. The response time was calculated using the proposed formulas and we obtained the experimental results from the stress tests using Apache JMeter, the tests were run 10 times automatically and averaged, in order to eliminate external factors. For each scenario, results between the models and the experiments are very similar, for the case of response time the results show that when the number of requests exceeds the processing capacity of the service, the response time increases until the requests are denied (Figure 4.6(a)); service
availability decreases exponentially (Figure 4.6(b)), and throughput remains constant once the service reaches the maximum capacity (Figure 4.6(c)).

4.5. Dynamic Composition based in QoS using Queue Model

In order to compose a service dynamically and based on the quality attributes, we need to select the service components as late as possible (dynamic late binding) and at runtime. In this paper we propose a strategy of hybrid dynamic composition that combines the techniques of workflow driven composition and declarative composition. Composition techniques guided by a workflow define an abstract process model for the composed service.
TABLE 4.2. Quality attributes aggregation functions depending on control-flow patterns

<table>
<thead>
<tr>
<th></th>
<th>Sequential ⊗</th>
<th>Loop</th>
<th>Parallel ∥</th>
<th>Conditional ⊖</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Response Time</strong></td>
<td>$\sum_{j=1}^{n} q(s_j)$</td>
<td>$\sum_{i=1}^{k} q(s)$</td>
<td>$\max_{j=1}^{n} q(s_j)$</td>
<td>$\max_{j=1}^{n} q(s_j)$</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>$\prod_{j=1}^{n} q(s_j)$</td>
<td>$\prod_{i=1}^{k} q(s)$</td>
<td>$\prod_{j=1}^{n} q(s_j)$</td>
<td>$\min_{j=1}^{n} q(s_j)$</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>$\min_{j=1}^{n} q(s_j)$</td>
<td>$q(s)$</td>
<td>$\min_{j=1}^{n} q(s_j)$</td>
<td>$\min_{j=1}^{n} q(s_j)$</td>
</tr>
</tbody>
</table>

and later each activity of the model is bound to a particular service. Declarative composition techniques, on the other hand, use customer-defined rules and constraints to determine if the resulting composite service meets customer expectations. In this case, service selection will depend on the fulfillment of each user-defined condition. Quality attributes of the resulting concrete service composition can be evaluated using a utility function using global and local optimization techniques (Alrifai et al., 2012).

On the other hand, the workflow can be seen as an arrangement of control-flow operators (e.g. sequence, conditional, parallel, etc.) that impact the utility function defined in a declarative technique. For example, the response time of a composed service that includes the sequential invocation of service components, results in the sum of the response time of the service components. If the service invocation happens in parallel, the response time of the composed service is the maximum value of the response time of the invoked services. When the invocation occurs within a loop, the composed service response time is the product of service component response time and the times that it is invoked. For the case of the conditional control-flow pattern, the response time of the composed service, in the worst case, is the maximum response time of the components to be chosen by the alternative condition. In (Alrifai et al., 2012), the influence of control-flow operators in the quality attributes of composed services are defined as aggregation functions as can be seen in Table 4.2.

A typical example of service composition is the Travel Planner. In this composite service, the search of places for entertainment (Attraction search) is performed in parallel with the reservation service and an airline reservation service of a hotel. Then, the distance between the place of entertainment and the hotel is calculated and depending on the outcome,
a car rental or a bike rental service may be chosen. The abstract process of this composite service can be seen in Figure 4.7.

According to the algebra of service composition (Hamadi & Benatallah, 2003), the Travel Planner service is defined by \( W_{tp} = \{(AttractionSearch \parallel (TicketBooking \odot Hotelbooking)) \odot DriveTimeCalculation \odot (BikeRental \diamond CarRental)\} \). In the notation, control-flow dependencies are denoted by symbols, for instance \( \parallel \) defines parallel invocation, \( \odot \) defines sequential invocation, and \( \diamond \) defines a conditional invocation. Each task of an abstract process can be implemented using one of many existing services, which results in a composite service that should meet the user expectations. The number of possible ways for composing this Travel Planner service depends on the number of service candidates for each task. That is, if each task in the abstract process has two service candidates, then there are \( 2^6 \) different ways to implement the composite service. The objective of dynamic composition is to find the combination that gives the user the greatest benefits. In the remainder of this section we introduce SAW, which proposes a utility function that identifies the best combination according to multiple criteria.

\[ W_{tp} = \{(AttractionSearch \parallel (TicketBooking \odot Hotelbooking)) \odot DriveTimeCalculation \odot (BikeRental \diamond CarRental)\} \]

---

**Figure 4.7.** Abstract BPMN model of the composite service travel planner

### 4.5.1. Simple Additive Weighting (SAW)

One way to assign a score to each possible service combination is by using the Simple Additive Weighting (SAW) utility function defined by Formula 4.13 (Zeleny & Cochrane, 1982). The SAW technique consists of a process of assigning scores to each combination through two phases: scaling and aggregation or weighting. At the stage of scaling the values of the quality attributes of the candidate services are normalized between 0 and 1 by a comparison between the maximum and minimum values. Then, during the aggregation phase the following formulas are used to calculate the score of a composite service.
To define a particular composed service, we consider an abstract model $CS_{\text{abstract}} = S_1, S_2, ..., S_n$, restricted by customer defined constraints $QoS_{\text{constraints}} = C_1, C_2, ..., C_m$. The process that assigns a score for each combination of services $CS_x = s_1, s_2, ..., s_n$ of the abstract model works as follows: the quality attributes of each candidate service $s$ is represented by the feature vector $q(s) = q_1, q_2, ..., q_k$, therefore $q_k(s)$ represents the $k$ quality attribute value for service $s$. Thus, $Q_{\text{min}}(j, k)$ and $Q_{\text{max}}(j, k)$ defined in Formula 4.11 represent the minimum and maximum value of the $k$ quality attribute for the service candidates to implement a service $S_j$ task.

\[
Q_{\text{min}}(j, k) = \min_{s \in S_j} q_k(s) \\
Q_{\text{max}}(j, k) = \max_{s \in S_j} q_k(s) 
\]  
(4.11)

Then $Q'_{\text{min}}(k)$ and $Q'_{\text{max}}(k)$ defined in Formula 4.12 calculate the minimum and maximum value of the $k$ quality attribute for the abstract model using the aggregation functions $F$, defined in Table 4.2 for each operator of the model.

\[
Q'_{\text{min}}(k) = F_{k,j=1}^n (Q_{\text{min}}(j, k)) \\
Q'_{\text{max}}(k) = F_{k,j=1}^n (Q_{\text{max}}(j, k)) 
\]  
(4.12)

Finally, the utility function assigns a score to the composite service $CS_x$, calculated by Formula 4.13 where the scores obtained for each quality attribute $k$, are weighted by the user preferences $W = (w_1, w_2, ..., w_k)$

\[
U'(CS_x) = \sum_{k=1}^{r} \frac{Q'_{\text{max}}(k) - q'_k(CS_x)}{Q'_{\text{max}}(k) - Q'_{\text{min}}(k)} \cdot w_k 
\]  
(4.13)

4.5.2. Simple Additive Weighting using Queue Model (SAW-Q)

Since dynamic composition aims to find the best service for each task at runtime, the time required to meet this objective has an important role. It is for this reason that
techniques such as integer programming, dynamic programming, heuristic and distributed processing are mainly used in order to reduce the time service selection. However, most techniques ignore service demand while evaluating quality scores, which causes that the resulting composition may not be the most useful in practice. Hence, we modify the SAW technique to take into account the expected customer demand.

Unlike the previous model, in this model each service candidate $s$ is defined by the vector of quality attributes $q(s, \lambda) = q_{rt}(s, \lambda), q_{av}(s, \lambda), q_{th}(s, \lambda)$. That is, by modeling REST services as a queuing system, we can calculate the quality attributes of a service according to the expected customer demand for the composite service ($\lambda$), using the formulas 4.8, 4.9 and 4.10 (Section 4.4.4). Hence, the SAW formulas presented before are modified introducing the user demand:

$$Q_{min}(j, k, \lambda) = \min_{s \in S_j} q_k(s, \lambda)$$

$$Q_{max}(j, k, \lambda) = \max_{s \in S_j} q_k(s, \lambda)$$

(4.14)

$$Q'_{min}(k, \lambda) = F_{k=1}^n (Q_{min}(j, k, \lambda))$$

$$Q'_{max}(k, \lambda) = F_{k=1}^n (Q_{max}(j, k, \lambda))$$

(4.15)

The proposed utility function including clients’ demand, SAW-Q, is defined by Formula 4.16:

$$U'(CS, \lambda) = \sum_{k=1}^r \frac{Q'_{max}(k, \lambda) - q_k^*(CS, \lambda)}{Q'_{max}(k, \lambda) - Q'_{min}(k, \lambda)} \cdot w_k$$

(4.16)

The component services are selected according to the values obtained from SAW-Q for each abstract workflow. There are four factors that determine such value: the number of tasks the abstract model of the composite service; the number of candidates of the composite service services; quality constraints defined by the customer; and the control-flow patterns used in the abstract model. The control-flow patterns determine the execution path.
of a composite service. Depending on this, the component services can be invoked in sequence, in parallel, iteratively, or following alternative conditional execution paths. The quality attributes of a composite service are determined by the aggregate functions in Table 4.2. The utility value ranks an implementation (of all possible) by calculating the expected behavior for each quality constraint. Therefore, since each control-flow pattern influences the utility function in various ways, the number and variety of control-flow patterns of an abstract model influence the choice of the best implementation and the best service for each task. As implementation options grow, the utility function becomes more important in the decision process to choose the best service for a task component.

4.6. Implementation and Evaluation

4.6.1. Implementation

Our principal hypothesis is that a dynamic composition approach based on quality restrictions that does not consider the user demand leads to implementations that can be erroneous in practice. In this section, we describe the implementation of a REST service composer based on SAW-Q, which is a prototype that allows us to illustrate our results experimentally.

Figure 4.8 describes the prototype architecture. The module Receiver accepts the arriving requests ($ request); it is responsible for recording service access and delivering the message to the next component. The App Logic module handles the request and determines its purpose in order to create a new instance of an existing composed service (e.g. POST) or to update the status of an existing one (e.g. PUT). When a new instance is requested, the WS Composer Service module determines the service components for the abstract process ($CS_x$). It accomplishes this task by invoking the Optimizer component which runs either SAW or SAW-Q in order to determine the utility values for the combination of service candidates. The Optimizer uses the Queue model library to perform equations 4.8, 4.9, and 4.10 (Section 4.4.4). The WS Composer Service defines the service invocation plan to follow. On the other hand, if the application intends to update the status of an existing
composed service, then the request is derived to the WS Orchestrator module (Orchestrator Engine) that is responsible for executing the composition plan.

The Orchestrator Engine chooses at runtime the next service candidate with the highest score according the utility function determined by the WS Composer (Invocation Handles), and finally the Control-flow Handler executes the composition plan (i.e. prepares the HTTP request messages to be sent).

Control-flow patterns were implemented considering REST architectural constraints as proposed in (Bellido, Alarcón, & Pautasso, 2013). Hence, we use redirections in order to keep application state on the client. The Control-flow Handler module extends the service response with header metadata corresponding to control-flow patterns as defined. This allows us to implement service compositions that are fully decentralized and stateless.

Services were implemented similarly to those of section 3.5, that is, as Python scripts (Django) running on an Apache Web server with PostgreSQL persistence. The experiment was performed online, the client run on an Intel PC with 4GB RAM, and 4 cores, the server
(Ubuntu) runs on a virtual machine with 4 cores, 3GB RAM. Each service candidate was replicated up to 10 times in a LAN configuration.

4.6.2. Evaluation

4.6.2.1. Theoretical Evaluation

The SAW and SAW-Q techniques may yield different results in certain scenarios. Each scenario depends on the behavior of the component services according to the workload that the composite service must support. To compare the two techniques, we defined an abstract process and generated a random set of service candidates for each task of the abstract process. Candidates are characterized in terms of processing time, degree of replication, capacity (the maximum number of possible clients), and a general REST service description (Section 4.4.4).

In order to demonstrate the differences between SAW and SAW-Q, we created a script that calculated the ranking of the composite services using both techniques. The script receives as a parameter the abstract business process and the number of candidates for each task of the abstract process services. From these data, the script generates the set of candidate services with random attributes within a given range (i.e. processing time varies between 0.5 and 1 second, and replicas vary between 5 and 10). Then, we ranked the generated composite services using SAW and SAW-Q. If the difference between the two rankings was greater than 50 positions the algorithm terminates returning the resulting set of candidate services that gave rise to such difference in the ranking. Otherwise it continues the calculation by randomly generating another set of candidate services.

Once evaluated, candidate services from Table 4.3 give different results depending on using SAW and SAW-Q. The greatest difference in the rankings is found when comparing both techniques simulating a hundred different scenarios. Table 3 presents the pairs of candidate services for each abstract process task for which the differences between SAW and SAW are greater than 50 rank positions. Figure 4.9 shows the results of such scenario using SAW and SAW-Q.
Table 4.3. Characteristics of the candidate services to implement the Travel Planner abstract process.

<table>
<thead>
<tr>
<th>Task</th>
<th>Atomic Service</th>
<th>Processing Time ((1/\mu))</th>
<th>Replication ((c))</th>
<th>Size ((N))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attraction Search</td>
<td>AS(_1)</td>
<td>0.157012985</td>
<td>3</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>AS(_2)</td>
<td>0.182865502</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Ticket Booking</td>
<td>TB(_1)</td>
<td>0.117748184</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>TB(_2)</td>
<td>0.108819849</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Hotel Booking</td>
<td>HB(_1)</td>
<td>0.213374301</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>HB(_2)</td>
<td>0.140538262</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Drive Time Calculation</td>
<td>DTC(_1)</td>
<td>0.212942655</td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>DTC(_2)</td>
<td>0.115707261</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>Bike Rental</td>
<td>BR(_1)</td>
<td>0.112331783</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>BR(_2)</td>
<td>0.129107224</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>Car Rental</td>
<td>CR(_1)</td>
<td>0.108363495</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>CR(_2)</td>
<td>0.166074335</td>
<td>5</td>
<td>36</td>
</tr>
</tbody>
</table>

The implementation suggested by SAW is one of the worst in the ranking obtained by SAW-Q. According to the SAW ranking (Figure 9(a)) the concrete composed service that delivers the best utility values for the Travel Planner is \(C_{63}\), which is defined as \(C_{63} = \{(AS2 \parallel (TB2 \circ HB2)) \circ DTC2 \circ (BR2 \circ CR1)\}\). Whereas for the SAW-Q ranking (Figure 9(b)) the \(C_{63}\) composition is one of the worst and the \(C_7\) composition is the best suited for this scenario. \(C_7\) is defined as \(C_7 = \{(AS2 \parallel (TB2 \circ HB1)) \circ DTC1 \circ (BR1 \circ CR1)\}\).

This result can be explained when considering the demand, \(\lambda = 8 \text{req/s}\) (for simplicity and to compare results the demand was constant in all our experiment). For the case of SAW, \(C_{63}\)’s services are replicated at most 3 times, which causes that the demand exceeds the component service capacity and request end up waiting in the queue (or denied), hence according to SAW-Q, \(C_{63}\) has a bad utility value. For the case of \(C_7\), HB1 and DTC1 are replicated 4 times, which positively influences the utility value.

4.6.2.2. Experimental Evaluation

In order to validate the obtained results empirically, we implemented both techniques SAW and SAW-Q to choose at runtime the best service candidates using the previously described prototype. Services workload (requests/second) analysis was performed using
Figure 4.9. Ranking scores for both techniques: SAW (a) and SAW-Q (b). Composite services are ordered according to their scores.

The load-testing tool JMeter, for a time long enough to obtain stable results. The results obtained in this experiment confirms that SAW-Q’s rankings fit better the experimental results.
Our experimental scenario comprehends a dataset of 6 tasks corresponding to the abstract BPMN model shown in Figure 4.7. For each task in the business process we implemented 2 alternative services, that is, we implemented 12 services which were characterized with a random arrival rate between 4 to 10 requests per second, with random replicas (between 1 to 5) and a random capacity for each replica of 10 to 40 services in the waiting queue.

The response time is a quality attribute that negatively affects the quality of service when it grows. In this experiment the average response time for a service following SAW is higher than the one obtained using SAW-Q when the user demand is taken into account. Figure 13 shows the impact of changes in the demand from 1 to 19 requests per second on SAW and SAW-Q. For each value of the demand, we considered the average response time of the composed service with the highest score for SAW and SAW-Q (Figure 4.10(a)). The differences between both techniques can be appreciated when the demand scales up to 8 requests per second, from then, the services with the highest score according to SAW obtains a higher response time when compared to the top score services obtained by SAW-Q, hence quality of the services selected by SAW is worst than those services selected by SAW-Q. Figure 4.10(b) and Figure 4.10(c) shows the impact of SAW and SAW-Q selections on the availability and throughput of the services with highest score. Again the difference between both techniques appears since the workload is 9 requests per second.

In Figure 4.11 we go further analyzing the situation when the demand reaches 8 requests per second. (\(\lambda = 8 req/s\)). The dots represent the obtained results and the continuous line, the average. Note that in the composition proposed by SAW the standard deviation is bigger than the composition using SAW-Q.

The availability of the services composed using SAW-Q, on average, is greater than the compositions obtained using SAW. Figure 4.12 shows the availability of both services when the demand is 8 req/s. The dots are the obtained results an the continuous line shows, the measures average.
Composite service throughput using SAW-Q is slightly higher than the one using SAW which means that the SAW-Q composite service has the capacity to serve more requests per minute than the SAW composite service. Figure 4.13 shows the results of both services.
when the demand is 8 requests per second. Again, the dots are the obtained results and the continuous line shows the average.
4.7. Conclusions

Quality attributes of a REST service such as, availability, response time and throughput can be modeled with better accuracy using queuing theory since it considers implementation details that are particularly relevant for service architecture and processing time of the application, such as the number of times the service is replicated, the maximum number of clients that can be handled by each service replica at a given time and the request that remain in the waiting queue. These characteristics increase the response time in a subtle way and also determine the number of denied requests. Ignoring such characteristics creates a false idea of the service performance and the service availability. The proposed queue model for REST can be also applied for designing load balancers and determine capacity planning for composed and distributed services.

Naturally, results will vary depending on the architectural design, for simplicity and to reduce noise when comparing SAW and SAW-Q we considered a simple design where servers are replicated and a single queue handles all the requests. More complex and realistic scenarios shall include elements such as load balancers (with their respective queues), various queues (one per replica), etc. conforming to a more complex queue network. We believe that a more refined version of SAW-Q may obtain even more accurate results, and we plan to address such challenge as future work.

Different ways of measuring the quality of a service can lead to errors in the time-based service quality dynamic composition. The quality attributes that do not consider the demand or expected workload affecting the service lead to erroneous predictions and unexpected scenarios. The dynamic composition technique proposed, SAW-Q, considers the attributes of service quality as a function of the demand for requests thus obtained composed services behave better than those determined by SAW.
Finally, the presented approach contributes a deeper analysis on the scalability related attributes. Considering the request demand or workload when modeling services and composed services is critical particularly to certain quality attributes such as throughput, response time, and availability but also fault tolerance and even price, which are not considered in our study. Utility functions may include other variables not subjected to demand but to service design such as flexibility, security, extensibility, etc. Such function may result in a complex model, which can be simplified by considering that naturally not every attribute is equally relevant for every consumer or for every development phase.
5. CONCLUSIONS

5.1. Regarding Decentralized, stateless, complex service behavior in REST

In Chapter 2 a proposal for the design and implementation of complex composed service behavior is presented. This proposal places emphasis on REST architectural constraints, having as goal to achieve scalability and statelessness for the composed service behavior. With these considerations in mind, a set of well-known control-flow patterns that are used to implement simple and complex behavior in traditional Web services were recreated.

The main conclusion from the experience is that a decentralized, stateless implementation of a composed service satisfies REST architectural constraints and provides significant improvements regarding throughput and availability, which are non-functional goals of REST.

Second, when following REST architectural constraints and a decentralized, stateless approach where the client (User Agent) shares the interaction responsibility with the server, control-flow patterns design in REST differ from those in SOA where state (information interaction) is kept in a centralized component (the orchestrator).

Third, one of the extensibility mechanisms of HTTP, namely status codes, was used to implement the presented approach. Other alternatives could be used, such as link headers, or ad-hoc media types (e.g. a specialized JSON document). However, the precedence for link processing indicates that such messages must be processed after the representation is fully received and processed by the client and after users have performed the actions they required (e.g. click on buttons, or run javascript controls), which introduces not only delays but also security risks.

Finally, the presented approach requires that the client knows in advance how to process the messages, so that, it shall be a process-oriented User Agent. In addition, this design
choice also includes the typical vulnerabilities of nowadays User Agents. Additional measures such as digital signatures must be included in order to guarantee a safe interaction between services (mediated by the User Agent).

5.2. Regarding hybrid (static and dynamic) service composition in REST

In Chapter 3 a technique that exploits hypermedia-centric REST service descriptions (defined at design time) is used at runtime to determine the feasibility of service composition and actually enacting a composition with an authentication service based on OAuth. Again a decentralized approach, a choreography, was followed.

The main conclusion from this approach is that hypermedia-centric REST service descriptions can actually serve as a basis for a well-behaved User Agent traverses complex paths on the Web of services. However, since such descriptions are created at design-time, dynamic changes on the service provision (e.g. changes on the service interface) could not be reflected on the descriptions. Descriptions that are out of sync with the service implementations may impede the User Agent to continue its work, although a good service description can provide information to the client developer so that the changes can be easily noticed.

Second, a QoS domain (security) is addressed in this chapter not only because an important authorization technique (OAuth) is an example of a highly scalable and well-known choreography, but also because QoS-aware service composition is a field that have been extensible studied in order to support automatic and dynamic service composition. QoS attributes, particularly security are playing a major role in the current Web due to the massive scale, performance, availability and evolvability requirements that pervade modern Web applications. However, most techniques reduce QoS complexity to single values (e.g. booleans or numbers) when in practice some, such as security, shall be represented as a combination of diverse algorithms and protocols that could be available or not, or even worst shall be tried to follow in order to discover the feasibility of choosing a service as a component for a composed service.
5.3. Regarding dynamic service composition in REST

In Chapter 4, SAW-Q (an extension of SAW) is proposed as a novel dynamic composition technique that follows the principles of the REST style. SAW-Q models quality attributes as a function of the actual service demand instead of the traditional constant values.

The main conclusion is obtained when comparing both techniques SAW-Q is much more accurate than SAW when compared to real implementations, positively improving the quality of dynamic service compositions. Quality attributes of a REST service such as, availability, response time and throughput can be modeled with better accuracy using queuing theory since it considers implementation details that are particularly relevant for service architecture, such as the processing time of the application, the number of times the service is replicated, the maximum number of clients that can be handled by each service replica at a given time and the request that remain in the waiting queue.

Second, choosing the right candidate service in dynamic composition is a critical task. Different ways of measuring the quality of a service can lead to errors. The dynamic composition technique proposed, SAW-Q, considers the attributes of service quality as a function of the demand for requests thus obtained composed services that behave better than those determined by SAW.

Finally, the present approach contributes a deeper analysis on the scalability related attributes. Considering the request demand or workload when modeling services and composed services is critical particularly to certain quality attributes such as throughput, response time, and availability but also fault tolerance and even price, which are not considered in our study.
References


doi: 10.1007/978-3-642-30955-7_10


doi: 10.1007/978-3-642-17881-8_9


