REST

Representational State Transfer

Michael Jakl
mj@int-x.org
0226072 – 033-534

University of Technology Vienna

Abstract. Many modern Web-services ignore existing Web-standards and develop their own interfaces to publish their services. This reduces interoperability and increases network latency, which in turn reduces scalability of the service. The Web grew from a few thousand requests per day to million requests per hour without significant loss of performance. Applying the same architecture underlying the modern Web to Web-services could improve existing and forthcoming applications. REST is the idealized model of the interactions within an Web-application and became the foundation of the modern Web-architecture, it has been designed to meet the needs of Internet-scale distributed hypermedia systems by emphasizing scalability, generality of interfaces, independent deployment and allowing intermediary components to reduce network latency.

1 Introduction

In order to build highly interoperable Web-applications we cannot ignore standards. The Web – as the biggest Web-application available – is not defined by a particular implementation, it is defined by its standard interfaces and protocols.

Interoperability leads to increased usage, so scalability is another big issue within Internet-scale distributed systems. The Web has scaled from a few thousand requests per day to millions of requests per hour, so relying on established standards seem to be viable for forthcoming Web-services.

Several standards can be used to address the same problem domains, each with its own strengths and weaknesses, so relying on standards is only one half of the story. We should also use guidelines (or architectural styles) to support our quest of developing interoperable Web-services.

The industry has developed a set of competing ways to enable developers to build Web-services as easy as possible. Many of these rely on a Remote Procedure Calls (RPC) to pass around the messages between involved services. Using RPC-like models for Web-services can lead to severe drawbacks in terms of performance and scalability.

Modern Web-applications should support independent deployment, we cannot count on a Big-Bang deployment where all components are deployed at the same time, especially services crossing the organizational boundaries have to be prepared for gradual and frequent change.

The structure of this article is as follows:

- First we will develop a basic understanding of the REST architectural style – the foundation of the modern Web-architecture.
- Building on the basics we will design a small example Web-application to deepen the understanding how independent components can be orchestrated to form an interoperable and scalable Web-application.
- In the last chapter we will compare REST to RPC-like protocols. The Simple Object Access Protocol (SOAP) will be the RPC-like example. SOAP itself is more a protocol building framework than a protocol, but it is promoted by leading companies as the only technology for implementing Web-services.

2 Representational State Transfer

The architectural style underlying the Web is called *Representational State Transfer* or simply *REST*. REST answered the need of the Internet Engineering Task Force (IETF) for a model how the Web *should* work. It is an idealized model of the interactions within an overall Web-application.

> Roy T. Fielding defines REST in [6] as an coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.

2.1 Architectural elements

REST distinguishes three classes of architectural elements:

- data elements,
- connecting elements (connectors) and
- processing elements (components).

Data elements The key aspect of REST is the *state* of the data elements, its components communicate by transferring representations of the current or desired state of data elements.

Resource The key abstraction in REST is the *resource*. Anything that can be named can be a resource. A resource is a conceptual mapping to a set of entities, only the semantics of a resource are required to be static, entities may change over time. This means the entity behind a resource may change over time. For example in an version control system the `Version 1.2` always points to the same entity whereas `HEAD`s entity changes over time.

This concept allows an author to reference a concept instead of a single representation.

REST uses *resource identifiers* to distinguish between resources. In a Web-environment the identifier would be an Uniform Resource Identifier (URI) as defined in the Internet RFC 2396 [1].

Representation All REST components perform actions on representations of resources. Representations capture the current or the intended state of a resource and can be transferred between components.

A representation consists of

- a sequence of bytes (the content)
- representation metadata (describing the content)
- metadata describing the metadata (e.g. hash sums or Cache-Control)

REST is not tied to a specific data format as long as all components can process the data. A intermediary cache, for example, does not need to know the semantics of the data, only if the request or response is cacheable or not.

The data format of a representation is called *media type*. Some media types can be processed by computers, others are intended primarily

for the human reader and few can be automatically processed and viewed by a human reader. The design of a media type can directly effect the performance of the system. Any data that must be received before the processing of the request can begin adds to the latency of the interaction. A media type with the important data at the beginning of the stream can be processed while it is received, and thereby lowers the overall time for answering the request. For example a Web-browser which is capable of rendering the page while receiving it provides a better user perceived performance than a browser without this ability.

In modern Web-services a commonly used representation is XML (Extensible Markup Language) as defined by the World Wide Web Consortium (W3C) in [11], other representations suitable only for the human reader like HTML, JPEG and PDF are also very common. Metadata like Cache-Control, last-modified time and media-type can be used to dynamically choose among different possible representations and improve scalability by explicitly define cacheable representations.

Connectors Connectors manage the network communication for a component.

The connectors present an general abstract interface for component communication leading to

- separation of concerns,
- hiding the underlying implementation,
- enhancing simplicity and
- substitutability.

Every REST interaction is stateless, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it.

All these constraints lead to following advantages:

- no need for the connector to retain application state, leading to simplified and scalable components
- requests can be processed in parallel because no interaction semantics have to be understood
- requests can be understood in isolation leading to simplified orchestration and dynamic service rearrangement
- enables caching

REST encapsulates different activities of accessing and transferring representations into different *connector types*:

- client - sending requests, receiving responses
- server - listening for requests, sending responses
- cache - can be located at the client or server connector to save cacheable responses, can also be shared between several clients
- resolver - transforms resource identifiers into network addresses
- tunnel - relays requests, any component can switch from active behavior to a tunnel behavior

A component can implement more than one connector type.

Examples for connector types used in modern Web-services are *libwww* as client and server connector, any *browser cache*, or the *Akamai cache network* as cache connector, *bind* as resolver connector (only the library) and a *SOCKS* proxy as tunnel connector.

Components REST Components are identified by their role within an application.

User agent uses a client connector to initiate a request and becomes the ultimate recipient of the response.

Origin server uses a server connector to receive the request, and is the definitive source for representations of its resources. Each server provides a generic interface to its services as a resource hierarchy.

Intermediary components act as both, client and server in order to forward – with possible translation – requests and responses.

Example components are *Apache httpd* as origin server, any Web-browser as user agent and *Squid* as Proxy or Gateway.

2.2 Architecture

REST defines the architecture by placing restrictions onto component interactions instead of predefining a particular component topology. It ignores the implementation details and protocol syntax to achieve

- scalability of component interactions,
- independent deployment of components and to
- allow intermediary components to reduce interaction latency.

The components of a REST architecture can be dynamically rearranged, intermediaries can be placed into the flow of a representation and act similar to a pipe and filter style. The stateless nature of REST interactions allows each request to be independent of the others, removing the need for an awareness of the overall component topology, an impossible task in an Internet-scale distributed system.

REST is not tied to a particular protocol, it provides seamless integration to different protocols by allowing intermediaries to transform

representations on their way through the system, although HTTP is the most widely used protocol in the Internet.

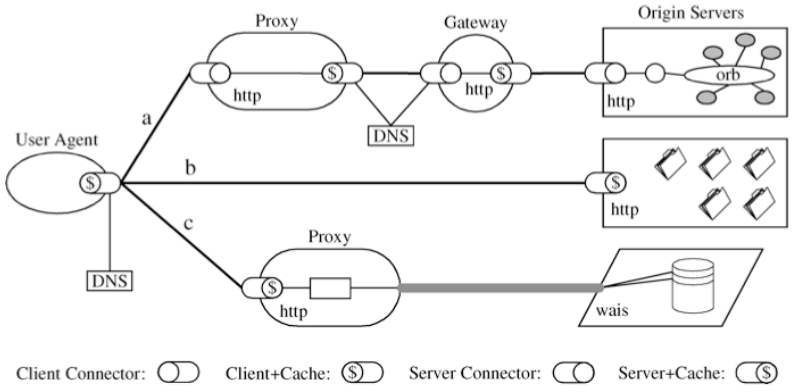

Fig. 1. Example architecture (Picture taken from [4]). Here a user agent has three running interactions (a, b, c) with different intermediaries and different origin servers. Interaction c uses a proxy to translate the HTTP message into a WAIS[2] request.

RESTs primary goal is to allow Internet-scale distributed hypermedia systems by reducing network latency with caching and reducing server load by omitting session states. Thus a user-agent has to store the state, providing the possibility to directly manipulate the state. For example the Web-browser history or bookmarks will be valid unless the underlying concept is removed, in a state-aware application history and bookmarks are often broken.

2.3 Orchestration

REST Web-service orchestration is not a big issue since the protocols are already standardized. Only the data structures and URIs of the involved services have to be known.

We may use every script language with decent HTTP support. Even modern XSL processors can be used to combine existing Web-services.

3 Example REST Web-service

To deepen our understanding of the REST architectural style we will develop a sample Web-service. The application will be small to remain educational.

The service will provide this functionality:

- the user can upload a picture
- metadata can be attached to pictures
- pictures and attached metadata can be deleted
- a list of pictures, the picture and the metadata of a picture can be retrieved

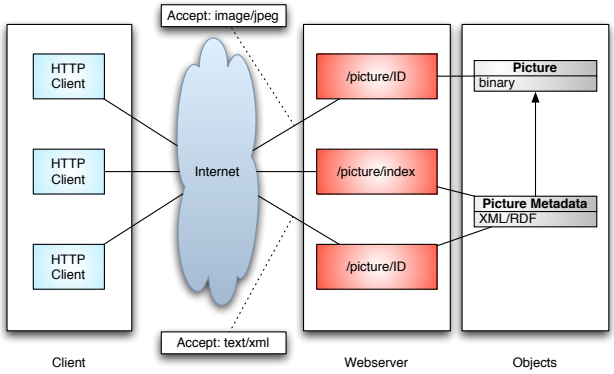

Fig. 2. Overview of the Web-service demo application. Note that the interface should not expose how the objects are saved! The "Object" layer could consist of flat files as well as a database.

3.1 Resources

The key abstraction in REST is the resource, so we will begin by identifying the resources within our application.

- Picture
- Picture-Collection

3.2 Representations

Each resource has an associated representation:

- Picture: binary and XML
- Picture-Collection: XML

3.3 Addressing

The resources in our case are addressable via an URI. Only resources can be addressed, not the representations, so we had to split the metadata from the picture itself.

- Picture: `/picture/ID`
- Collection: `/picture/index`

Instead of explicitly address the metadata and the binary representation of the resource, the client could use *content negotiation* to determine which representation should be returned. `Accept: text/xml` in the HTTP header of the request would get the XML representation of the resource, this is also the default, and `Accept: image/jpeg` requests the binary representation.

3.4 Methods

HTTP[3] defines a set of methods, in our example we will use GET, PUT, POST and DELETE. Additional methods like HEADER and OPTIONS can be used to employ improved caching (HEADER), and return a service description (explanation of available methods etc.) of the given URI (OPTIONS).

PUT is used to upload a new picture to the server. It requires the user to authenticate himself via HTTP AUTH[7]. The PUT request returns the *201 Created* response code and an URI of the created resource. PUT can be carried out on `/picture`.

Important: The definition of HTTP [3, 9.1.2] requires that all methods excluding POST are idempotent, so the generated ID for the picture has to be stable! For example a hash sum of the content or something similar should be used.

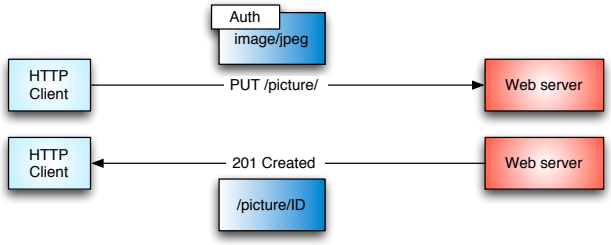

Fig. 3. Example of PUT usage

Request

```
PUT /picture HTTP/1.1
Authorization: Basic dGVzdDp0ZXN0MQ==
Host: localhost:2000
```

```
Content-Length: 13077

<binary-data...>
```

Response

```
HTTP/1.1 201 Created
Connection: Keep-Alive
Date: Wed, 23 Feb 2005 12:18:23 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
Content-Length: 44

http://localhost:2000/picture/11091611039546
```

POST is used to append more metadata to the addressed resource. For example GPS data like longitude and latitude can easily be appended to the resource. POST can be carried out on `/picture/ID`. It returns the updated representation of the picture-metadata, and requires authentication.

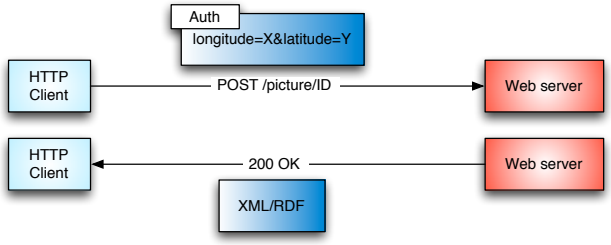

Fig. 4. Example of POST usage

Request

```
POST /picture/11091611039546 HTTP/1.1
Authorization: Basic dGVzdDp0ZXN0MQ==
Host: localhost:2000
Content-Length: 33
Content-Type: application/x-www-form-urlencoded

longitude=16.3221&latitude=48.199
```

Response

```
HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Wed, 23 Feb 2005 12:24:14 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
Content-Length: 671

<?xml...>
```

DELETE can be used to delete a resource. It can be applied to `/picture/ID`, and requires authentication. If the server accepts the request, the response status code is *202 Accepted*.

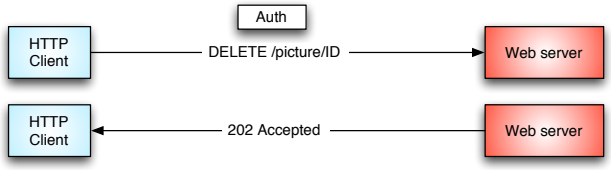

Fig. 5. Example of DELETE usage

Request/response

```
DELETE /picture/11091611039546 HTTP/1.1
```

```
Authorization: Basic dGVzdDp0ZXN0MQ==
Host: localhost:2000

HTTP/1.1 202 Accepted
Connection: Keep-Alive
Date: Wed, 23 Feb 2005 12:28:49 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
Content-Length: 2

OK
```

GET is used to retrieve a representation of the specified resource. It does not require authentication and can be applied to `/picture/index` to get a list of available pictures, GET on `/picture/ID` with `Accept: text/xml` to get the picture-metadata and `Accept: image/jpeg` to get the binary representation.

Request/response for the metadata

```
GET /picture/11091611039546 HTTP/1.1
Host: localhost:2000
Accept: text/xml

HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Wed, 23 Feb 2005 11:56:31 GMT
Content-Type: text/xml
Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
Content-Length: 576

<?xml ...?>
```

Request/response for the binary representation

```
GET /picture/11089954788757 HTTP/1.1
```

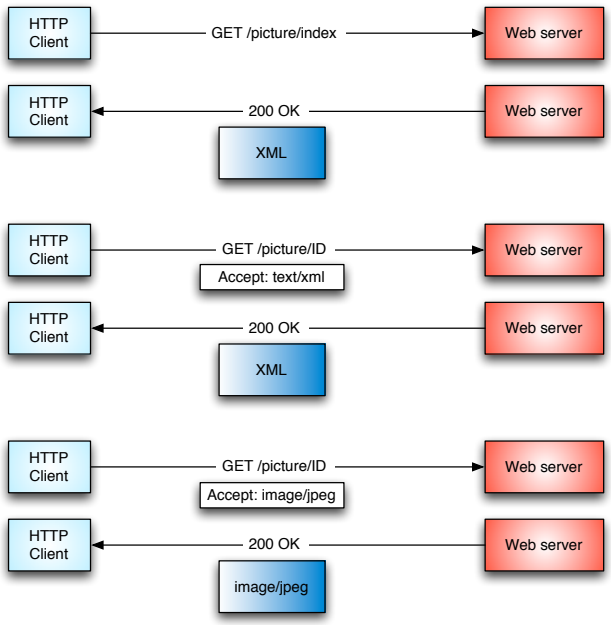

Fig. 6. Examples of GET usage

```
Host: localhost:2000
Accept: image/jpeg

HTTP/1.1 200 OK
Connection: Keep-Alive
Date: Wed, 23 Feb 2005 11:56:50 GMT
Content-Type: image/jpeg
Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
Content-Length: 19516

<binary data>
```

OPTIONS could be used to return a service description of the accessed service. Such a service description is yet to be specified.

Here an example of a service description for the different URIs used in the application.

```
<resource name="collection" uri="/picture/">
    <PUT>
        <returns type="text/plain" name="uri">
            <body name="image" type="image/jpeg">
                Binary representation of the picture to be uploaded.
            </body>
        </returns>
    </PUT>
</resource>

<resource name="picture" uri="/picture/{ID}">
    <variable name="ID" type="integer">
        ID of the referenced resource.
    </variable>
    <GET>
        <returns type="text/xml">
            <input>
                <header name="Accept" value="text/xml">
                    Defines the representation
                </header>
            </input>
        </returns>
        <returns type="image/jpeg">
            <input>
                <header name="Accept" value="image/jpeg">
                    Defines the representation
                </header>
            </input>
        </returns>
    </GET>
    <POST>
        <returns type="text/xml">
            <input>
                <query name="longitude" type="float">
                    Specifies the longitude of an geographic location.
                    Should denote the place, where the picture was taken.
                </query>
                <query name="latitude" type="float">
                    Specifies the latitude of an geographic location.
                    Should denote the place, where the picture was taken.
                </query>
            </input>
        </returns>
    </POST>
</resource>

<resource name="list" uri="/picture/index">
    <GET>
        <returns type="text/xml" name="metadata">
        </returns>
    </GET>
</resource>
```

4 SOAP vs. REST

In order to compare REST and the Simple Object Access Protocol (SOAP) it is useful to examine different approaches to define new protocols, since SOAP and REST differ radically in their approach.

4.1 Protocol Approaches

Custom Protocol Approach This is the manual approach, a team studies the problem domain and develops a protocol upon some existing protocol. Examples are HTTP or SMTP which are based on TCP. If the protocol is not standardized, this strategy does not allow much reuse between the different protocols, only via wrappers.

The custom approach is very hard to do right and therefore only suited for experts.

Framework Approach The framework approach is based on the idea that we will constantly need to create new protocols, so it makes sense to establish a framework for doing so. SOAP together with the Web-service Description Language (WSDL) is such a framework. It provides a

- common type-system
- common service description language
- common addressing model
- common security

The framework approach has many advantages over the custom protocol approach:

- pooling of knowledge

– allows development of a common infrastructure
– lower entry barrier for business developers
– tool support possible

The disadvantage, of course, is that different protocols are not interoperable, which brings up the need for a standardized effort for building domain specific and interoperable protocols.

SOAP/WSDL is the most popular framework. It started 1998 as a Microsoft initiated technology, and defined a handful of primitive types and composites which could be tunneled through the Web. SOAPs intend was to enable DCOM over the Web without being caught by a firewall, in other words SOAP was designed to be a RPC middleware that uses Web-protocols (SOAP 1.0). IBM joined the efforts leading to SOAP 1.1 which got submitted to the W3C, the specification was merely reorganized, and no major features were added. The W3C took over SOAP and released the SOAP 1.2 recommendation in the middle of 2003.

SOAP itself does not provide application-level interoperability unless the participants agree on *how to use* SOAP. An analogy is that SOAP lets us define many *verbs* on many *nouns*, or in other words: we can define operations (verbs) on objects (nouns).

Horizontal Protocol Approach The third approach uses general purpose protocols. Instead of developing new domain-specific protocols we could agree on few general-purpose applications protocols. Here we define a common set of polymorphic operations which can be used to operate on objects (resources).

The horizontal approach is very common in the computer age. Horizontal protocols often define the operations in terms of CRUD (Create, Retrieve, Update and Delete). SQL, Filesystems and HTTP ad-

here to the CRUD pattern. SQL with *INSERT, SELECT, UPDATE and DELETE* and HTTP with *PUT, GET, POST and DELETE*, common operations on resources were the key enabler for interoperable services – a filesystem is as happy to store a picture as it is to store an email.

4.2 Standardizing

Application-level protocols can basically standardize three concepts (based on Paul Prescods article on *Standardization*[10]):

- Addressing
- Methods
- Messages

Addressing REST Web-services use the URI as the general resource identifier, allowing global effects, since no proprietary extensions are needed.

SOAP has no global addressing scheme, URI centric addressing is possible, but requires vendor support. To support URI centric addressing, an additional step is needed: endpoints have to be converted into URIs. This requires the server vendor to allow dynamically associate an endpoint with an URI.

```
stockservice = new WebServiceProxy("http://...")
IBM_endpoint_URI = stockservice.getStockEndpoint("IBM")
IBM_endpoint = new WebServiceProxy(IBM_endpoint_URI)
IBM_endpoint.getStockValue()
```

To simplify life, SOAP toolkits allow implicit addressing of the required service.

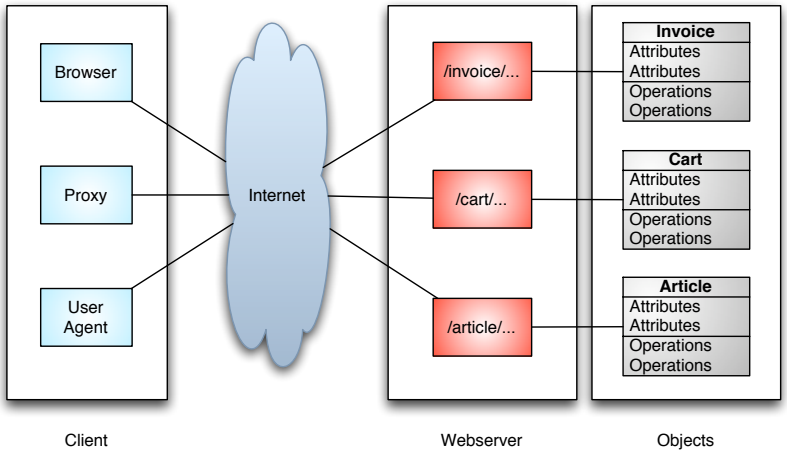

Fig. 7. In a REST architecture, each resource (in this example objects) have their own URI and can therefore be addressed directly.

```
stockservice = new WebServiceProxy("http://...")
stockservice.getStockQuote("IBM")
```

This addressing scheme is completely proprietary, and allows no interoperability!

Methods REST Web-services using HTTP have their standard set of methods (PUT, GET, POST and DELETE), every component capable of HTTP is able to talk to REST Web-services. No further agreement on the methods is needed.

SOAP requires a separate language (WSDL) to describe the methods that services offer.

Messages Neither REST nor SOAP standardize the message payload. This would be an pointless task since requirements change very frequently. The best we can do is to make it extremely easy to cope

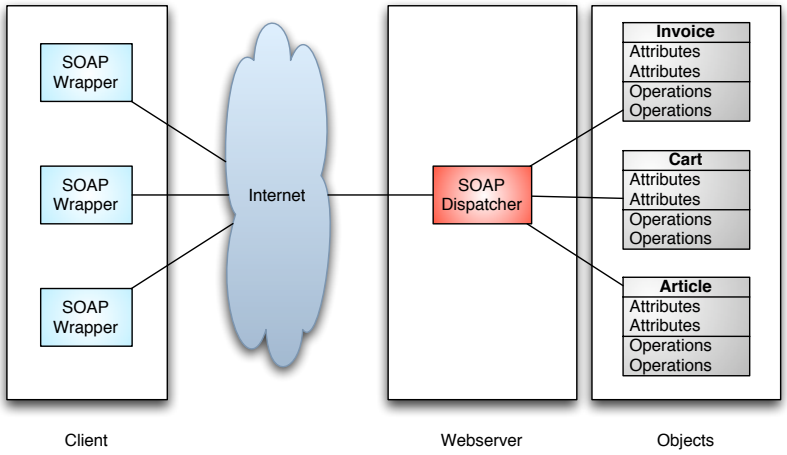

Fig. 8. In a SOAP architecture, each message has to go through an known endpoint (the dispatcher). The proprietary implicit addressing allows the dispatcher to determine which object is addressed.

with changing payloads. XML and associated standards simplify the task of defining the message payload.

REST does standardize *addressing* and *methods* so developers can focus on the message payload. SOAP requires us to define addressing and methods before we can even begin to think about the message payload.

4.3 More differences

REST explicitly defines intermediaries which are capable of reducing the network load – and therefore improving the perceived performance – by caching. SOAP messages do not differentiate between cacheable and non-cacheable responses, which lead to scalability problems. Intermediaries are not the only concept within REST to improve scalability: the requirement of the server to be stateless con-

tributes as much to scalability as do intermediaries. SOAP services do not explicitly give a recommendation on the server-side state.

Another upcoming problem has to do with security. SOAPs first intend is a to be a RPC protocol over HTTP because existing RPC protocols like CORBA and DCOM get blocked by firewalls, whereas HTTP is generally open. Existing firewall products do not understand SOAP messages since each method has its own semantics. REST uses predefined methods firewalls can interpret and block if necessary. For example GET is allowed, but PUT, POST and DELETE are blocked or restricted to predefined URIs directly on the firewall. One possible outcome could be, that all SOAP messages get blocked on the firewall for security reasons.

REST itself is no product, or standard so no company is going to promote it. SOAP instead has rich vendor support and toolkits allowing developers to expose objects as Web-services.

Amazon – as one of the largest Web-service provider – reported, that their REST interface is much more popular than their SOAP interface. This seems to be an indication that REST is indeed easier and at least as powerful as the SOAP approach.

SOAP has the advantage of a service description language and is very extensible. HTTP has an extension mechanism, which is seldom used and not well known.

5 Future

REST applications have proven to be scalable as well as easily integrateable, and with the dawn of the *semantic web* machine readable and interpretable data types are on the way to support dynamic composition.

The HTTP OPTIONS method provides a reasonable good way for Web-services to promote their interface in a – yet to be specified – service description language. Some attempts of a REST service description language were made, but none got standardized by now.

6 Related Work

Roy T. Fielding was the first to articulate REST in his dissertation [4], together with Richard N. Taylor they published the concept as *Principled Design of the Modern Web Architecture* in 2000 [5] and 2002 [6] providing the basics of the Representational State Transfer.

Rohit Khare and Richard N. Taylor published a paper on *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems* in 2004 [8].

Paul Prescod compares SOAP and REST in his paper [10] and provides some background to the issue in [9].

Michael zur Muehlen, Jeffrey Nickerson, and Keith Swenson explain in their paper Developing Web Services Choreography Standards [12] REST and SOAP choreography.

References

1. T. Berners-lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): generic syntax. Technical Report Internet RFC 2396, IETF, 1998.
2. F. Davis. WAIS interface protocol prototype functional specification. Technical report, Thinking Machines Corporation, 1990.
3. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Technical Report Internet RFC 2616, The Internet Society, 1999.
4. Roy T. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, Dept. of Information and Computer Science, University of California, Irvine, 2000.

5. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416. ACM Press, 2000.
6. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.
7. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. Technical Report Internet RFC 2617, The Internet Society, 1999.
8. Rohit Khare and Richard N. Taylor. Extending the representational state transfer (REST) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.
9. Paul Prescod. Roots of the REST/SOAP debate. Technical report, Prescod.net, 02 2002.
10. Paul Prescod. SOAP, REST and interoperability. Technical report, Prescod.net, 02 2005.
11. W3C. XML base. Technical report, W3C, 06 2001.
12. Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standards. *Support Systems*, (40):9–29, 1 2005.