Web Linking-Based Protocols for Guiding RESTful M2M Interaction

Jesus Bellido, Rosa Alarcon, and Cristian Sepulveda

Computer Science Department
Pontificia Universidad Catolica de Chile
`jbellido@uc.cl`, `ralarcon@ing.puc.cl`, `cmsepul@uc.cl`

Abstract. The *Representational State Transfer (REST)* style has become a popular approach for lightweight implementation of Web services, mainly because of relevant benefits such as massive scalability, high evolvability, and low coupling. It was designed considering the human-user as the one who drives service invocation and discovery. Attempts to provide machine-clients a similar autonomy have been proposed and recently, interesting discussion evaluate explicit semantics in the form of well-defined media types but introducing higher levels of coupling. We explore Web linking as a lightweight mechanism for representing *link* semantics and guiding machine-clients in the execution of well-defined choreographies and illustrate our approach with the OAuth and OpenId protocols exploring asynchrony and machine expectations as the interaction moves forward.

1 Introduction

The web has become a platform not only for the delivery of content, but also for the provision of services. Diverse functionality is made available to massive amount of users, and new services are built on top of others offering aggregated value. Popular service interfaces are generally classified into WSDL-based or REST based services, although other variants such as XML-RPC, Atom, JSON-RPC, etc. are also available[1]. In addition, the reuse of services into compounds (service composition) is highly desirable not only because it reduces costs and provides aggregated value, but also because it allows the creation of enriched applications, leveraging the Web as a services platform.

A REST service is a web of interconnected *resources* identified with *URIs*, that can be manipulated through a *uniform interface* (e.g. HTTP operations), whose *state* is served through *representations* (e.g. an HTML page) embedding *links* and *controls* (e.g. a `form` indicating a `POST` operation), which define the underlying hypermedia model that determines not only the relationships among resources but also the possible net of resource state transitions. REST consumers discover and decide which links/controls to follow/execute at *run-time*. This constraint is known as HATEOAS (Hypermedia As The Engine Of Application

[1] see `http://www.programmableweb.com/apis/directory`

State). A composition of REST services can be seen not only as the availability of new resources but also of new navigation paths (links/controls) that allow clients to traverse the hypermedia model corresponding to various independent REST services.

Providing support for automatic composition is also desirable since it may reduce development time and costs, but is far from trivial since REST services lack a standard machine-readable description, so that REST service providers describe their APIs in natural language (e.g. HTML pages) forcing machine-client developers to interpret the intended way of use of the API, to identify any change manually and to program the clients accordingly; most often, old APIs version are not supported. REST suppose humans as its principal consumer and they are expected to drive resource discovery and state transition by understanding the representation's content semantics, i.e. the links/controls embedded in representations such as HTML pages. The lack of explicit domain-level semantics in current media-types (e.g. HTML), makes harder for machine-clients to select, among the available links and controls, those they must follow in order to accomplish a specific navigation path or to engage in a predetermined way with various resources, as is the case for instance, of business processes, choreographies or authentication protocols. Some [1], propose the definition of domain specific media-types that portray the resource's state and the related hyperlinks. A machine-client that is aware of such custom media-types could then understand such representations and proceed accordingly. However, this requires that both client and servers agree on the media-types meaning, which introduces a strong coupling.

We are interested in exploring Web Linking [2] as a mechanism for specifying application-domain semantics for complex interaction such as business processes. In this paper we analyze the OAuth [3] and OpenID [4] protocols as case studies that implement Web choreographies, including control flow, asynchronous calls, out-of-band interactions and various media-types. The proposal allows a machine-client to understand resources' representation and to dynamically determine a navigation path, enacting the expected choreography. The paper is organized as follows, section 2 discuss related work, section 3 presents our approach, and finally section 4 present our conclusions.

2 Related Work

A few languages have been proposed to create machine readable RESTful services description. The Web Application Description Language (WADL) [5] describes RESTful services as *resources* identified by URI patterns, media types and the schemas of the expected *request* and *response* as well as *representations*. The latter supports *parameters* that can contain links to another resources. WADL, however, does not support link discovery or link generation for new resources, the resulting model is operation-centric and introduces additional complexity with unclear benefits for both human and machine-clients.

In [6], we proposed ReLL (*Resource Linking Language*), a hypermedia-centric REST service description. A ReLL description considers not only resources and representations, but fundamentally links and the mechanisms for identifying changes in the described REST service (e.g. changes in the URIs). ReLL allows machine-clients to retrieve, on run-time, links and state information embedded in representations so that a simple Web machine-client (a crawler) is able to traverse and discover the interlinked resources of a REST service. A ReLL description requires to annotate described resources and links/controls with *types*, serving as the basis for generating a semantic model. This approach made possible to semantically integrate independent REST services and execute queries that traverse the integrated web [7,8]. ReLL was used also as the basis for building machine-clients that traverse the Web enacting a predetermined workflow defined by a Petri Net[9]. The latter approach delegates on the Petri Net the responsibility of determining, at design time, the navigation path a machine-client must follow, resources, however, are dynamically bound. One of the main drawbacks of this approach is that, even though separation of concerns facilitates the design of workflows, it introduces coupling between the ReLL and Petri layers (horizontal interfaces [10]), so that changes on the ReLL description would make clients fail since the Petri Net is unaware of such changes.

Other approaches [1], avoid the need of a description by defining domain specific media-types (e.g. an XML schema for a company's bills) that portray the resource's state and the related hyperlinks. Authors define a *Domain Application Protocol (DAP)* as a collection of media types, URI entry points, HTTP idioms and the link relations portrayed in the representations. The DAP determines the set of legal interactions between a consumer and a set of resources involved in a business process and is also an implicit contract between the disparate parties in the composition, it is not clear though, how a machine-client may understand how to comply the DAP, unless both client and servers agree on the media-types meaning, which introduces a strong coupling.

In [11], Steiner and Algermissen acknowledge the limitations of relying on media-types to portray both content for human-consumption (that may require human-friendly formats such as HTML) and semantics directed to machine-clients (that may require RDF) and they propose content-negotiation (HTTP Options) to dynamically find out the appropriate media type. They propose also an extension of the RDF HTTP Vocabulary in order to become the media-type intended for machine-clients as well as the usage of links served as HTTP Headers annotated according to the Web linking standard [2].

The standard specifies *relation types* for Web links, defines a registry for them, and regulates its usage in HTTP headers (Link headers). A *link* is a typed connection between two resources, that involves a context URI (the origin resource URI), a link relation type, a target IRI, and optionally, target attributes. No restrictions are placed on cardinality or relative ordering of the links. Target *attributes* are key/value pairs that further describe the link or its target (e.g. `media="text"`). A *link relation type* identifies the semantics of a link (e.g. `rel="copyright"`) and there are two kind of relation types, registered and extension. The former are well-defined, registered tokens; while extension relation types are URIs that uniquely identify the relation type. Steiner [11] relies on registered tokens and media-types that lack domain-level semantics which introduces less coupling but makes impossible for a machine-client to make sense of the presented information and decide which link to follow and which information may be relevant for such decision. In addition the proposed media-type do not allow to dynamically discover links and controls to related resources.

3 REST Services Composition and Interaction Protocols

We are interested in the development of machine-clients that enable service composition involving REST services. Statelessness is a key REST constraint that dictates not to store the state of the interaction between clients and servers on the server side. This constraint has two consequences, stateless servers are much less complex than stateful ones, providing massive levels of scalability and fault tolerance (e.g. hardware replicas); but this also requires that each request to the server must contain all the information needed to provide a response. Service composition has traditionally focused on stateful approaches where a central component orchestrates the dialogue between the parties and store all the necessary information to move forward the interaction.

A stateless, RESTful scenario where there is no such orchestrator but a cooperation of the involved resources, that is a choreography, requires that the representations served to each other mediate the interaction. The HATEOAS constraint is fundamental in this scenario, provided that machine-clients can understand the semantics of the links and controls served in the representations, and they have the required semantics to move forward the interaction.

We could argue that at a very general level, Web linking registered relation types such as `start`, `previous`, `next`, `first`, `last` [2], could be used to embed instructions within the served representations and add basic semantics to guide resources interaction. However, interaction have explicit semantics in particular domains that can be exploited for servers to steer machine-clients. For instance, let's consider the REST APIs implementing the OAuth and OpenId protocols; callbacks and redirection are part of the interaction; they implement an interrupted, asynchronous conversation where third parties (out-of-band) later affect resources' state and dynamically generate pieces of information that are expected to be carried out at various steps of the interaction.

3.1 Security Domain: OAuth 2.0 and OpenId

Modeling non functional aspects of services have captured the attention of researchers as a medium for enriching and constraining automatic compositions and one of these aspects is security. In [12], a survey determines that most Web APIs use one of five authentication mechanisms, namely, they use credentials (API key or username and password) to restrict access to a service, Web authentication protocols (HTTP Basic Authentication, HTTP Digest Authentication and OAuth), or even ad-hoc authentication mechanisms (parts of the HTTP request). OAuth accounts for a mere 6% of the APIs surveyed, however recent adoption of stronger security capabilities such as OAuth and HTTPS for mayor players in the industry (e.g. Facebook, Twitter) will have an influence on applications developed on top of these platforms.

OAuth 2.0. The OAuth 2.0 authorization protocol allows to grant third-party applications limited access to an HTTP service on behalf of a user, by orchestrating an approval interaction protocol between the user and the HTTP service. 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. Each grant type defines an authorization interaction flow between four parties, the *client*, the *resource owner*, the *authorization server* and the *resource server*.

The authorization code grant type flow is illustrated in Figure 1. The client obtain some credentials (1, 2) and requests authorization from the resource owner directly, or preferably through an authorization server (A). The server authenticates the resource owner through a user-agent (e.g. a form displayed in a Web browser). This communication occurs out-of-band between the Resource Owner (e.g. LinkedIn) and the user. Once the resource owner grants access to the required resources, the authorization server redirects the user-agent to the callback and includes an authorization code provided to the client (C). The authorization code is used to request an access token (D) from the authentication server, once the token is granted (E), the client application can use it to access resources stored in the resource server (F, G).

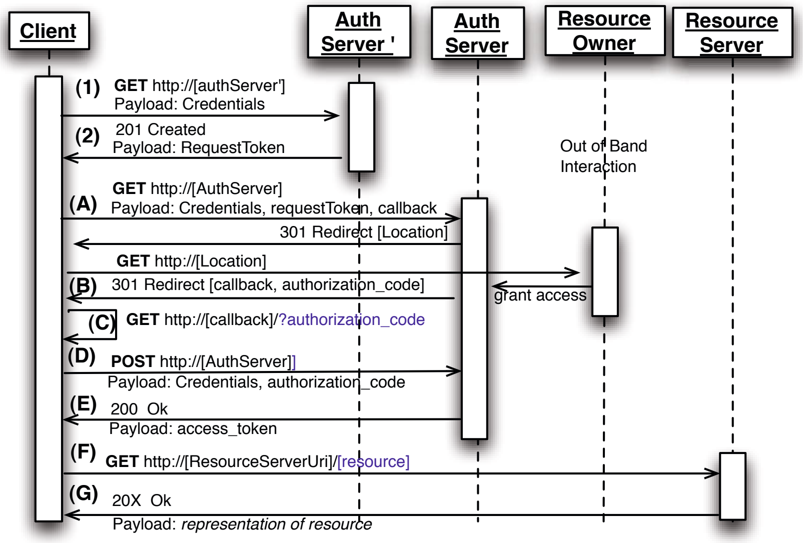
Client
Auth Server '
Auth Server
Resource Owner
Resource Server
(1) GET http://[authServer']
Payload: Credentials
(2) 201 Created
Payload: RequestToken
Out of Band Interaction
GET http://[AuthServer]
(A) Payload: Credentials, requestToken, callback
301 Redirect [Location]
GET http://[Location]
(B) 301 Redirect [callback, authorization_code]
grant access
(C) GET http://[callback]/?authorization_code
(D) POST http://[AuthServer]]
Payload: Credentials, authorization_code
(E) 200 Ok
Payload: access_token
(F) GET http://[ResourceServerUri]/[resource]
(G) 20X Ok
Payload: representation of resource

Fig. 1. OAuth 2.0 Abstract protocol sequence diagram

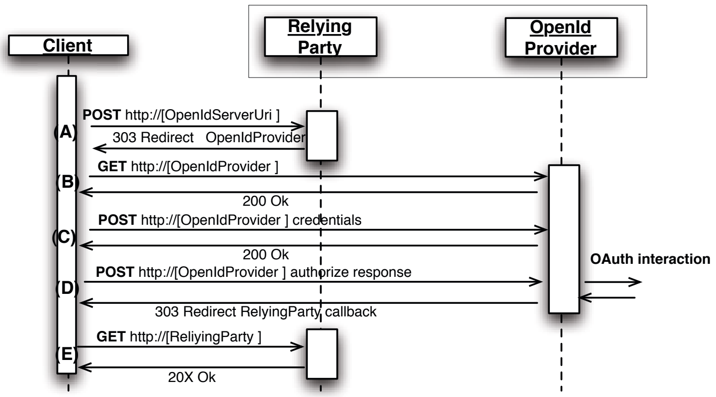
Client
Relying Party
OpenId Provider
(A)
POST http://[OpenIdServerUri]
303 Redirect OpenIdProvider
(B)
GET http://[OpenIdProvider]
200 Ok
(C)
POST http://[OpenIdProvider] credentials
200 Ok
(D)
POST http://[OpenIdProvider] authorize response
303 Redirect RelyingParty callback
OAuth interaction
(E)
GET http://[ReliyingParty]
20X Ok

Fig. 2. OpenID abstract protocol sequence diagram

OpenID. The OpenID protocol allows consumers to present, to a service, claims about their identity that have been authenticated by an identity provider trusted by that service. OpenID allows a service to delegate the responsability for storing consumer credentials to one or more OpenID providers. The providers are responsible for checking a consumers credentials and informing a service if an identity claim is valid.

Figure 2 shows the protocol for a client to register an OpenID URI that they claim to own to a Relying Party (e.g. LiveJournal). In an Initiation step (A), the Party redirects the client to the proper OpenId Provider (e.g. Blogger) (B) that requires the user to provide both, credentials (user, password) and optionally to choose a preferred authentication server (C). With that information, the OpenID provider validates user credentials and if necessary may redirect the client to the appropriate OpenId provider (e.g. Google, PayPal, Yahoo, etc.), this in turn verifies the consumer credentials and confirms the registry of the user OpenID, otherwise, it redirects the consumer to the OAuth server in order to grant access to identity information (e.g. Google, Facebook, etc.).

3.2 Linking Requirements: Modeling Stateless Choreographies

In order to design stateless interaction, state (client-server interaction) must be explicitly modeled either as a different resource [10], as cookies, or be embedded in the representations so that clients can build later the subsequent requests properly. As seen in Figure 3, the latter approach can be accomplished without requiring extensive changes in the representations by exploiting Web Linking. The text in italics (red) shows our proposal for Link Headers, the URI part represents the URI of the resource to be retrieved. The semantics of the link are explicitly presented by the `rel` parameter as an extension relation type (an abbreviated URI in our case) that refers to a particular realm, process or application domain, a target attribute identify the *expected* state that can be achieved

```
(0) Link:  <https://api.linkedin.com/uas/oauth/requestToken>; rel="oauth:start";
           state="[oauth:started | oauth:denied]"; method="GET"

(1) GET https://api.linkedin.com/uas/oauth/requestToken
    Authorization=OAuth
    oauth_consumer_key= ...,
    oauth_nonce="180098101",
    oauth_timestamp="1284497324",
    oauth_signature=...,
    oauth_callback="oob",
    oauth_signature_method="HMAC-SHA1",
    oauth_version="1.0"

(2) HTTP/1.1 201 Created
    Content-Length=236, null=HTTP/1.1 201 Created, Date=Tue, 14 Sep 2010 20:52:18 GMT,
    Content-Type=text/plain, Server=Apache-Coyote/1.1
    Link:  <https://api.linkedin.com/uas/oauth/authorize>; rel="oauth:grant";
           state="[oauth:granted | oauth:denied]"; method="GET";
    Params:
    oauth_token=142e1172-aca0-40e8-9a3f-163f52969cda
    oauth_token_secret=b795c3ae-bf72-4451-baaf-eb31b6b024e1
    oauth_callback_confirmed=true
    oauth_request_auth_url=https://api.linkedin.com/uas/oauth/authorize
    oauth_expires_in=599

(A) GET https://api.linkedin.com/uas/oauth/authorize?oauth_token=142e1172-
    aca0-40e8-9a3f-163f52969cda

(C) HTTP/1.1 200 OK
    Link:  <https://api.linkedin.com/uas/oauth/accessToken>; rel="oauth:accessToken";
           state="[oauth:authorized | oauth:unauthorized]"; method="POST";
    https://www.linkedin.com/uas/oauth/authorize/oob?
    oauth_token=4be35e7e-9d5b-4cb9-82fa-3dfd6b694fdc

(D) POST https://api.linkedin.com/uas/oauth/accessToken
    Authorization=OAuth
    Params:
    oauth_consumer_key="..."
    oauth_nonce="-46807422"
    oauth_timestamp="1284754819"
    oauth_signature="UCgAG4ueyGRcSZluUwz8dhOYCOk%3D"
    oauth_verifier="92577"
    oauth_callback="oob"
    oauth_signature_method="HMAC-SHA1"
    oauth_token="ae98a651-36a0-41c8-ab24-e2a2e1672bcb"
    oauth_version="1.0"
```

Fig. 3. Messages exchanged during OAuth 2.0 protocol, for a LinkedIn implementation

by the machine-client if the link is followed, as well as the method to be performed. Control flow operators that are common in service composition such as conditional invocation, selection of the best result, parallel execution, etc. [13], should be also considered. We model such controls as XPath expressions (`operators`) that are evaluated at run-time with the assistance of a ReLL description. For the OAuth case, the choreography starts with a first link (Figure 3.0). For a more general case, such as a business process, this will indicate that a process initiates a subtask at a particular entry-point (which can be dynamically discovered). The request (1) to the URI changes the state of the `oauth:start` resource. The new served response (2) is processed by the machine-client using ReLL as a means to derive some hints from the content.

For instance, it verifies that the URI matches clients' expectancy (no changes in the URI); it retrieves the representation expecting to be encoded as *text/plain*, and verifies whether a regular expression is contained. In such case, it determines wether the expected state `oauth:started` (a 201 HTTP code indicates, in the LinkedIn implementation, that a new Request Token was *created*) was achieved and prepares to discover the next step of the orchestration. The link `request grant` is followed in order to obtain an `oauth:grant` resource, the link is retrieved from the representation by executing a `select` expression that can be encoded as a regular expression or as an XPath expression depending on the content media type. Since Web Linking determines rules to transform links to XML, we decided to transform the link to XML and use XPath expressions for illustrating the dependance on the media-type. The method can be also retrieved from the content. Once the link is followed (A), the new served response includes instructions to force a redirection on the machine-client which will lose control of the interaction, and could either wait for an asynchronous message to regain control, look for an answer later, or stop its execution and trust that the next message will contain the necessary information for resuming the interaction without losing information. We implemented the latter alternative in the machine-client by sending all which is necessary to continue the interaction, since the OAuth protocol allows for an `extra` parameter for such kind of purposes. Eventually, the interaction is resumed by a message received through a callback (`http://darwin.ing.puc.cl ...`). Figure 4 presents a snippet of a ReLL description for an Linkedin OAuth implementation shown in figure 3.

```
<resource xml:id="oauth:start">
    <uri match="https://api.linkedin.com/uas/oauth/requestToken" type="regex"/>
    <representation xml:id="requestToken-text" type="iana:text/plain">
        <name>oauth_token request parameters</name>
         <state name="oauth:started" select="HTTP\/1\.1\ 201" type="regex"/>
         <link type="request_grant" target="oauth:grant" minOccurs="0" maxOccurs="1">
           <selector name="href"    select="//Link/@href" type="xpath"/>
           <selector name="state" select="//Link/@state" type="xpath"/>
          <protocol type="http">
                <request>
                   <selector name="method" select="//Link/@method" type="xpath"/>
                </request>
                <response media="iana:text/plain"/>
           </protocol>
        </link>
    </representation>
</resource>
<resource xml:id="oauth:grant">
     <uri match="http://darwin.ing.puc.cl\?oauth_token=[a-zA-Z0-9\-]*" type="regex"/>
      ...
 </resource>
```

Fig. 4. ReLL snippet describing the RequestToken resource according to LinkedIn implementation (step 1 in Figure 2)

On run-time and starting from a seed, a machine-client retrieves a resource (e.g. an HTML form indicating that the user must authenticate by clicking a button), such page (e.g. `https://api.linkedin.com/uas/oauth/requestToken`) is described as an `oauth:start` resource, and the corresponding ReLL declarations are applied; that is, the XPath expressions or selectors, retrieve both state variables (`state`), and `links`. Since a control flow operator is omitted, the `request_grant` link determines that the next link to be retrieved corresponds to the `oauth:grant` category (`target`). State variables are carried along and stored by the machine-client.

It is also possible to dynamically generate new Links from the state variables and (part of ReLL dynamic late binding characteristics) add cardinality constraints for links. In Figure 2, step D indicates a REST composition of both OpenId service (e.g. Blogger's OpenId) and OAuth service (e.g. LinkedIn's OAuth). Again, this interaction is triggered by the OpenId Provider sending a GET message to LinkedIn in order to access a resource. The message is directed to the resource URI and a state variable (security_token) is sent in the body of the message. If valid, LinkedIn will confirm user authorization, if not, the user will follow OAuth from step 1.

3.3 Coupling Facets in Our Approach

One of the risks of supporting REST services descriptions is the increasing of coupling between clients and servers. Coupling has been described as a multidimensional property [10], where dimensions or facets include relevant design aspects that determine the degree of coupling in a system. In our approach ReLL serves as an abstraction layer between RESTful services and a machine-client. According to the defined coupling facets, ReLL will not increase the coupling degree between RESTful Web services and machine-clients as detailed below:

– *Discovery.* RESTful web services can be discovered by decentralized referrals exchanging hyperlinks. Services are not registered in any standardized way (e.g. UDDI). ReLL allows a machine client to discover resources by following the links encountered in resource representations. ReLL's `select` expression allows the machine client to retrieve embedded links, and `generate-uri` allows to dynamically mint new URIs from expressions embedded in a resource representation. The latter feature allows a designer to compensate the lack of hypermedia on current RESTful APIs.
– *Identification.* A URI globally identifies RESTful web services, URIs however are not constrained only to the `http` scheme. URIs identify services in different contexts and services are free to use different identifications schemes. ReLL allows a machine-client to follow a link that leads to discover another URI under any scheme (i.e. any protocol).
– *Binding.* Dynamic binding resolves at run-time the URI to be invoked, the binding is established only when it becomes necessary. ReLL allows a machine client to follow and resolve, at run-time, the URI of the link encountered in a representation obtained as a result of an invocation.

- *Platform.* Platform independency requires that services built using different and heterogeneous platforms can interact with each other without a bridge. ReLL describes resources using XML in a platform independent way. XPath expressions are also well known and standard.
- *Interaction.* Asynchronous interaction allows two services to interact without being available at the same time. HTTP is a synchronous communication protocol, and ReLL supports both synchronous scenarios and asynchronous scenarios which, in the Web context, requires to perform callbacks to a URI. This is not trivial, since servers (instead of clients) redirect user-agents to the callback URI which causes machine-clients a loss of context of the previous interaction. For the case of REST, this is solved by carrying out context information along the interchanged messages, hence, no state of the interaction is stored on the served side (session). ReLL supports this feature by injecting links and context information (through the `generate-uri` expression) that restores the interaction sequence obtained from the resource representations only when it is required.
- *Interface Orientation.* Vertical interfaces rely on using protocols for allowing components to directly communicate between them, horizontal interfaces or layered architectural styles introduce a stronger dependency among the layers which makes the architecture more coupled. ReLL relies on protocols to allow client machines to interact with services, protocols are described in terms of methods to be invoked and the media-types to be expected when a link is followed. Our previous version, used a Petri Net layer that introduced a stronger coupling between the ReLL description and the Petri Net making it hard to support server evolution without breaking the machine-client.
- *Model.* Self describing messages do not require to share a model for marshaling and unmarshaling messages. ReLL do not require any particular message format (i.e. a canonical media-type), instead it allows a client machine to recover information from representations by using XPath or regular expressions. However, if the server changes the representations, the machine-client will fail to retrieve information embedded in the content and proceed with its intended interaction. By using Web Linking however, servers can change arbitrarily the URLs of the resources involved in the choreography without causing machine-clients to break.
- *State.* Stateless services keep the state in the messages that are passed between cooperating services instead of storing the client-server interaction on the server side. ReLL discovers a cooperating service URI and its parameters from the served resource representation, and then, it generates the link to be invoked as well as the protocol and method. That is, it assumes that the message contains the state information, and it is even capable of extracting part of the message and mint context and links.
- *Generated Code.* A service description can be used to generate code, automatically (stub), that represents the service facade, either on the server or the client side. It introduces a strong contract between clients and servers and hence strong coupling. Code generation only works if the communications requirements are completely specified in a machine-readable form. It is not

possible to generate code from a ReLL service description, because it does not have a full detail of the URIs (i.e. do not register all the available URIs, nor a URI pattern), nor a full detail of the representations (i.e. it annotates the expected media-type and expected patterns in the representations, but not the content itself). A ReLL document is a partial, arbitrary (since it represents a particular client view of the service) description of a REST service. Such description expresses the expectations of a generic machine-client when interacting with a REST service but do not force servers to comply with the description, allowing then servers to freely evolve.
– *Conversation.* A reflective inspection mechanism enables clients to interact with the service by inquiring it about the possible future steps of the interaction. In our proposal, servers have full control of the links and representation served and can change them at any time, ReLL descriptions represent the expectancies of a machine-client but do not constraint in any way the server actions, instead, it contains the mechanisms (select expressions) to discover on run-time the hyperlinks. By enriching the links with Web Linking features (`rel` and target expressions such as `state`), the server explicitly indicates to its clients what state could be achieved when following a link.

4 Conclusions

A set of media-types determined a priori (e.g. XML) allow machine-clients to make sense of the contents and proceed accordingly, however, application-domain media types evolve continuously, sometimes media-types with no support for links or structure (e.g. binary) are required, and furthermore, they require an agreement between clients and servers, introducing stronger coupling and limiting service evolvability. By relying in well formed REST representations that fully support the HATEOAS constraint, it is possible for a machine-client to pursue a series of operations that transform the resource state.

Service descriptions (e.g. ReLL), increase coupling between clients and servers but in less degree. ReLL allows machine-client designers to encode rules and assumptions for the understanding and processing of the resources without limiting service evolvability. It facilitates to detect whether some assumptions have changed (e.g. more links than expected are served, the URIs have changed, the protocol have changed etc.), and take a proper action. Web Linking relations can be formally described as vocabularies with well-defined semantics so that machine-clients can make complex assumptions and derive plans dynamically. As for future work, we are interested in the definition of business processes enabled by lightweight infrastructures that steer a machine-client dynamically through the underlying hypermedia, so a vocabulary for Web Linking that extends relation types for Business Processes will be our next endeavor. This goal is quite challenging because it requires also to deal with conversation state [10], but also with user interaction, events, complex control flow and complex information transformation. OAuth and OpenId features such as asynchronous communication, callbacks, and state handling shed some lights for facing events and user interaction as out-of-band communication, delegating control on third parties and resuming later the navigation provided that state can be carried along the interaction.

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

References

1. Webber, J., Parastatidis, S., Robinson, I.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly & Associates, Sebastopol (2010)
2. Nottingham, M.: Web linking. Internet RFC 5988 (October 2010)
3. Barnes, R., Lepinski, M.: The oauth security model for delegated authorization. Internet Draft draft-barnes-oauth-model-01 (2009)
4. Recordon, D., Reed, D.: Openid 2.0: a platform for user-centric identity management. In: Juels, A., Winslett, M., Goto, A. (eds.) Digital Identity Management, pp. 11–16. ACM (2006)
5. Hadley, M.: Web application description language. World Wide Web Consortium, Member Submission SUBM-wadl-20090831 (August 2009)
6. Alarcón, R., Wilde, E.: Restler: Crawling restful services. In: Rappa, M., Jones, P., Freire, J., Chakrabarti, S. (eds.) 19th International World Wide Web Conference, pp. 1051–1052. ACM Press, Raleigh (2010)
7. Alarcon, R., Wilde, E.: Linking data from restful services. In: Third Workshop on Linked Data on the Web, Raleigh, North Carolina (April 2010)
8. Alarcón, R., Wilde, E.: From restful services to rdf: Connecting the web and the semantic web. School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2010-041 (June 2010)
9. Alarcón, R., Wilde, E., Bellido, J.: Hypermedia-driven restful service composition. In: Feuerlicht, G., Lamersdorf, W., Ortiz, G., Zirpins, C. (eds.) 6th Workshop on Engineering Service-Oriented Applications (WESOA 2010), San Francisco, California (December 2010)
10. Pautasso, C., Wilde, E.: Why is the web loosely coupled?: a multi-faceted metric for service design. In: Proceedings of the 18th International Conference on World Wide Web, WWW 2009, pp. 911–920. ACM, New York (2009), `http://doi.acm.org/10.1145/1526709.1526832`
11. Steiner, T., Algermissen, J.: Fulfilling the hypermedia constraint via http options, the http vocabulary in rdf, and link headers. In: Pautasso, C., Wilde, E., Alarcón, R. (eds.) Second International Workshop on RESTful Design (WS-REST 2011), pp. 11–14 (March 2011)
12. Maleshkova, M., Pedrinaci, C., Domingue, J., Alvaro, G., Martinez, I.: Using Semantics for Automating the Authentication of Web APIs. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 534–549. Springer, Heidelberg (2010)
13. Hamadi, R., Benatallah, B.: A petri net-based model for web service composition. In: Schewe, K.-D., Zhou, X. (eds.) Fourteenth Australasian Database Conference (ADC 2003), CRPIT, vol. 17, pp. 191–200. ACS, Adelaide (2003), `http://crpit.com/confpapers/CRPITV17Hamadi.pdf`