PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

TOWARDS AUTOMATIC QOS AWARE RESTFUL SERVICE COMPOSITION: SECURITY DOMAIN

CRISTIÁN MATÍAS SEPÚLVEDA OLLIER

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

Advisor:
ROSA ALARCÓN C.

Santiago de Chile, November 2012

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

TOWARDS AUTOMATIC QOS AWARE RESTFUL SERVICE COMPOSITION: SECURITY DOMAIN

CRISTIÁN MATÍAS SEPÚLVEDA OLLIER

Members of the Committee:
ROSA ALARCÓN C.
VALERIA HERSKOVIC M.
MARÍA CECILIA BASTARRICA P.
CHRISTIAN OBERLI G.

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

Santiago de Chile, November 2012

To my family, for their support and patience and to my wife, my eternal companion of adventures

ACKNOWLEDGEMENTS

At the end of this long journey I would like to thank Rosa, my advisor, for all her trust in me, her understanding and all the collaboration to accomplish this work. It has been a long process during which we have been widely separated most of the time and without her willingness to support me, this could not have been possible. This enriching experience has yield significant learnings on the Computer Science field which has been as valuable as the chance of sharing and knowing such a nice and admirable person. I could not be more thankful and glad of working with her on this.

This thesis marks the end of my student life which has been fully supported by my family in many ways but particularly in an emotional one. Thanks for all the values granted, your continuous and wise advice and for your patience and love. In every goal I manage to achieve your effort is present and remembered.

Nothing of this would have been possible with all the support of my wife. This started as another project, between many things we have started together, but we didn't expect how many changes in our personal lives were going to happen along with it. It has been tough and challenging to go through it, but tremendously rewarding to accomplish this stage, relying on our love. Thank you for being with me even from a distance and specially in the toughest moments.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Web services have evolved to become a widely used architectural style for both enterprises and independent software development. REST, the architectural style that underlies the Web, is one of the most influencing and followed architectural style in the recent years and is recently provoking research efforts to exploit and extend their functionality while maintaining the non functional properties it is preferred for (e.g. massive scalability, high performance, high evolvability). One of current research topics on REST is automatic service composition. It requires enabling automatic resource discovery and retrieval, which can be achieved trough a machine readable service description such as ReLL. ReLL responsibility is to expose a service description that follows REST architectural constraints. In this work we extend ReLL in order to enrich such descriptions including QoS constraints, particularly security constraints.

Services usually restrict the access to their resources with different kind of security standards and protocols that need to be described in a machine readable format. Existing descriptions focus on traditional, and centralized Web services and result in cumbersome formats. In this work we start with a semantic analysis of security concepts in order to define a security ontology that serves as the basis for the definition of a minimal security constraints format (ReLL-S) for machine-clients consumption that supports Confidentiality, Integrity, Authentication and Authorization security goals. We illustrate our approach through six security approaches widely used on the Web, including the OAuth and OpenId protocols. ReLL-S has been implemented as an extension for RESTler (a ReLL based machine-client) and a full example of Flickr ad-hoc OAuth has been used as a proof of concept, without requiring extra integration effort. This work represents a step forward towards automatic RESTful service composition.

Keywords: Service composition, security, REST, choreographies

RESUMEN

Los servicios web han evolucionado hasta transformarse en un estilo arquitectónico ampliamente usado tanto en software corporativo como independiente. REST, es el estilo arquitectónico que subyace a la Web, es uno de los estilos más influyentes y seguido en los último años, ocasionando investigación que explota y extiende su funcionalidad, manteniendo sus propiedades no funcionales (ej. escalabilidad masiva, alto desempeño, alta evolución). Un tema de investigación actual en REST es la composición automática de servicios, que requiere el descubrimiento y acceso automático a recursos, lo que se puede lograr con descripciones de servicios legibles por máquinas, tales como ReLL, que expone una descripción de servicio que sigue las restricciones arquitectónicas de REST. En este trabajo se extiende ReLL para enriquecer tales descripciones incluyendo restricciones de calidad de servicio (QoS), particularmente en seguridad.

Los servicios restringen el acceso a recursos mediante estándares y protocolos descritos en un lenguaje legible por máquinas. Las descripciones existentes se enfocan en servicios Web tradicionales, centralizados, y son altamente complejas y engorrosas. En esta investigación se parte con un análisis semántico de conceptos de seguridad (ontología) como base de un formato de restricciones de seguridad minimalista (ReLL-S), que guía el consumo de servicios por parte de clientes de máquina, proveyendo las metas de seguridad de Confidencialidad, Integridad, Autenticación, y Autorización. Se ilustra el enfoque mediante seis estrategias de seguridad ampliamente usadas en la Web. ReLL-S ha sido implementado como una extensión de RESTler (un cliente de máquina basado en ReLL). Un ejemplo completo del protocolo OAuth adaptado por Flickr ha sido usado como prueba de concepto, sin requerir ningún esfuerzo extra de integración. Esta investigación representa un avance hacia la composición automática de servicios RESTful.

Palabras Claves: Composición de servicios, seguridad, REST, coreografias.

1. INTRODUCTION

1.1. Problem context

A Web service implements and exposes specific functionality (e.g. search capabilities, rendering videos, allowing to buy a product, etc.) on the Web, that assist a user in the accomplishment of certain tasks and needs.

Traditional Web services, based on WSDL/SOAP protocol, evolved within the corporative world where specific, structured and complex standards were developed. Large companies, with the support of standards bodies such as the W3C (World Wide Web Consortium) and OASIS (Organization for the Advancement of Structured Information Standards), have gathered shared efforts to design artifacts defining the behavior and interaction between Web services like WS-Security, WS-Policy and the entire WS-* standards suite.

On the other hand, REST (*Representational State Transfer*) (Fielding, 2000) the underlying architectural style of the Web has attracted attention due to its non-functional properties such as massive scalability, high performance and high evolvability of the components. Naturally, REST based services have gained attention as a technology that may overcome traditional Web service's complexity and limited scalability, performance and evolvability. REST based services started to be widely used by independent developers and emerging companies, without a regulation similar to the one existing for traditional Web services. Although they have proved organically, under highly demanding platforms (e.g. Twitter API, Amazon services API, etc.) the benefits of its design choices. The lack of complex standards to describe REST based services facilitate them to thrive fast, becoming the most popular services style as reported by Programmable Web, and major players in the industry.

Regardless a service is REST based or traditional, it can be used with other services to fulfil a need otherwise not possible. Combining services data and functionalities can

provide new solutions those services could not provide independently. The process of combining services is know as service composition and is discussed in the next section.

1.1.1. Service composition

Is not rare that for most of the common needs of a client, more than one service is required. Web service composition is the process of combining services functionality into a new *composed* service. Service composition has flourished mainly in enterprise business processes, where reusability of existing functionality and reliability were key elements that drove such interest. Based on traditional Web service description standards, such as WSDL, complimentary standards addressing service composition, such as BPEL4WS (BPEL for short), were developed.

BPEL is an executable language that enables the execution of automated flows implementing business logic processes described through control elements (e.g. conditional, loops, etc) and data flow (i.e. value passing and transformations). BPEL is widely known and represents a centralized, stateful approach that differs from the decentralized, distributed, stateless approach existing in REST.

So called RESTful services are mainly focused on providing access to Web resources,we could say that they are still in a early stage of development, and the efforts have been focused on enabling first a way to have machine clients discovering and consuming services, whereas, complex tasks that are common in the traditional Web services field (e.g. service composition) are not yet addressed, and it is not clear due to REST nature (i.e. decentralized, stateless, hypermedia-driven, etc) whether such approaches can be straightforwardly applied.

A major difference between traditional Web services and RESTful services is the lack of a standard service description. This issue raises big concerns among the REST community since it leads to a loss of independence between client and servers if such description becomes a tight contract. Automatic composition, or even automatic discovery

and consumption of RESTful services from machine-client seems unfeasible without the existence of such service description.

1.1.2. RESTful Service Composition

According to Alonso et al. (2003) service composition includes six dimensions: *component model*, a *data access model*, a *service selection model*, an *orchestration model*, *transactions*, *exception handling*. The component model defines explicitly the assumptions about the components being part of the composition, the fewer the assumptions the more flexible but also the more heterogeneous components are allowed. The data access model defines how data is specified and exchanged between components (Dustdar & Schreiner, 2005). The service selection model determines how a service is selected as a component for the composition either in a static (at design-time) or dynamic (at run-time) way. The orchestration model defines the mechanisms to determine the order, and conditions, of service components invocation, when invocation control is centralized the style is properly called *orchestration*, but when such responsibility is delegated to each component in a distributed fashion it is called *choreography* and the overall activity is performed as peer-to-peer interaction. Transactions defines which transactional semantics can be associated to the composition and how they are guaranteed. Exception handling defines how to handle exceptional states during the execution of the composite service without failing altogether.

1.1.2.1. RESTful Services Composition Dimensions

Resources or *collections* of resources can be considered as the *components* in a RESTful services scenario (Pautasso, 2009a), (Pautasso., 2009). Unlike WSDL-based services, REST resources have standardized, few and very well known assumptions at the application level (i.e. the Web) instead of the domain level. For instance, resources must be identified with a URI, and must be manipulated through *links* and *controls* (e.g. an HTML form that can issue a POST message) embedded in resource's *representations* (e.g. HTML page), which allow service consumers to change resource's state (hypermedia constraint).

The data access model is fundamental in REST, but unlike WSDL-based services, *representations* standardized data models do not refer to domain specific information. In the REST case standardization efforts focus on generic models, or media types (e.g. `text/html`, etc), that can be interpreted by generic clients (e.g. Web browsers). State information is interchanged between clients and servers following the rules of a network protocol (e.g. HTTP). Messages exchanged must include all the required information so that the server can process the request (stateless principle). Application state (i.e. the collection of state information exchanged during client-server interaction) is stored on the client-side.

Regarding service component selection, the *hypermedia* constraint (Hypermedia As The Engine Of Application State) indicates that since servers embed controls and related resource URIs into resource representations, such URIs allow servers to explicitly steer clients across a path of resources and state transitions. Given that, clients must inspect representations for such hints (*dynamic late binding* (Pautasso, 2009b)) in order to *discover* service component URIs. In addition, it is possible to generate a URI from the information conveyed in the representation and ad-hoc rules, or even to choose a particular URI among a predefined set previously registered. Dynamic late binding, however, 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, provided the client can retrieve the URI from the representation. This condition can be hard or impossible to do for some media types, or for representations whose structure has changed (i.e. the links are misplaced). This is not a problem if the URI discovery is performed by humans (e.g. browsing the representation and click on the links), but for machine-clients it poses a huge challenge since they must make sense of the representations and current media-types (e.g. HTML, XML).

There have been efforts to implement transactions in REST, for instance, RETRO (Marinos et al., 2009) supports transactions through locking mechanisms associated to GET and PUT operations. Exception handling is implemented in a standardized way

though HTTP codes that define a policy to propagate or mask errors (Pautasso, 2009b). The REST orchestration model is discussed further in the following subsections.

1.1.2.2. Service orchestrations

The state of an orchestration is controlled locally (Mendling & Hafner, 2008) by a coordinator through actions including data transformations, and component invocation. Communication actions can be unavailable to external parties. Coordinator activities can be implemented as executable processes specified in particular languages such as the Web Services Business Process Execution Language (WS-BPEL) and executed by an orchestration engine.

Because of the hypermedia constraint, REST came up as a navigational style that naturally supports a workflow style of interaction between components. Interaction is decentralized and may represent long-running processes, components are loosely coupled and can mutate at any time. This characteristic is very important since it allows the independent evolution of clients and servers which have been a fundamental property for the evolution of the Web, it also poses a challenge to service composition since components (resources) may change unexpectedly. To cope with this uncertainty clients must have few assumptions about resources and must delay the binding with actual resources (Pautasso, 2009a).

Research related to REST composition has focused on orchestration, being JOpera (Pautasso, 2009a) the most mature framework. In this case, control and data flow are visually modeled and there is an engine that executes the composed resource. The control flow centralizes the communication with components and such communication is implemented by two components: *tasks* that are dynamically bound to *adapters* which perform the communication themselves. Adapters associate tasks with local UNIX programs, remote SSH commands, remote WSDL services invocation, and REST invocation over HTTP. In addition, "glue" adapters perform local computations (e.g. XPath queries, XSLT transformations, Java snippets, and local Java methods). Tasks and adapters

have input and output parameters, for instance, HTTP adapters support four parameters: `Method`, `URI`, `Body` and request headers (`headin`). Adapter invocation order is regulated by *control tasks* that define conditional synchronization points, conditional loops, and forks. The composition is written as a BPEL extension for REST (Pautasso., 2009).

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 (dynamic late binding) is mediated by ReLL descriptions (Alarcón & Wilde, 2010b). ReLL provides a declarative meta-model for RESTful services including mechanisms for URI generation, extraction, parsing, and dynamic late binding, so that a machine client can traverse the graph of resources that underlies a RESTful service. The meta-model introduces also an arbitrary set of types for resources and links at the domain level so that operations such as obtaining a semantic equivalent for the service while traversing its resources is also possible (Alarcón & Wilde, 2010a)

1.1.2.3. Service choreographies

Choreographies refer to the messages exchanged between parties that may belong to different organizations and can be described from a global and local perspective, where the former specifies the message exchanges from an overall point of view and the latter defines it from the perspective of one party. A choreography also captures the dependencies between the interactions, including causal and/or control-flow dependencies (preconditions and order of interactions), exclusion dependencies (a given interaction excludes or replaces another one) data-flow dependencies, interaction correlation, time constraints, transactional dependencies, security constraints, etc. (Barros et al., 2005).

The Web Services Choreography Description Language (WS-CDL) is a W3C candidate recommendation that describes how peer-to-peer participants collaborate regardless of the supporting platform or programming model. Participants are referenced through a WSDL-based service description, which is not executable, it just defines collaborations between participants described in terms of types of roles, relationships, and channels.

While WS-CDL provides a definition of the information formats being exchanged by all the participants, BPEL provides the information formats exchanged from the orchestrator point of view. 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. BPEL specifies *active* rules that are executed to determine what to do once the rule is computed. 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 Muehlen et al. (2005), choreography standards evolution corresponding to RPC SOAP-based and REST-based services, as well as some advantages and disadvantages, is presented. The paper proposes that business processes implemented as choreographies can be represented also as single resources, separating the model in two levels; the higher level corresponds to the choreography or process factory and the lower level corresponds to the resources themselves or process instances. It is not clear though whether the higher level resource's logic implements only an orchestration or is a partial view of the choreography and, other that process state visibility, does not address decentralized choreographies.

In addition, the synchronous messaging nature of the HTTP protocol causes a client to wait for a server response, which may be delayed or even lost if the invoked services invoke as well additional third party services either in a choreography or an orchestration paradigm. Asynchronous interaction may alleviate this problem by allowing the server to notify events to clients when the response is available, which is particularly useful when implementing complex business processes, however, this task is far from trivial. For instance, in Bellido et al. (2011) the OAuth protocol is analyzed as a Web choreography case. They found that one of the main issues relates to the *loss of context* during the interaction. That is, REST requires that all the information that is required by a server to process a message is sent in the message itself instead of being kept on the server-side (stateless). The problem is that clients initiate the interaction, but servers can also redirect clients to different services where a conversation between the client and the

new service occurs out of the interaction band between the client and the former service. Hence clients are forced to provide a callback URL in order to receive an asynchronous update from the server and later regain control of the interaction, and additional parameters that serve as state information that is passed among services offered by diverse servers, must be provided.

In a REST implementation of a choreography (e.g. the OAuth protocol), all the required information for servers to process a request and provide a response must be included in the request message, and also, responses must include all the controls and links required to steer clients towards the next interaction steps. Current implementations of the OAuth protocol do not provide such guidance and forces developers to hardcode URLs and interaction steps into the clients. Furthermore, the lack of a machine-readable service description force developers to hardcode their understanding of the service's Web API (usually documented in natural language), that is, content and format of parameters and payload, as well as certain assumptions, such as invocation order, or information transformation. These characteristics are particularly limiting for supporting automatic service composition, either under the choreography or orchestration paradigms, and even worse if composition criteria is enriched with additional concerns such as quality of service, also known as QoS.

1.1.3. Refining service composition with QoS

In Beek et al. (2007) a review of various approaches for automatic dynamic (using OWL-S for semantic annotations) and static (using BPEL) Web service composition are presented. Composition itself is implemented using various techniques such as automatas, Petri nets, and process algebras, without considering non-functional properties and focusing on orchestration instead of choreographies. Non-functional properties, commonly referred to as quality of service (QoS), 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 and infrastructure, costs, etc.

In Kritikos & Plexousakis (2009), a framework matches users' QoS requirements with the appropriate services that are described at a functional (WSDL) and QoS (OWL-Q) levels. OWL-Q is an ontology that models non-functional properties at a high level of abstraction through concepts such as service, QoS attribute, measurement, metric, scale, time, unit, etc. QoS attributes are narrowed down to specific concepts, for instance, response time is a *performance* attribute measured in time units (e.g. seconds); flexiblity is a *dependability* attribute measured as *inflexible*, *flexible*, or *very-flexible* values, whereas *security* attributes are mostly defined as boolean values that indicate the presence of some security vulnerability. Such measures are normalized by assigning arbitrary numeric values in order to define a QoS vector (e.g. flexibility could be 0, 1 or 2). Matchmaking is performed using a constraint programming approach, that is, global constraint sets (e.g. range values) are defined in addition to the QoS attributes and values associated to Web services. A user request is evaluated against such constraint set, so that a solution space (i.e. a set of Web services) that satisfies the constraints is found.

In WSDL/SOAP based services, non-functional properties are represented by a set of standards collectively known as WS-*. *WS-Policy Attachment* makes possible to annotate WSDL descriptions with additional non-functional information 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 non-functional annotations enrich the service interface so that tasks such as automatic discovery and service composition can be facilitated. In Medjahed & Atif (2007) *WS-Policy* standard is extended in order to include semantic capabilities, modeled in a separated ontology, so that composite services that consider QoS properties can be automatically generated from high-level user specifications. Non-functional properties are treated in a very generic way, for instance, security is modeled as a simple boolean value indicating whether certain security support is provided (e.g. the service supports an specific encryption algorithm), and matchmaking is implemented through

specific rules. In the remainder of this work we will focus on security as a complex example of QoS constraints for REST services composition.

1.1.3.1. Security aware Web service composition

Determining whether a service offers a more secure interaction than others requires more than just defining a scalar metric as could happen with other QoS features. Security is basically defined in three dimensions, *confidentiality*, *integrity* and *identity*. When two services exchange messages where the content is secured in such a way that a third actor intercepting the message cannot read and understand the message, we can say that the interaction guarantees confidentiality. If the received message was not altered by a third party, 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.

The need for keeping interactions secure has grown in the case of composed resources in the Web and so has the need of describing security capabilities of services. On the one hand, for WSDL-based services, these needs pushed the creation of industry standards that allowed web services to describe and advertise their constraint policies regarding security (e.g. WS-Security, WS-Policy). Security constraints have been also modeled as an OWL vocabulary whith SAML assertions (included in WSDL documents) that specify services' security capabilities (Carminati et al., 2007). SAML assertions are the basis for security constraints modeled as boolean formulas while composed services (i.e. BPEL choregraphies) are also annotated with specific *WS-Agreement* tags to include creation constraints. Other approaches, such as Souza et al. (2009), consider the design, implementation and enforcement of services security provision. Under this approach, security requirements are modeled as BPMN annotations (business level), that are translated to BPEL composition annotations (implementation). An engine executes BPEL compositions and translatess BPEL annotations into the corresponding WS-* specification and module configuration in order to enforce the security mechanisms.

For the case of REST, research initiatives on service composition are fairly recent and interest on non-functional attributes has 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 resources served to users according to HTTP operations (Graf et al., 2011). In Field et al. (2011) a server-side obligation framework is proposed; the framework allows designers to extend existent policies with rules that may prevent users to access information and may trigger additional transaction (e.g. sending a confirmation e-mail, register the information access attempt in a log), or may even modify the content of a response or the communication protocol (e.g. require HTTPS).

These initiatives contribute to bring closer the traditional research and techniques of WSDL-based services with the REST perspective, however, their approach focuses on hiding on the server-side the logic related with non-functional attributes, in a way that clients cannot make informed decisions regarding how to proceed, which contravenes the REST principle of *visibility*. That is, by making relevant information visible to Web components (e.g. through metadata, status codes, etc.), such as proxies or clients, it is possible for them to adjust their behavior and achieve some desired properties (e.g. scalability) or 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 also 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.

In addition, the complexity of WS-* standards has resulted in a cumbersome approach that can be dealt only with the appropriate set of middleware tools and infrastructure. There is no such standardization effort in the RESTful services domain, instead. Web security approaches have evolved organically in order to deal with technological and user requirements, resulting into lightweight and decentralized approaches.

1.2. Objectives

Taking ReLL, the general objective of this work is to continue the challenge of enabling automatic composition for RESTful services by extending ReLL descriptions with security constraints. Such extension must be optional, minimalistic, and modify ReLL the least possible. Specific objectives are detailed below:

1.2.1. Modeling a security ontology

In order to be able to describe security constraints, it is necessary to know what kind of security properties shall be considered. This requires a deep understanding of the concepts involved in the services security scope and how they are related to each other. This objective is to *semantically describe the concepts involved in RESTful services security*, and produce an ontology written in OWL for future references.

1.2.2. Defining a security constraints description, ReLL-S

The concepts defined in the ontology must be used to design an extension to ReLL at a conceptual level, that is, at a meta-model level, and to produce a data model that allows to specify security constraints, either using a format similar to ReLL (XML) or an entire new one. The chosen format must be machine readable and must include support for current Web services security approaches that can be distilled from the previous semantic analysis.

1.2.3. Implementation as an extension of RESTler, a machine-client

RESTler has to be extended in order to read and process the security constraints description defined in the previous objective. RESTler should be able to understand the constraints and execute the flows they define. Any new module or component necessary to be able to consume restricted resources has to be implemented and integrated with the already existing functionality.

1.3. Methodology

For accomplishing the objectives, the process is broken down into four different stages. These are described in the next four subsections.

1.3.1. State of art revision

An exhaustive revision of what has been done in the field of service composition will be performed. Special care has to be taken regarding the kind of services studied as composition since most of them correspond to traditional Web services rather than RESTful ones. Also, the revision has to look for QoS aware composition specially the ones considering security, in order to establish the current state towards fully automatic composition.

1.3.2. Services security understanding and analysis

Security is a very broad concept so it has to be scoped and understood in the right context. Based on the work done for the WS-Security standard and taking into consideration that REST have explored and opened new security concerns for Web services, a semantic analysis has to be done for clarifying the concepts involved and how they are related to each other. WS-Security specification is taken as a base to start the work towards a semantic representation of security concepts that must result into a conceptualization in the shape of an ontology.

1.3.3. Bottom-up iterative definition and implementation process

A specific case study is used as the basis for an iterative an incremental definition and implementation of both security constraint description format, and RESTler extensions. The case has to be a RESTful services supporting constraints to restrict the access to some or all of its resources. The implementation of the case's constraints defines the important constructs that need to be present on the constraints description while it defines the necessary extensions on RESTler to interpret and execute the constraints requirements.

1.3.4. Validation, refinement and generalization

ReLL-S and the RESTler extension must be validated according to widely used Web security approaches which may introduce other important security constraints not presented in the case study. If limitations and conflicts arise, we will go back to the previous stage until achieving a stable solution. The final implementation is used as the base for a generalization of the work in order to make it applicable for as many relevant security constraints as possible. To determine which are the most relevant security constraints nowadays, a revision on services repositories has to be done looking for the most used security protocols and standards. Finally the stable version must be validated as well through an ad-hoc approach supported by a major player in the industry in order to evaluate the format flexibility and expressiveness.

1.4. Thesis structure

This document is divided in five chapters. The first one is an overview of all the work accomplished during this research. The main results of the research work is presented as an article in the second chapter which has been submitted to the World Wide Web-Internet and Web Information Systems journal. The third chapter enumerates all the assets elaborated during the research. Overall conclusions are presented in the fourth chapter and finally some options for future research work are detailed on the fifth chapter.

2. TOWARDS AUTOMATIC QOS AWARE RESTFUL SERVICE COMPOSITION: SECURITY DOMAIN

Abstract

Current research on QoS aware service composition focuses on a centralized, synchronous and static setting where quality attributes are modeled through simple metrics. In this research we explore RESTful services composition which is characterized by a decentralized, stateless and hypermedia-driven environment. Our approach focuses on the *security* domain and consists on a ReLL (a REST service description) extension (ReLL-S) that can be processed by machine-clients in order to interact with secured services. An ontology provides a model for the security domain, and our approach supports simple and complex security approaches on the Web.

2.1. Introduction

A Web service exposes data and functionality that can be consumed by humans or other services. Traditional Web services provide a WSDL document as a description of its interface and conditions to be consumed. The process of combining the functionality of two or more services (*components*) into a new service (*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), and the composite behavior (i.e. the services invocation order, as well as the required data transformations) can be determined also in a manual or automatic fashion, either at design-time (static) or run-time (dynamic) (Alonso et al., 2003).

To refine the suitability of a service as part of a composite, additional information such as non functional properties, usually regarded as QoS (Quality of Service), can be considered (Kritikos & Plexousakis, 2009). Most research efforts on QoS aware service composition are focused on traditional Web services which are characterized by a centralized model (orchestration), whereas other paradigms such as RESTful services are characterized by a decentralized, stateless and hypermedia-based approach (Fielding,

2000). REST (*Representational State Transfer*) is the architectural style that underlies the Web and provides it with high scalability, performance, and evolvability properties that are desirable also for Web services. RESTful services are gaining momentum, but it is no clear if traditional research on QoS aware service composition can be straightforwardly applied to RESTful services.

One of the quality attributes that has become highly relevant for Web services (and service composition) is security. The security domain, in addition, becomes an interesting case study because it can be hardly measured by a single unit since it addresses various dimensions (e.g. confidentiality, authorization, etc.) whose provision can determined statically (e.g. must be encrypted) but also may require a dynamic support (e.g. authentication protocols).

In this research we explore security aware service composition for RESTful services. Section 2.2 presents our approach, called ReLL-S, as an extension of ReLL, which is a RESTful service description (Alarcón & Wilde, 2010b), that 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. Security domain is modeled through an ontology (section 2.2.1) and we illustrate our proposal with a set of widely used security approaches for the Web (e.g. HTTP Basic Authentication, OAuth) (section 2.3). Section 2.4 presents implementation details and a case study based on Flickr authentication mechanism. Finally section 2.5 presents our conclusions.

Our research serves as a basis to illustrate the differences between traditional QoS aware service composition (mainly centralized and synchronous) and a decentralized, *choreography-based* approach where parties engage in a complex conversation that requires asynchronous interaction and where parties may change dynamically and require complex handling of the data passed along services.

2.2. Security aware RESTful Service Composition: ReLL-S

Due to the complexity of the concepts, standards and techniques in the field of security, various attempts have been made to provide conceptual frameworks for services security using for instance ontologies as a formalism (Blanco et al., 2008; Garcia & Toledo, 2008; Carminati et al., 2006). Most security ontologies explain security concepts either at a very high level of abstraction (Maleshkova et al., 2010) requiring machine-clients to encapsulate knowledge regarding the implementation of each security strategy (e.g. encoding schemes, cryptographic techniques, server URIs, etc.), hiding information that may be necessary to determine whether an API can be used or not (i.e. whether the service is suitable for a composition), or considering specific security domains in detail (Blanco et al., 2008), or providing a vocabulary of unrelated concepts where responsibilities of future components are unclear (Carminati et al., 2006).

Based on the work of Garcia & Toledo (2008) and the WS-Security standard, we provide an OWL-based ontology comprising around 30 main concepts, classified into three core concepts: `Security goals`, `Security tokens` and `Protocols` (Figure 2.1). The ontology is later used to define a security constraints description called ReLL-S.

2.2.1. Security Ontology

The proposed ontology is an extension to previous works. It aims to include semantics and concepts introduced by modern RESTful services. As discussed before, we identify three security goals, `Confidentiality`, `Integrity`, `Authentication` (identity) and we add `Authorization`, understood as the mechanism that assigns specific access rights to resources. `Encryption` mechanisms provide cryptographic transformations that make messages unreadable by a third party (enabling `Confidentiality`). `Integrity` is usually enabled by different `Signature` algorithms which are `Encryption` mechanisms that produce a

digital signature from the message content in such a way that the recipient can verify that the message has not been modified (Figure 2.1).

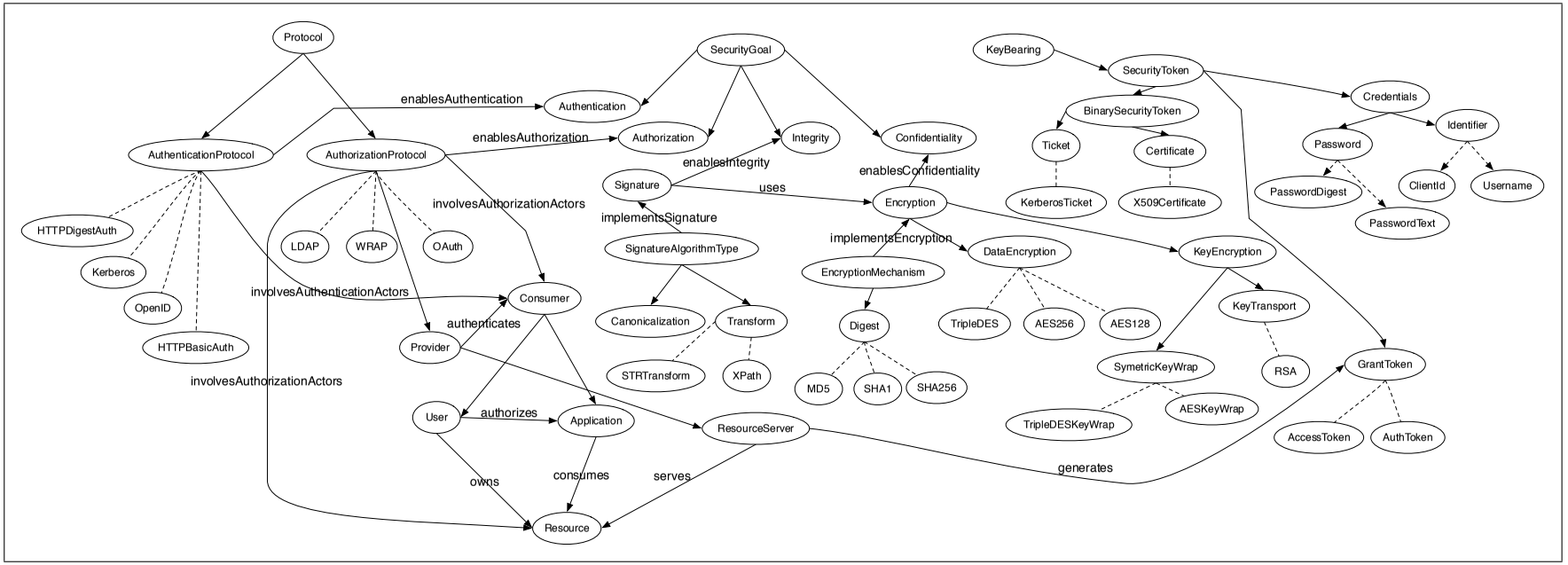

Figure 2.1. Security ontology

The identity of a service Consumer that could be a User or an Application is proved through Authentication mechanisms, such as Authentication Protocols, and 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 a Key bearer of a 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 used in particular security strategies.

Tokens are interpreted and exchanged in the context of security Protocols, for instance, in a SOAP services context, Kerberos and X.509 are examples of authentication protocols. OAuth, on the other hand, is a popular authorization protocol widely used in REST implementations. Unlike previously mentioned protocols, OAuth requires the interaction of three parties, Providers, resource Consumers and Users. A User that owns a resource grants access rights to a third party Application (a Consumer proxy). In order to do so, the Application must provide the proper credentials, such as a ClientId (API key), to a ResourceServer (origin service provider) and engage in the authorization protocol that produces the proper grant tokens (AuthToken).

Other protocols may behave differently, but the basics 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 fundamentals concepts in the realm of security but are insufficient to fully support a secured interaction between REST services since 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) of Web APIs for REST services increase 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 barcode them in specific clients. In order to facilitate to machine-clients the consumption of RESTful services, in Alarcón & Wilde (2010b), ReLL (*Resource Linking Language*) is proposed. In the remaining of this section ReLL is briefly introduced, as well as ReLL-S an extension supporting security constraints.

2.2.2. ReLL: Resource Linking Language

ReLL is a lightweight description for RESTful services that allows a generic machine-client to retrieve REST resources and either traverse the hypermedia 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* (e.g. a network *protocol* such as HTTP is used to issue *request* messages and receive *response* messages) through which linked resources can be accessed (see upper box in Figure 2.2). 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.

ReLL metamodel (Figure 2.2) is the basis of an XML Schema (Third Workshop on Linked Data on the Web, 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. For instance, for XML-based representations an XPath expression allowing selection within XML document trees, becomes a proper language. A selector refers to a *location* (representation's content or its metadata such as HTTP headers) where the expression is applied. Links define an association between a resource's representation

and 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).

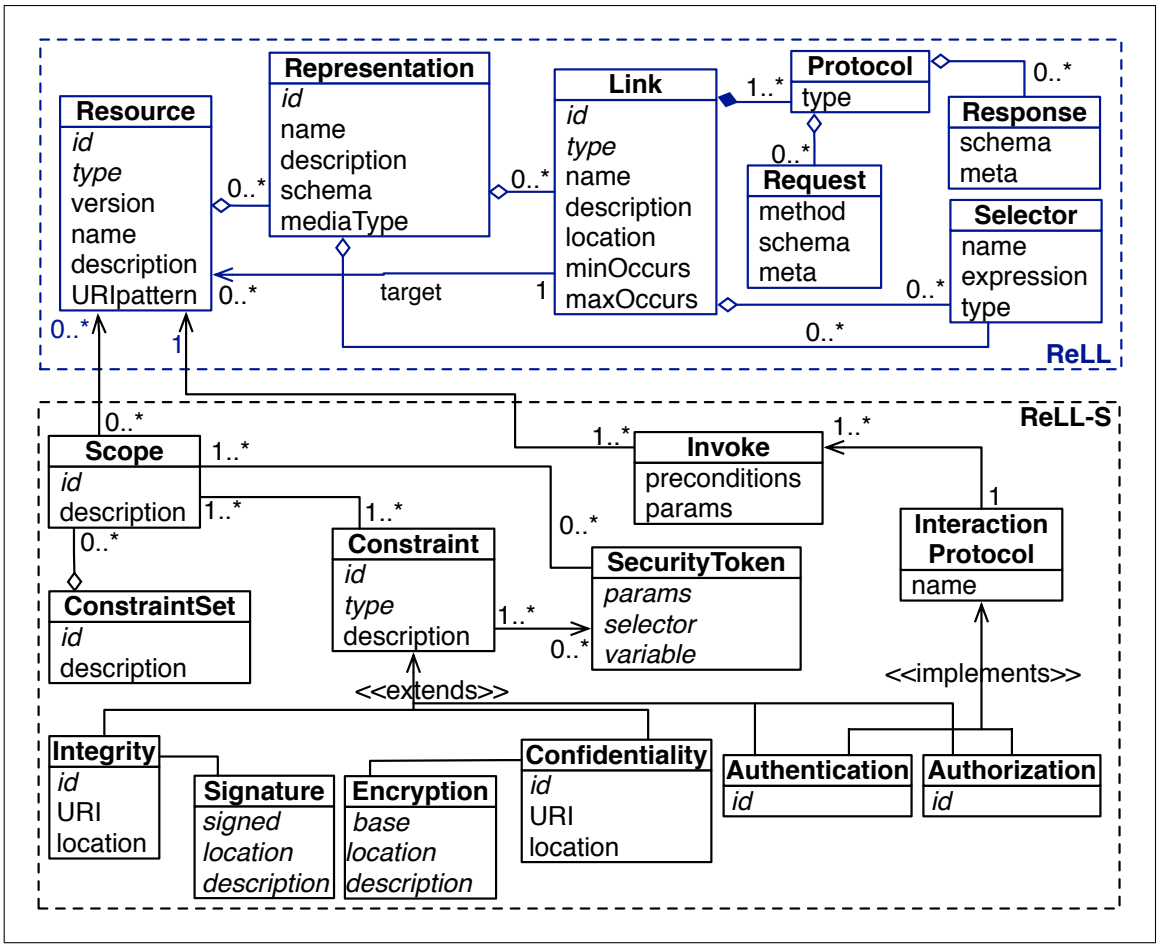

Figure 2.2. ReLL-S a security extension for Security aware service composition

A link has a *link type* which represents the semantics of the link, but their semantics are outside of the scope of the description language. A link can also contain a *protocol* description which specify the rules that govern the interaction with the linked resource. This is important because links in RESTful services not only have application-specific semantics, following the links also may require different ways of using the uniform interface provided by a certain protocol. It is possible that a links must be actually computed

so that a link can be generated by executing an *expression* such as a concatenation, expressions depend on a language (e.g. the concatenation could be written in XPath).

2.2.3. Security Constraints Model: ReLL-S

ReLL descriptions focus on the services' underlying hypermedia as well as the rules to traverse it. QoS are 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 tightening up the contract between servers and clients, must be also considered.

The lower box in figure 2.2 (ReLL-S) depicts our proposal. A `Constraints Set` comprehends a set of `Scopes` which refer to a single or many `Resources` or `Constraints`. Constraints can be reused by different scopes, and related constraints sets can define a rich, specific interaction `Protocol` which may require `Encryption` or `Signature` mechanisms, or `Security Tokens`. Protocols, however, must provide enough detail for machine-clients to resolve the interaction but must not include specific implementation details (e.g how to produce a message) in order to foster flexibility, reuse and avoid strong coupling with any implementation. Each element in the extension module must be identified. Protocols may require to invoke 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 2.2.1, security constraints can be one of confidentiality, integrity, authentication and authorization and are briefly detailed here.

- Confidentiality: 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 are applied to a signed string 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: Refers to the protocols used to prove clients identity. These include HTTP Basic and Digest authentication, OpenID, 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: 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 bind dynamically 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 simple as possible, adding just the minimal elements to be readable by a client machine, but avoiding the complex format 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 where our proposal is used to describe widely used security strategies.

2.3. Supporting current security mechanisms on the Web

Compared to traditional services, security is addressed in a different way on the Web. In Maleshkova et al. (2010) a review of 18% of the self declared REST Web APIs corresponding to the ProgrammableWeb site [1] were analyzed regarding their support of security mechanisms. They found diverse strategies ranging from API Keys (38%), to the OAuth protocol (6%). In this section we present ReLL-S descriptions supporting each of the security mechanisms identified in Maleshkova et al. survey.

2.3.1. 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, there is no confidentiality nor integrity constraint 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`).

[1] http://www.programmableweb.com

Listing 2.1. A simple authentication constraint specification (Api Key)

```
<constraint id="apiKeyAuth" type="Authentication">
  <APIKey>
      <query_param select="$client_id" name="client_id"/>
  </APIKey>
</constraint>
```

2.3.2. 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 do not need to be encrypted or signed and the message content is specified similarly to the listing 2.1.

Listing 2.2. A simple authentication constraint specification (Username/-Password)

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

2.3.3. 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 Base64encode, and a simple signature algorithm where 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 2.3. An authentication and confidentiality constraint specification (HTTP Basic Auth)

```
<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>
  <location url="http://tools.ietf.org/html/rfc2617"/>
  <description>Concat userid a semicolon and password to
     generate base string. The base string has to 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 HTT 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 uses as base string a concatenation of a `userid` and a `password`. The values of such variables must be provided at run-time 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.

2.3.4. HTTP Digest Authentication.

HTTP Basic Auth is not a very secure protocol for user authentication because the encoded entity is transmitted as cleartext 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 a `realm` and a `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, first, when a client attempts to retrieve a resource protected with HTTP Digest, a (`401 Unauthorized`) message is sent by the server including sensible information in the HTTP headers. This representation can be interpreted by a machine-client through a ReLL description as shown in listing

2.4. 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.

Listing 2.4. A ReLL description of an unauthorized HTTP Digest message

```
<service xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="rell.xsd"
         targetNamespace="http://example.org/some_service/"
         xml:base="http://example.org/some_service/">
  <name>Some Service</name>
  <resources>
    <resource id="sampleResource" type="anyResource">
      <uri match="http://www.example.com/resources/.*"
         type="regex"/>
      <representation id="restricted-resource"
         type="http://www.iana.org/assignments/media-types
         /text/xml">
        <name>Restricted Resource</name>
        <selector name="realm" select="realm=(\w*)"
           type="regex" location="header"/>
        <selector name="nonce" select="nonce=(\w*)"
           type="regex" location="header"/>
      </representation>
    </resource>
  </resources>
</service>
```

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 (a la 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 to concatenate 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 2.5. An authentication and confidentiality constraint specification (HTTP Digest Authentication)

```
<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>

<scope token="credentials">
  <constraint id="constraintConf" type="Confidentiality">
    <var name="A1"
       select="concat($userid,':',$realm,':',$password)"/>
    <var name="A2" select="concat($METHOD,':',$REQUESTED_URI)"/>
    <value-of select="digest(checksum($A1), concat($nonce, ':',
       checksum($A2)))"/>
  </constraint>
</scope>

<encryption name="checksum">
  <param name="string1"/>
  <value-of select="MD5($string1)"/>
</encryption>

<encryption name="digest">
  <param name="string1"/>
  <param name="string2"/>
  <value-of select="checksum(concat($string1, ':', $string2))"/>
</encryption>
```

2.3.5. OpenID.

The OpenID 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 2.6. An authentication constraint specification (OpenID)

```
<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 anyway the client prefers. Once the client has the token, it can be passed it as a `query_param` to satisfy the authentication constraint.

2.3.6. 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 access 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 additional grant types. The protocol flow is flexible and depends on the type of authorization that is going to be granted. So in this flow 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 involves more than two entities that can communicate using a protocol that has being designed for client-server communication, as it is HTTP, in a stateless one-to-one conversation. Although, this characteristic presents particular issues to deal with, which makes it really interesting to be described in more detail.

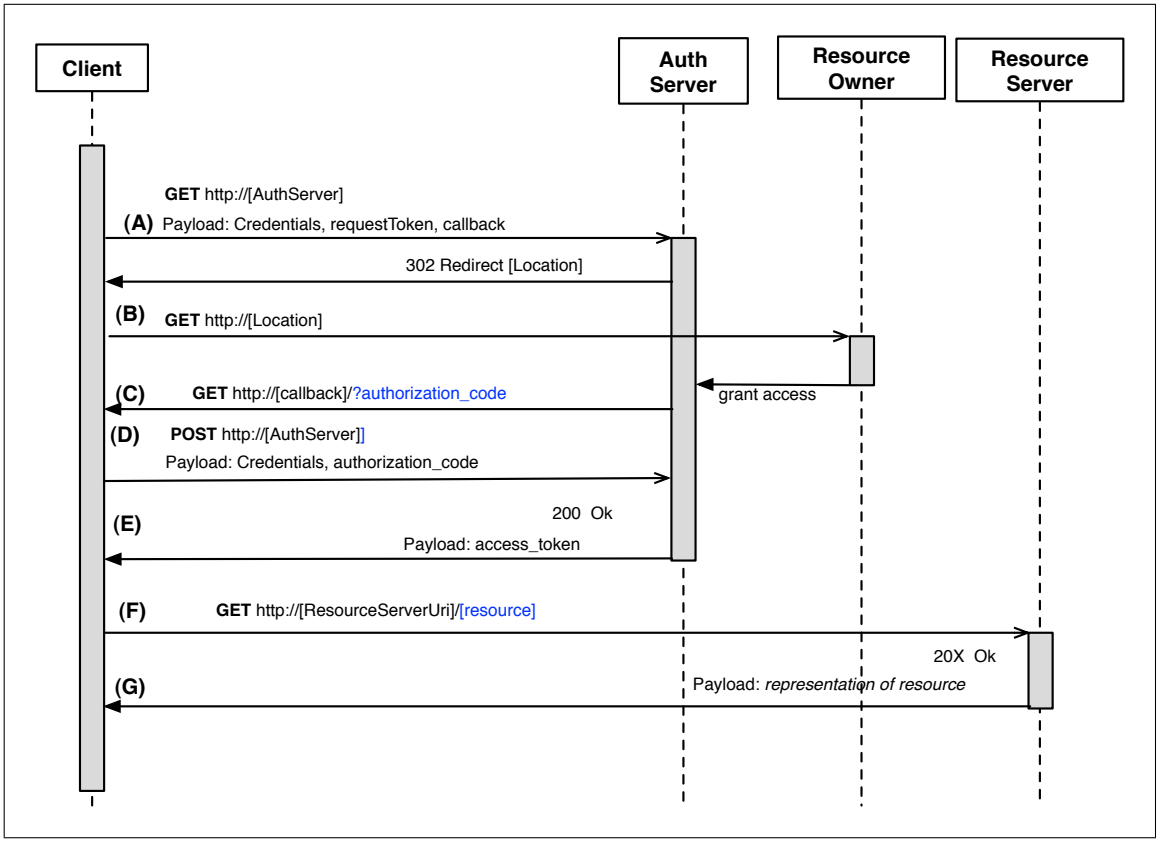

Figure 2.3. OAuth interaction

Once the client has required the access grant, the authorization server starts a conversation with the resource owner that, from the client perspective, occurs completely out of its control. This conversation has the goal of asking the user if he or she is giving authorization to access the resources, but 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 which, after verifying the permission, generates a final token (access token, or oath token) that can be used in future calls to the resource server to access to restricted user resources. It could have an expiration time and some authorization servers provide the feature to refresh or renew the token.

Listing 2.7. An authorization constraint specification (OAuth)

```
<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 two calls must be done by the client 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 in the callback call so the client can use it to keep the flow state. The callback call will include also the authorization code which is used by the client to get the final authorization token during a second call. The order of both calls is basically defined by their preconditions, in this protocol it implies the client will invoke the first call initially because it doesn't have the `auth_code` variable and then it will be able to preform the second call to finally resolve the authorization constraint.

2.4. Implementation and Case Study: RESTler and Flickr

RESTler (Alarcón & Wilde, 2010b) is a machine-client that can consume services and its resources following links in their representations. In order to accomplish this task, it uses information about the available resources described in ReLL file. So far RESTler has just been used to crawl all the resources available using a ReLL description but we want to constraint such feature by introducing security constraints that may force RESTler to engage in security protocols in order to proceed with access to certain resources.

2.4.1. ReLL-S extensions for Restler

RESTler is implemented in Java and was extended with various modules (Figure2.5).

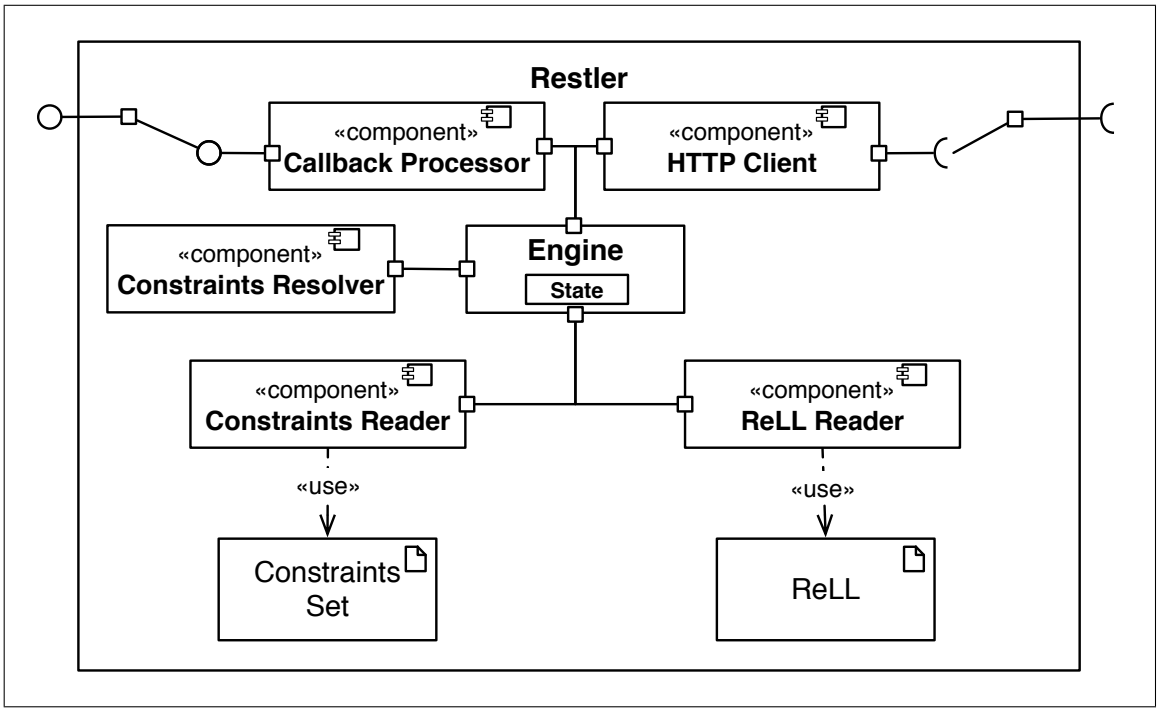

Figure 2.4. RESTler architectural components

A constraints (ReLL-S) reader and parser has been added to the ReLL reader and parser which was slightly modified. A constraints resolver component has been added along with improvements to the HTTP client. As some of the security protocols that RESTler needs to support involve asynchronous interactions, a callback connector has been added as a generic endpoint to receive HTTP requests.

A detail of RESTler logics for interacting with secured resources can be seen in Figure 2.5. Upon need, the RESTler engine fetches a RESTful service ReLL description and processes it. If running in crawler mode, the engine will try to retrieve all the resources (i.e. in a loop) it can reach from a provided seed using the ReLL description to derive some meaning regarding how to interact with such seed and the retrieved representations. Currently, the engine performs a recursive execution in order to implement an in-depth crawling. Resources URIs are retrieved dynamically by inspecting received representations and the HTTP client is used to fetch the next resource representation.

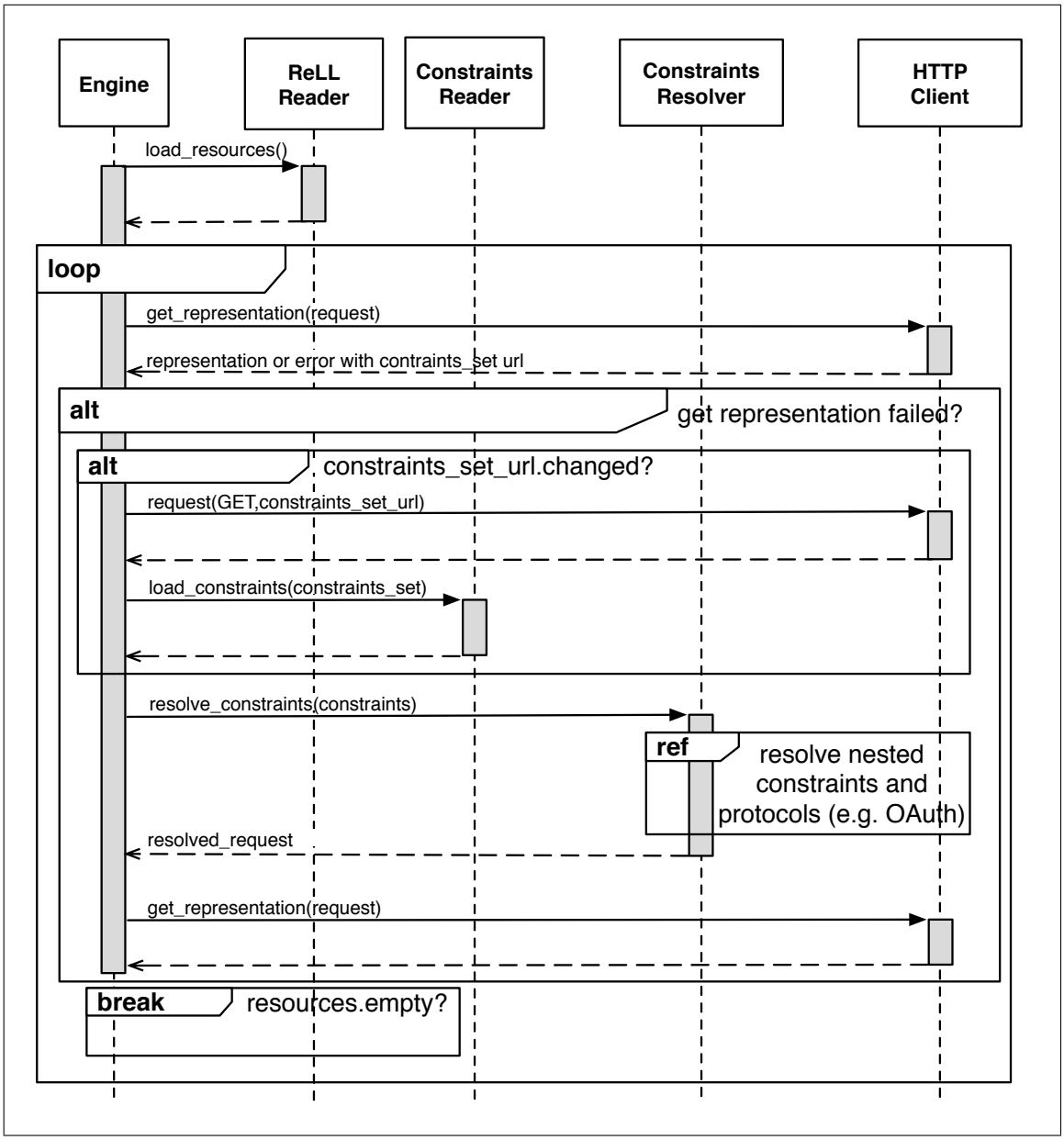

Figure 2.5. RESTler Sequence Diagram for crawling restricted resources

If a resource is protected, the representation will include metadata (an HTTP header) indicating the causes. A (`404 Unauthorized`) code may indicate the cause of the failure. Under a hypermedia driven approach, the message shall include all the information required so that the user-agent can reach its goal. Such information shall include

the address of the security constraints set so that the user-agent can fetch (`GET`) the description and proceed accordingly.

The obtained description is parsed by a reader and processed by the constraints resolver that will execute the corresponding instructions. As a result state information must be produced so it can be used by the user-agent to retry its initial goal, that is, to fetch the protected resource. A complex example of one of such security constraints for consuming protected resources provided by the Flickr service is illustrated below.

2.4.2. OAuth for Flickr

OAuth does not dictate how it must be implemented and has some room for customization, so implementations can vary from a service to other. There are four different types of authorization grant, there are different ways to require the user authorization and authentication, the authorization server could be service independent from the resource server or they could be integrated, etc. Due to this flexibility it is worth to analyze one specific implementation of the protocol. Figure 2.6 depicts the steps followed by the engine when resolving the constraint set description for Flickr.

The constraint resolver initiates by processing and following the instructions of a constraint set description, as is the case of Listing 2.8. As indicated, the first constraint to be resolved is an authorization one (`flickrAuth`), since the scope of the constraint affects the resource that the engine is trying to retrieve (`flickr:members`).

Flickr implements its own application authentication API. It has to be called to obtain an access token necessary for many of the service API methods that an external application could want to invoke. So the security constraint over those methods is an authorization constraint that specifies a protocol. That protocol is an implementation of OAuth specifying two calls that must be invoked in order to get an access token that allows to retrieve restricted resources from the service. The first one in this case is http://www.flickr.com/
services/auth/ that receives the permissions that will be requested to the resource owner,

these could be read, write or delete. This call is an asynchronous one, which triggers a special behavior detailed below.

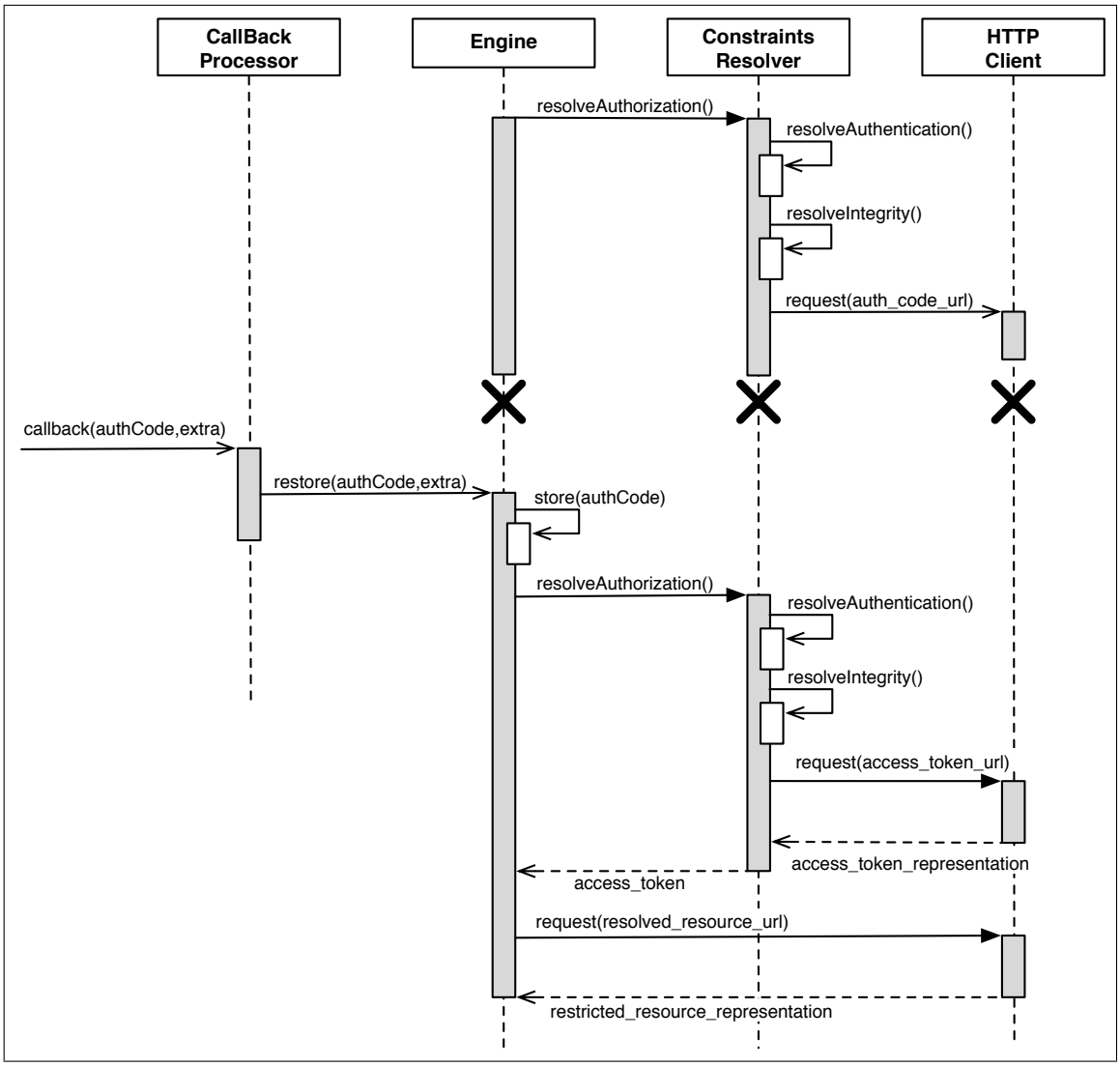

Figure 2.6. Flickr OAuth sequence diagram

The call has constraints itself that RESTler has to inspect too, in this case there are two: authentication and integrity. Both of them require including elements to the request, the former needs the application key, that must be previously obtained from the service in design time, and the latter needs a digital signature built under the service specifications. In order to sign the message RESTler must have a library to implement this custom

signature because it is not a standard mechanism. The Flickr signature is built calculating the md5 hash (an implicit confidentiality constraint) over the string composed by the application secret token - that is also registered in design time - concatenated with request parameter values in alphabetic order according to its keys.

Once the authentication and integrity constraints are resolved, finally the first call needed for authorization is invoked including the resource currently being retrieved as the extra parameter. Given this is an asynchronous call it does not return the expected authorization code in the response of the request which makes the current resource retrieval flow to be terminated because it doesn't have the preconditions for the next call, then RESTler could continue with another resource or just finish its execution depending on the mode it is running on.

Listing 2.8. An authorization constraint specification (OAuth) for Flickr

```
<constraintSet xmlns:wl="http://www.example.com/wl-security/">
  <scope resource="flickr:members">
    <constraint id="flickrAuth" type="Authorization">
      <accessToken>
        <query_param name="auth_token"
           select="$auth_token||wl:OAuth"/>
      </accessToken>
    </constraint>
    <constraint id="appAuth" type="Authentication">
      <appToken>
        <query_param select="$api_key" name="api_key"/>
      </appToken>
    </constraint>
    <constraint id="md5Sign" type="Integrity">
      <query_param select="FLICKR_MD5" name="api_sig"/>
```

```
  </constraint>
  <constraint id="md5Conf" type="Confidentiality">
    <mechanism name="wl:MD5"/>
  </constraint>
</scope>

<scope resource="securityToken">
  <constraint ref="appAuthenticationConstraint"/>
  <constraint ref="signatureConstraint"/>
  <constraint ref="confidentialityConstraint"/>
</scope>

<protocol name="wl:OAuth">
  <invoke
     url="http://www.flickr.com/services/auth/?perms=read"
     pre="not($auth_code)">
    <query_param select="$state" name="extra"/>
    <query_param select="$callback" name="callback"/>
  </invoke>
  <invoke url="http://api.flickr.com/services/rest/?method=
     flickr.auth.getToken" pre="$auth_code">
    <query_param select="$auth_code" name="auth_code"/>
    <store selector="token" persist="auth_token"/>
  </invoke>
</protocol>

<digitalSignature name="FLICKR_MD5">
  <signed  input="api_secret"/>
```

```
    <location
       url="http://www.flickr.com/services/api/auth.spec.html"/>
    <description>Sort parameters value alphabetically based on
       parameter name and concat this to your SECRET
    </description>
  </digitalSignature>
</constraintSet>
```

Some time later, depending on the outbound process where the user eventually grants the access, the resource server requests an endpoint in the callback connector of RESTler, that is registered with the resource server during design time. The callback request contains the authorization code and the extra parameter that RESTler included in the asynchronous call, both as URL parameters. At this time RESTler stores the auth code in its local variables and uses the extra parameter as the seed to restore the resource retrieval. With this, RESTler retries to resolve the authorization constraints but this time it will be able to invoke the second step in the authorization protocol that is getting the access token. That call also has constraints: integrity and authentication, so the same process of the previous step has to be done including the authorization code as a parameter and finally RESTler gets the access token, it stores it in a cookie collection to use it the next time without following the whole protocol and send it along other parameters to retrieve the restricted resource.

2.5. Conclusions

Considering the complexity of nowadays technology, 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. Such task will fall under the

choreography category and even though considering the understanding of RESTful service interfaces (as supported by ReLL) or even its related semantics (as proposed by the Linked Data initiative), fundamental consideration such as QoS must be taken into account for the case of RESTful 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. In the absence of a security description, a machine client is simply not able to retrieve restricted resources, the understanding of the service interface (protocol, method, parameters) and the knowledge of the resource URL is not enough.

Unlike WSDL, SOAP-based service composition, REST architectural constraints requires loosely coupling of the components in order to foster independent evolvability (i.e. a service provider can change its interface unexpectedly), hence service descriptions including functional and non-functional properties should not tighten such coupling. WSDL descriptions including QoS requirements become a strong contract that increases the coupling between clients and servers. ReLL provides the means for a client to interpret a RESTful service interface, but many interpretation may hold and ReLL do 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, RESTful service composition must take into account the dynamic nature of RESTful services, hence automatic approaches must also include automatic and dynamic recovery mechanisms. For a machine-client to determine whether it may or not interact with a service under security constraints, it may consider whether it supports the required encryption mechanisms and signature algorithms, and also whether it can be engaged into a series of interactions (e.g. simple request-response or a protocol) in order to satisfy the constraint. Following that path may also have an impact on availability, performance of the machine-client (i.e. latency induced by the unexpected interaction), unexpected complexity, and

risks since it may interact (e.g. due to redirections) with unexpected services. Security models then cannot be reduced to boolean values or integers since it requires to determine to which extent the machine-client is willing to distract its goal in order to satisfy a constraint. Our approach serves as the basis for the machine-client to determine the space of new services (URLs) to visit, the number of interactions (steps, or calls) to expect, and the complexity of the required algorithms beforehand, so based on a multidimensional metric, derived from the ReLL-S metamodel, it may asses the suitability of a service component. Since the constraint sets documents are discovered on run time, changes on the security approach must be reflected on the the security constraint document.

ReLL-S sits in between a flexible service documentation, which for RESTful services usually ends up in an ad-hoc HTML document of the service's API specification, and a highly structured format that describes services, like WS-Security, and becomes such a strong contract that can serve as stub. While the machine client needs to access restricted resources, the description must also leave to the client the responsibility of implementing specific mechanisms for encryption, canonicalization and others security task, that can be identified but not described to the detail by ReLL-S. Finally, RESTful service interaction greatly benefits from hypermedia, since it allows servers to change its interfaces dynamically and since the proper URLs (and hopefully the links to the updated ReLL and ReLL-S descriptions) are embedded in the response messages (e.g. a 404 Unauthorized), clients may recover dynamically while allowing the required decoupling. A fundamental task then, for RESTful services automatic composition focuses 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.

Acknowledgment

Research supported by the Center for Research on Educational Policy and Practice (CONICYT), Grant 11080143.

3. FURTHER RESULTS

3.1. Security Ontology

The WS-Security standard has defined a semantic base of security concepts. Using Protege software, an open source ontology editor and knowledge-base framework, a security ontology has been built to describe how this different concepts are related to each other. The ontology has been written in OWL language and it has turned to be an almost hierarchical model with three root concepts: Protocol, Security Goal and Security Token and a Figure depicting its fundamental concepts can be seen in Figure 2.2.

Traditionally, security has three different goals: Integrity, Confidentiality and Identity. After analyzing the current service security requirements, the last goal has been split into two different and more specific ones: Authentication and Authorization. This is intended to meet the needs that RESTful services have raised around allowing third party services to access resources owned by a service, a process that requires resource owners (i.e. users) authorization in a secured fashion.

The goals can be reached using standard mechanisms and protocols specifically defined for it. The latter ones are actually protocols defining the interaction between different clients and services in a composition. For this reason, they have been differentiated in the ontology. Protocols as well as some mechanisms, use different kind of tokens during the communication between the actors involved in their execution, which leads to the third root concept: Security Tokens.

The security ontology aims at identifying important concepts used so far in traditional services with those introduced by new technologies. Also it has been a necessary step towards the definition of security constraints and a helpful exercise in order to define the format to be used for describing them.

3.2. Security Constraints Set format

This is the main result of this work and it has concentrated most of the efforts. Using XML, the constraints set document has been defined to support the description of different kind of security constraints a service can have. The constraints types are based on the different security goals defined on the ontology: Authorization, Authentication, Integrity and Confidentiality. As resources of a certain type can have multiple constraints applying on them, scopes that associate all the constraints applied on a resource type have been defined as the highest level element in the constraints set. On the other hand, the same constraint could be applied on multiple resource types, so in order to support that feature, scopes can include references to previously defined constraints.

Just like the ontology defines it, goals can be reached using different protocols and mechanisms, so in the constraints set both of these are independent nodes that can be referred by constraints. This establishes how and when the protocols and mechanisms have to be executed and what constraints they are solving. Due to the fact they have been defined independently of the constraints, a protocol can be used by more than one constraint.

This format has proved to be able to fully describe services using the following security mechanisms: API Tokens, username and password, HTTP Basic, and HTTP Digest Auth, OpenID and OAuth. The last two have presented the most different efforts as they require to implement interaction protocols which have been successfully modeled and described.

The constraints set format has been implemented as a RESTler extension module, and validated to consume a restricted resource from the Flickr service. In order to do this, the machine-client (RESTler) has been extended to understand and execute the format and also to support the interaction protocol flows. This updated version of RESTler is a secondary result that can be used for further work on service composition.

3.3. Publications

As for publications, a workshop paper (Bellido et al., 2011) exploring choreographies for REST and extending the OAuth messages with semantics using the Web Linking protocol has been published. This paper has served as the basis for understanding relevant characteristics of REST choreographies, namely, statelessness and asynchronous communication and its consequences, such as the loss of context when out-of-band interaction occurs. The content of Chapter 2 is part of a paper under submission to the World Wide Web-Internet and Web Information Systems, Springer and reports the core of this research work.

4. CONCLUSIONS

This work started with a semantic analysis of security concepts organized in an ontology representing them as well as their relationships. This task raised the differences of security between traditional and RESTful services, as the latter require different interaction protocols that behave under the choreography, decentralized paradigm instead of a centralized orchestration as is usual in traditional Web services. Then, my efforts were focused on designing and implementing a security constraints description format following an iterative and incremental process which included continuous validation using RESTler as the machine-client. As the constraints set were something new for RESTler, it had to be extended and keep pace with the changes the constraints set document were experiencing as it evolved leading to a stable functional implementation.

It is expected that the results of this work contribute to shed light to QoS handling and become a step towards automatic RESTful service composition, considering security constraints. In the case study presented, the extension implemented for RESTler made possible to define a constraints set for Flickr service authentication, written as a ReLL-S document, and execute a machine-client interaction to consume restricted resources without additional intervention. This is interesting since Flickr OAuth implements more complex and ad-hoc data handling than a simple, standards-base OAuth. ReLL-S has proved to be flexible enough to support the description of other widely used security approaches.

Even though it requires an extra effort compared to generate a human readable HTML version of the services, that is a natural language description, as it is the current practice, describing security capabilities of RESTful services in a machine readable way has proved to be useful. It makes possible to have machine clients crawling and consuming resources even when these are restricted by security mechanisms. It also allows the service and the client to evolve independently as long as the security constraints set keeps updated. If that is not the case, it serves as a medium to identify such points of change and may facilitate developers to fix the clients easier rather than reading a new

specification written in natural language to find such changes. Finally, the constraints set can be served as another resource of the service so this can be updated, cached, and scale for its consumption. Clients able to consuming this kind of resources (*.rells) may adapt themselves to changes on the service interface as well as to better manage the caching of sensitive information with its own replacement rules. Standards such as HTML 5's Web Storage may become an improvement towards a more sensitive and secure client-side data handling.

5. FUTURE WORK

At the end of this work is clear that there is still a lot of work to do in order to fully support automatic RESTful service composition. The constraints set format is a step towards understanding QoS constraints in composition. It is clear that complex QoS properties such as security cannot be reduce to simple metrics such as booleans indicating whether some security mechanism is supported or not. The provided ontology serves also as a classification of such mechanisms and may facilitate such decision. The question however shall be posed in the other direction, that is, *as a client, am I able to engage in an interaction where the server demands from me to support certain security mechanisms?* In addition, the very nature of REST does not provide any guarantee that such properties will remain stable during the interaction, servers may change arbitrarily their interfaces and if proper linking guiding is provided (hypermedia) clients shall be able to recover dynamically of such failures. Clients, however, must be prepared with a repertoire of mechanisms to adapt to such changes. Again, an ontology, understood as an agreed description of mechanisms relevant in a domain, may assist clients in preparing their capabilities. A limit however can be reached, and ReLL-S provides the medium (i.e. location) to indicate clients how to be prepared for future interaction even though the current may fail.

The constraints set format itself has still plenty of room for improvement. The current implementation is supported by the extended version of RESTler, but it must be tested to support a wider variety of security mechanisms in order to guarantee its flexibility. Some elements defined on the constraints set, like the *store* node and the *pre* and *persist* attributes, require refinement in the behavior of the machine client that RESTler includes, but a more generic standard-base approach is necessary and probably Web Storage may become such improvement.

Constraints set can be either extended or integrated with mechanisms description languages like XML Signature, although supporting custom signature methods, would

still become a challenge. Constraints set could be integrated with more standard automation languages so other kinds of machine-clients could understand and support the format. Another strong dependency of the extended RESTler is the capability to support interaction protocols with asynchronous flows, in this case implemented by a Callback connector. Having a way to describe different ways to support the asynchronous flow (e.g. Web sockets) would be another valuable improvement to the portability of the format.

One important aspect that should be considered in a fully automatic composition implementation is the transactional support. Even though there have been efforts on supporting transactional recovery from functional failure states, supporting recovery from security interaction protocols failure states is not considered at all, setting limitations on the ability of the client to continue its resource retrieval task within a service composition context. For example, both OpenID and OAuth protocols involve out-of-bound flows of interaction where the resource owner, usually a human user, authenticates and/or authorizes a third party application, but it may be the case that either the step is not successful or it never even begins, hence, the protocol cannot be completed and this could raise an exception to recover to a previous state.

References

Alarcón, R., & Wilde, E. (2010a, June). *From restful services to rdf: Connecting the web and the semantic web* (Tech. Rep. No. 2010-041). Berkeley, California: School of Information, UC Berkeley.

Alarcon, R., & Wilde, E. (2010, April). Linking data from restful services. In *Third workshop on linked data on the web.* Raleigh, North Carolina.

Alarcón, R., & Wilde, E. (2010b). Restler: Crawling restful services. In *19th international world wide web conference* (p. 1051-1052).

Alarcón, R., Wilde, E., & Bellido, J. (2010). Hypermedia-driven restful service composition. In *6th workshop on engineering service-oriented applications (wesoa 2010).*

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2003). *Web services: Concepts, architectures and applications*. Berlin, Germany: Springer-Verlag.

Barros, A. P., Dumas, M., & Oaks, P. (2005). Standards for web service choreography and orchestration: Status and perspectives. In C. Bussler & A. Haller (Eds.), *Business process management workshops* (Vol. 3812, p. 61-74).

Beek, M. H. ter, Bucchiarone, A., & Gnesi, S. (2007). Web service composition approaches: From industrial standards to formal methods. In *Iciw* (p. 15). IEEE Computer Society. Available from `http://doi.ieeecomputersociety.org/10.1109/ICIW.2007.71`

Bellido, J., Alarcon, R., & Sepulveda, C. (2011). Web linking-based protocols for guiding restful m2m interaction. In *Lecture notes in computer science.* Springer.

Blanco, C., Lasheras, J., Valencia-García, R., Fernández-Medina, E., Álvarez, J. A. T., & Piattini, M. (2008). A systematic review and comparison of security ontologies. In *Ares* (p. 813-820).

Carminati, B., Ferrari, E., Bishop, R., & Hung, P. C. K. (2007). Security conscious web service composition with semantic web support. In *Icde workshops* (pp. 695–704). IEEE Computer Society. Available from `http://dx.doi.org/10.1109/ICDEW`

.2007.4401057

Carminati, B., Ferrari, E., & Hung, P. C. K. (2006). Security conscious web service composition. In *Icws* (pp. 489–496). IEEE Computer Society. Available from http://doi.ieeecomputersociety.org/10.1109/ICWS.2006.115

Decker, G. (2006). *Process choreographies in service-oriented environments*. Unpublished master's thesis.

Dustdar, S., & Schreiner, W. (2005). A survey on web services composition. *IJWGS*, *1*(1), 1–30. Available from http://dx.doi.org/10.1504/IJWGS.2005.007545

Field, J. P., Graham, S. G., & Maguire, T. (2011). A framework for obligation fulfillment in rest services. In *Second international workshop on restful design (ws-rest 2011)* (p. 59-66).

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. Unpublished doctoral dissertation, University of California, Irvine, Irvine, California.

Franks, J., Hallam-Baker, P. M., Hostetler, J. L., Lawrence, S. D., Leach, P. J., Luotonen, A., et al. (1999, June). *Http authentication: Basic and digest access authentication.* Internet RFC 2617.

Garcia, D. Z. G., & Toledo, M. B. F. de. (2008). Web service security management using semantic web techniques. In *Sac* (p. 2256-2260).

Graf, S., Zholudev, V., Lewandowski, L., & Waldvogel, M. (2011). Hecate, managing authorization with restful xml. In *Second international workshop on restful design (ws-rest 2011)* (p. 51-58).

Hammer-Lahav, E. (2010, April). *The oauth 1.0 protocol.* Internet RFC 5849.

Kritikos, K., & Plexousakis, D. (2009). Requirements for qoS-based web service description and discovery. *IEEE T. Services Computing*, *2*(4), 320–337. Available from http://doi.ieeecomputersociety.org/10.1109/TSC.2009.26

Kübert, R., Katsaros, G., & Wang, T. (2011). A restful implementation of the ws-agreement specification. In *Second international workshop on restful design (ws-rest*

2011) (p. 67-72).

Maleshkova, M., Pedrinaci, C., Domingue, J., Rey, G. A., & Martinez, I. (2010). Using semantics for automating the authentication of web APIs. In P. F. Patel-Schneider et al. (Eds.), *International semantic web conference (1)* (Vol. 6496, pp. 534–549). Springer. Available from http://dx.doi.org/10.1007/978-3-642-17746-0

Marinos, A., Razavi, A. R., Moschoyiannis, S., & Krause, P. J. (2009, July). Retro: A consistent and recoverable restful transaction model. In E. Damiani, R. Chang, & J. Zhang (Eds.), *2009 ieee international conference on web services* (p. 181-188). Los Angeles, California: IEEE Computer Society Press.

Medjahed, B., & Atif, Y. (2007). Context-based matching for web service composition. *Distributed and Parallel Databases*, *21*, 5-37. Available from http://dx.doi.org/10.1007/s10619-006-7003-7 (10.1007/s10619-006-7003-7)

Mendling, J., & Hafner, M. (2008). *From WS-CDL choreography to BPEL process orchestration.* Available from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.6099;http://wi.wu-wien.ac.at/home/mendling/publications/TR06-CDL.pdf

Muehlen, M. zur, Nickerson, J. V., & Swenson, K. D. (2005). Developing web services choreography standards - the case of REST vs. SOAP. *Decision Support Systems*, *40*(1), 9–29. Available from http://dx.doi.org/10.1016/j.dss.2004.04.008

Papazoglou, M. P., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *IEEE Computer*, *40*(11), 38–45. Available from http://doi.ieeecomputersociety.org/10.1109/MC.2007.400

Pautasso, C. (2009a, July). Composing restful services with jopera. In A. Bergel & J. Fabry (Eds.), *International conference on software composition 2009* (Vol. 5634, p. 142-159). Zürich, Switzerland: Springer-Verlag.

Pautasso, C. (2009b). On composing RESTful services. In F. Leymann, T. Shan, W.-J. van den Heuvel, & O. Zimmermann (Eds.), *Software service engineering.* Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. Available

from http://drops.dagstuhl.de/opus/volltexte/2009/2043

Pautasso., C. (2009). RESTful web service composition with BPEL for REST. *Data Knowl. Eng.*, *68*(9), 851-866.

Souza, A. R. R., Silva, B. L. B., Lins, F. A. A., Damasceno, J. C., Rosa, N. S., Maciel, P. R. M., et al. (2009). Incorporating security requirements into service composition: From modelling to execution. In L. Baresi, C.-H. Chi, & J. Suzuki (Eds.), *Icsoc/service-wave* (Vol. 5900, pp. 373–388). Available from http://dx.doi.org/10.1007/978-3-642-10383-4

Sun, J., Liu, Y., Dong, J. S., Pu, G., & Tan, T. H. (2010). Model-based methods for linking web service choreography and orchestration. In J. Han & T. D. Thu (Eds.), *Apsec* (pp. 166–175). IEEE Computer Society. Available from http://dx.doi.org/10.1109/APSEC.2010.28

Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). *Software architecture: Foundations, theory, and practice*. Wiley.

Vinoski, S. (2008, January). Serendipitous reuse. *IEEE Internet Computing*, *12*(1), 84-87.

Zeng, L., Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnanam, J., & Chang, H. (2004). QoS-aware middleware for web services composition. *IEEE Trans. Software Eng*, *30*(5), 311–327. Available from http://doi.ieeecomputersociety.org/10.1109/TSE.2004.11