Service-driven Computing with APIs: Challenges and Emerging Trends

Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski
Univ. of California at Santa Barbara, USA

ABSTRACT

While SOAP and REST have both been used widely to implement Web services, over time REST has emerged as the predominant approach. REST provides developers with a lower barrier to entry for implementation and greater development flexibility than SOAP. Its architectural conventions and best practices can be integrated into Web services incrementally as opposed to the all-or-nothing adoption of SOAP. In order to achieve generality, SOAP standards are extensive, rigid, and complex. This complexity can lead to implementations that introduce significant overhead on the network bandwidth consumption, execution times, and throughput of SOAP services especially in the emerging resource restricted mobile realm. This chapter provides an overview of the logical and physical design of modern Web services and discusses the strengths and weaknesses of the predominant styles. It provides evidence and reasoning behind the emergence of REST as the leader for the development of next-generation Web APIs and services. The chapter also delineates the key technologies that underlie REST and describes emerging and future research directions in support of REST-based APIs and service development.

INTRODUCTION

Web services (W3C, 2004b) simplify the development of network-accessible, distributed applications by combining the ubiquity of the Internet with familiar protocols of the World Wide Web (WWW), well-defined interfaces, and easily integrated software architectures. They expose functionality, business logic, and data as network-enabled modules, which can be consumed over the network by client applications written in a variety of programming languages. The Web service design and development model, also referred to as Service Oriented Architecture (SOA), describes how to architect, reuse, and integrate Web service modules as the building blocks for new systems and services (Dan, Johnson & Carrato, 2008). SOA is used increasingly for a wide range of application domains including business collaboration and productivity, Web/mobile access and communications, large scale data integration and analysis, multi-player games, and Cloud computing (compute resources, software, and data "as-a-service") (Haines & Haseman, 2009). Many technology industry leaders (e.g. Google, Yahoo, Amazon, eBay, and Facebook) use SOA as the basis of their products, which they expose to external developers for integration with new products, services, and platforms.

Systems implemented using Web services and SOA principles tend to be loosely coupled and resilient to change, failure, and interruption (Offermann, Hoffmann & Bub, 2009). The modular nature of Web services promotes separation of concerns, and makes it easier to design and reason about distributed applications. In addition, Web service encapsulation enables developers to choose the most appropriate programming language and technologies for their implementations and to isolate change across complex

systems. This service-driven approach for developing applications minimizes code duplication and eases the assembly of complex systems, thereby greatly improving developer productivity over non-service-oriented methodologies. By composing an application from existing services that encapsulate common tasks such as database access, logging and security; application developers can work at a higher level of abstraction, thereby saving development and debugging time. In particular, the reuse of services as modules increases reliability and stability in the resulting software system, since testing and quality assurance can focus on integration rather than on the constituent services.

Logically, a Web service consists of four primary components:

- An architecture that governs the logical organization of data, code and communication of the service.
- Abstraction that hides the implementation details of the service and that enables the service architect to control the functionality that is exposed to users and other programs (service clients).
- An implementation that contains program code for computation and data manipulation that is executed when a client accesses the service.
- An application-programming interface (API) that defines and controls the operations that clients perform to access the implementation as specified by the abstraction layer.

The architecture and abstraction are the logical constructs of Web service design and the API and implementation comprise its physical manifestation. From an implementation perspective, APIs decouple the functionality necessary to allow controlled access to a service from the technologies that are used to implement the functionality of the service itself (Beyer, Chakrabarti & Henzinger, 2005). That is, the API preserves the service functionality for clients accessing the service while the technologies used to implement it can change, particularly as technological advances reduce implementation costs. Similarly, it is possible for a user to swap in/out similar services if the APIs are compatible, with minimal client program modification.

There are two primary messaging styles for interfacing Web services to the WWW via APIs: SOAP and REST (REpresentational State Transfer) (Pautasso, Zimmermann & Leymann, 2008). SOAP services (Chavda, 2004) conform to a layered architecture that is described by a well-defined standard in which each layer is strictly controlled by one or more additional standards. The design of SOAP is as a very general protocol for exchanging data over a network using different transport layers via remote procedure calls (RPCs). When used for Web services over HTTP, SOAP promotes predictable behavior, service interoperability, and reuse. REST (Fielding, 2000) describes a significantly simpler client-server Web service style with straightforward architectural constraints and a set of "best practices" (as opposed to standards). REST is highly flexible, advocating the use of loosely coupled and stateless design principles. REST relies on lightweight abstractions and protocols that are prevalent in the WWW (e.g. HTML and HTTP) to facilitate interoperability.

While SOAP and REST have both been used widely to implement Web services, over time REST has emerged as the predominant approach. The primary reasons behind this trend are two fold. First, REST provides developers with a lower barrier to entry for implementation and greater development flexibility than SOAP (Pautasso, Zimmermann & Leymann, 2008). In particular, REST design conventions and best practices can be integrated into Web services by developers incrementally as opposed to the all-or-nothing adoption of SOAP. Second, to achieve generality, SOAP standards are extensive, rigid, and complex. This complexity can lead to implementations that introduce significant overhead on the network bandwidth consumption, execution time, and throughput of SOAP services (Mulligan & Gracanin, 2009; Potti, 2011). For emerging mobile, resource restricted, and battery-powered devices (key deployment targets of Web services today), this overhead can be prohibitive (Phan, Tari & Bertok, 2006). This

complexity can also make it difficult for developers to change or extend existing SOAP APIs and services without significant code modification and effort.

As a result of these differences, recent advances in Service-driven computing are increasingly based on REST. In this chapter, we provide an overview of the logical and physical design of Web services and compare and contrast the SOAP and REST messaging styles. We then focus on recent technological advances that have emerged for architecting REST-based APIs and services and describe emerging and potential research directions for service-oriented computing using REST.

BACKGROUND

As shown in Figure 1, a Web service is logically composed of four interrelated components. The components are the architecture and abstraction layer, and the implementation and its interface.

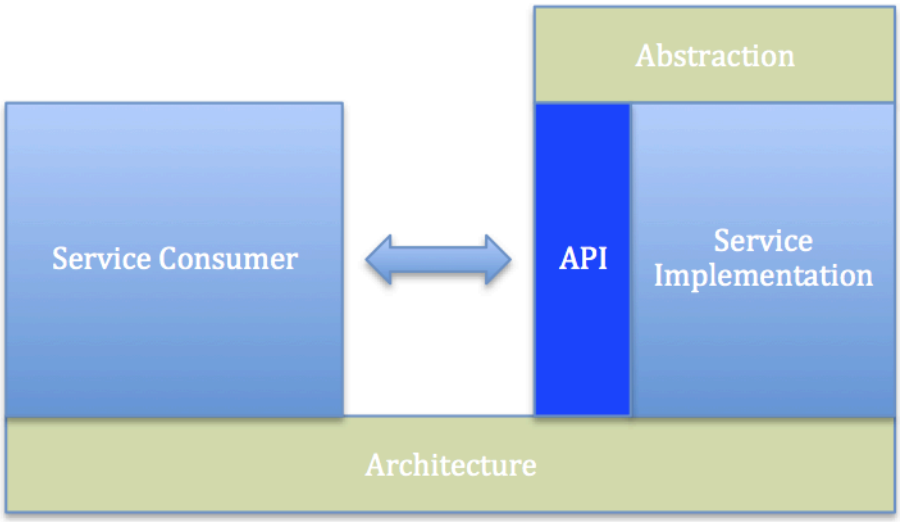

Figure 1: Logical composition of a Web service.

Architecture and Abstraction

The architecture of a Web service governs the logical organization of the data, code, and communication of Web services. The architecture describes the capabilities and restrictions stemming from the use of specific communication protocols, message formats, quality of service targets, and application state management techniques. The architecture also defines how data, code, and communication are encapsulated into modules, and how modules interoperate. As with any software design methodology, architectural decisions for Web services significantly impact performance, scalability, and maintainability.

The abstraction of a Web service describes the translation from what is exposed to clients of the service to what is implemented by the service. It is primarily concerned with two things:

- How a Web service views the data that it manipulates (data abstraction)
- How the executable code is structured in a Web service (code abstraction)

The data abstraction defines the data models and structures that are used by the Web service. For instance, a Web service can use an object-relational data model comprised of data structures such as objects, lists, tuples and blobs. The data abstraction determines how the physical data is mapped to the high-level data model that is exposed to the service client. A powerful data abstraction makes it easy to program a Web service and understand its functionality, thereby simplifying both the service and client implementation. Code abstraction dictates the programming model that the service uses, such as client-server, event-driven, and execution based on RPCs.

Implementation and API

The physical manifestation of a service is the implementation and its API. The service implementation contains the computational logic that is executed when a client invokes the service. It primarily consists of executable code, data, and service configuration information. Depending on functionality of the service, the service implementation may perform a wide range of tasks including computation, string manipulation, text processing, data analytics, database access, and cryptographic operations. A service implementation may invoke other remote services if needed. Theoretically, any non-service program can be transformed into a service implementation and exposed to clients as a Web service. However, some Web service middleware restricts certain operations for reasons of access control, portability, and security. For example, the Google App Engine and Microsoft Azure Cloud service platforms do not allow service implementations to access the local file system.

The API of a Web service controls the parts of the implementation that are visible to the clients, and thus how clients interact with the service programmatically. The API decouples clients of the service from the service implementation so that each can be developed and/or modified independently thereby helping to isolate the impact of change. As long as the interface of a service does not change or remains backward compatible, the implementation of the service can change without requiring modification to client code. This flexibility and separation of concerns is one of the primary reasons for the wide spread use and success of Web services today.

The API defines the requests that a service can receive and the responses it returns. Typically a service interface describes:

- The operations supported by the service (i.e. those that can be invoked by a remote user)
- The structure and format of the requests (inputs) and responses (outputs) that can be processed and generated, respectively, by the service
- The service endpoints (i.e. the transport level addresses of the service). For a service exposed over HTTP, these are the HTTP URLs that clients can use to access the service.
- Other transport level details (e.g. HTTP methods and headers)
- Any Quality of Service (QoS) constraints (e.g. if the service is secured, this describes the security protocols that should be used by clients when contacting the service)

Web APIs (i.e. APIs for Web services) inherit their role and benefits from programming language APIs. Table 1 identifies the mapping between the terminology used in Web services and programming language APIs.

Web API Element	**Programming Language API Element**
Operations	Functions/Methods
Request	Input arguments
Response	Return type

Table 1: Commonalities between Web APIs and Application Programming APIs

SOAP and REST Web Services

SOAP and REST are the two primary messaging styles for implementing Web services. The name SOAP is an acronym of "Simple Object Access Protocol" and REST is a contraction of "REpresensentational State Transfer." These two styles differ significantly from each other in terms of their abstractions, architecture, APIs, and implementation. SOAP services conform to a rigid architectural style where each layer of the architecture is strictly controlled by one or more well-defined standards. Message formats, endpoints, APIs and QoS guarantees are governed by individual SOAP standards.

REST services are based on a set of simple architectural constraints and best practices in which most of the details associated with service API and implementation are left to the developer. REST allows developers to choose the message format, endpoint control, APIs and QoS that is most appropriate for their service. Next, we overview each architectural style and discuss its strengths and weaknesses.

SOAP Services

Feature	Standards	Standard Body
Message format	SOAP	W3C
Service description language	WSDL	W3C
Service discovery & integration	UDDI, WS-Discovery	OASIS
Security	WS-Security	OASIS
Reliable delivery	WS-ReliableMessaging	OASIS
Data type description	XSD	W3C
Policy enforcement	WS-Policy	W3C
Binary data transfer	MTOM, SwA	W3C

Table 2: Significant standards associated with SOAP services

Table 2 lists the SOAP standards identified as key by W3C (W3C, 1994) and OASIS (OASIS, 1998). These standards define an architecture for developing, hosting and managing Web services. Figure 2 presents a high-level overview of the SOAP architecture and its standards. SOAP standards do not restrict how data is represented or what transport protocol is used (although HTTP is the most common) by a Web service. They do however specify that computation be performed via RPCs and that communication (messages) between clients and the service be formatted in XML. In general, SOAP APIs are similar to traditional (programming language) RPC interfaces (Dissanaike, Wijkman, & Wijkman, 2004; Davis and Zhang, 2002; Looker and Xu, 2003).

An RPC interface consists of one or more network-enabled functions where each procedure accepts zero or more inputs and may generate output. Similarly a SOAP API consists of one or more distinct operations that are identified by unique names. Most operations consume a SOAP request message as the input, and produce a SOAP response message as the output. In SOAP parlance, such operations are referred to as "In-Out" message exchange patterns (MEPs). There are additionally seven MEPs that can be incorporated into API designs. The similarity to traditional RPC systems, simplifies the process of replacing legacy RPC systems with SOAP-based Web services.

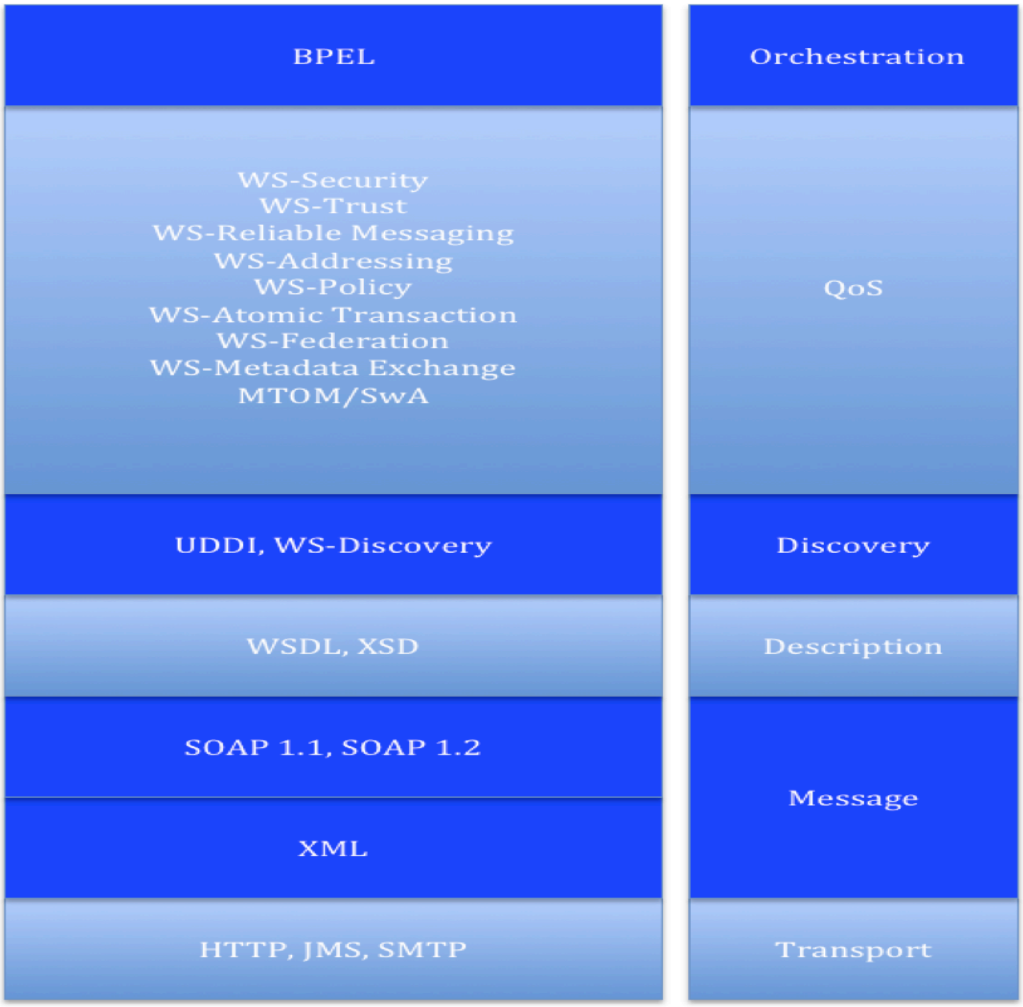

Figure 2: SOAP architecture (inspired by Microsoft, 2004)

The SOAP architecture is largely independent of application protocols and therefore critical metadata and control information must be embedded in SOAP messages. SOAP services implemented using HTTP for transport use the POST method for messages and status codes 200 and 500 for normal and exceptional responses, respectively. HTTP URLs, application layer headers, and status codes do not play a significant role in a SOAP API. Instead, SOAP services are described by stating the operations and the schema of the SOAP messages. SOAP service clients are tightly coupled to the service as a result of this embedding.

RESTful Services

REST is a stateless, client-server architectural style for designing distributed applications. A Web service that adheres to the REST architectural style is often termed a "RESTful" service. Roy Fielding coined the term REST in his doctoral dissertation (Fielding, 2000), and the core principles of REST were developed by the W3C Technical Architecture Group (W3C, 2001) in parallel with the HTTP/1.1 standard (W3C, 1999) of which Fielding was a principal author.

A REST architecture is characterized by six architectural constraints.

- ***Client-Server Model:*** The system is composed of several components. Some components function as servers (service providers), while the others operate as clients (service consumers).

Client-server interactions take place through a well-defined interface and the interface helps enforcing separation of concerns in the system design.

- ***Statelessness:*** Server-side components in a RESTful system are stateless. That is, the servers do not keep track of the clients that interact with the services. Each client request is treated as a self-contained entity and processed independently of any past or other concurrent client requests. Any important session state must be stored in the client-side. This constraint helps the server components to be lightweight and highly scalable.
- ***Caching:*** Client applications should cache responses. Doing so approach greatly speeds up client and reduces the load on the network and server-side components.
- ***Layered Design:*** REST principles mandate a layered design, in which a client request may have to traverse through multiple layers to reach a particular service endpoint. The client-side components in general are oblivious to layering which makes it possible to add, remove and change the layers without making client-side code changes.
- ***Uniform Interface:*** The interface (API) between clients and servers decouples critical components from each other and makes the system more flexible in the face of change. A clean and uniform API makes it easier for the developers to both use and extend the system.
- ***Code on Demand:*** REST optionally permits the server to transfer executable code to clients for execution by the clients.

RESTful services expose a set of entities known as "resources". Resources are the key data and code abstractions that serve as the basis of all RESTful interactions. Each resource typically represents some object or abstraction managed by the application.

RESTful services based on HTTP use unique URLs to represent each resource. Resource identifiers, references, and metadata can be exchanged among software agents by embedding such information as hypertext in service requests and responses. Service clients manipulate the resources by making different types of HTTP calls on the resource URLs. Services can use any HTTP method (e.g. GET, POST, PUT, DELETE) to implement an operation. Moreover, services can employ any content exchange format (e.g. XML, JSON, plain text) to represent resources in a machine or human readable form. Services can also use HTTP headers and status codes to exchange various control information and application metadata.

The API of a RESTful service differs greatly from the RPC style of SOAP services. REST APIs consist primarily of hyperlinks, HTTP method, and status codes. Instead of a set of operations based on the traditional request-response model, REST APIs are structured as a set of inter-connected resources. Clients typically manipulate resources and navigate through the service by following hyperlinks that connect one resource to another.

CLOUD, MOBILE APPS AND THE EMERGENCE OF REST

As SOA technologies have matured and Web services have gained wide spread use for exposing software and data over the network "as-a-service", REST has emerged as the architectural style of choice. Figure 3 illustrates this trend by showing how the distribution of Web-services registered with ProgrammableWeb.com (ProgrammableWeb, 2013) has changed from 2008 to 2010. This data indicates that REST is becoming the predominant architecture employed by developers for the design and implementation of Web services.

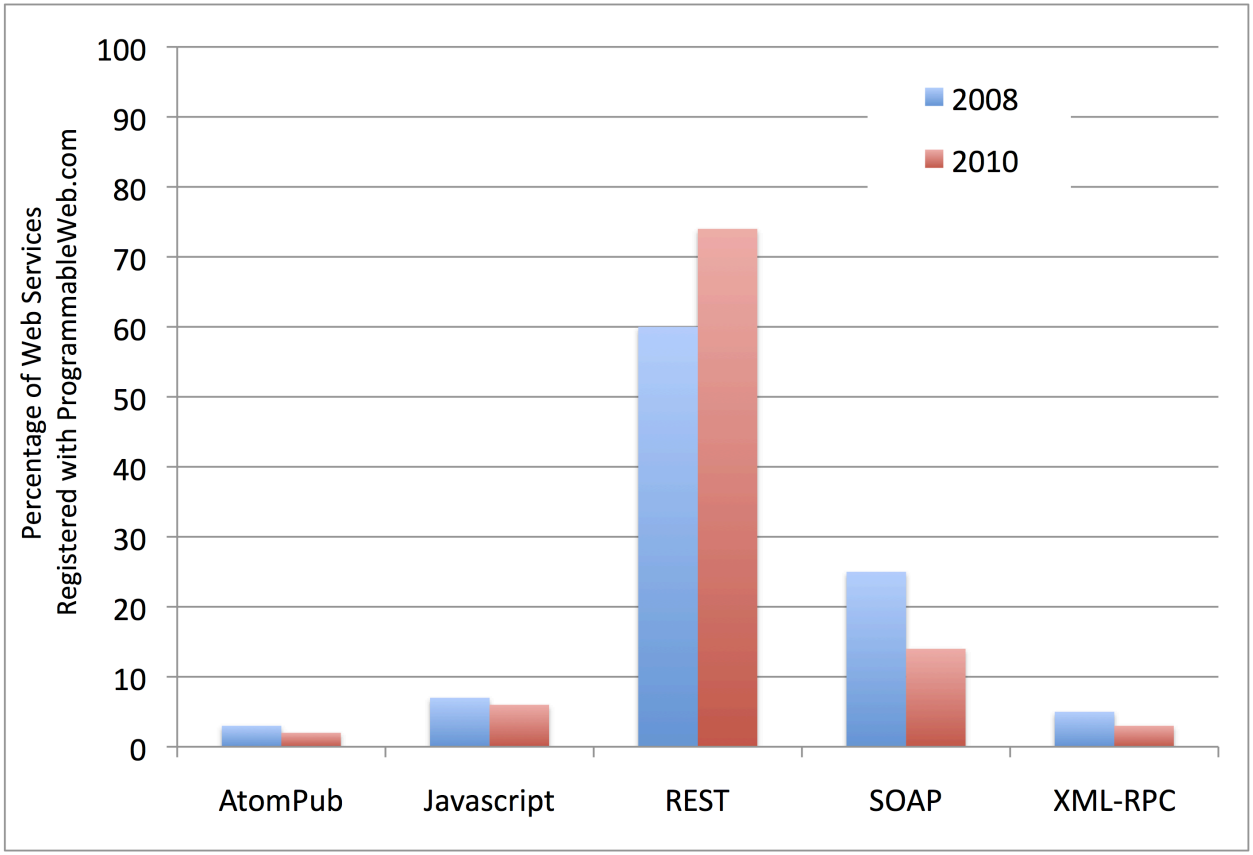

Figure 3: Distribution of API protocols and styles (Adapted from http://blog.programmableWeb.com)

There are two primary reasons for the rise in popularity of REST. First, because SOAP attempts to guarantee interoperability, the middleware that implements SOAP is significantly more complex than REST. This complexity must be adopted up front and in its entirety, impeding time to solution and the ability to prototype rapidly. REST, in contrast, leaves interoperability to the developer. REST-based code reuse suffers versus SOAP as a result of these differences, but because applications evolve so rapidly in the modern Web environment, the value of reuse is diminished relative to the speed with which code can be adapted.

Secondly, the complexity of SOAP middleware and the requirement that XML be used as the wire transfer format creates performance overhead in SOAP applications that is not imposed upon REST services. This overhead is particularly onerous for mobile applications that run on power restricted (e.g. battery powered) and resource-constrained devices. Studies have shown that SOAP services result in higher latencies and higher bandwidth utilization, thus making them a poor fit for integrating mobile apps with their backend services (Mulligan & Gracanin, 2009; Potti, 2011). The lightweight implementation of REST and the ability to prototype has clearly overshadowed the software engineering advantages of SOAP when it comes to implementing mobile apps.

The emergence and popularity of cloud computing has also contributed to the rise of REST. Cloud technologies today make extensive use of REST for exposing resources and cloud services via web APIs. Cloud computing systems are highly distributed by nature and the services they host and the resources they expose evolve continuously. Thu, the flexibility REST offers to API implementation and evolution make it a popular choice for remote integration and consumption of cloud services. This trend is further reinforced by the fact that most REST-based mobile and web applications are either directly hosted in the

cloud or frequently interact with one or more cloud services (e.g. Google App Engine, Amazon S3). Today, most Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) solutions are exposed to the users as a series of web-based REST APIs (IBM, 2013). In February 2012, ProgrammableWeb reported that 75% of all available cloud APIs follow the REST architectural style. Morover, most Infrastructure-as-a-Service (IaaS) offerings including Amazon EC2 (Amazon, 2013) and Rackspace (Rackspace, 2013) expose resources to customers via REST APIs.

Despite its popularity, there remain a number of challenges associated with using REST for the development of Web APIs and services. The key limitations include

- *The lack of a fixed and standardized framework:* The flexibility of REST has resulted in a large number of diverse RESTful applications that are incompatible or not interoperable with each other. Moreover, porting an application from one RESTful service to another similar one is typically a non-trivial task.
- *REST is tightly coupled to HTTP:* The simplicity facilitated by this coupling makes it challenging to port a REST service to another transport protocol. In addition, REST inherits HTTP's synchrony, which precludes the use of advanced message exchange (anything other than simple request-response communications) and server-initiated notifications. Moreover, as Web services become increasingly complex, HTTP operations become insufficient and the HTTP POST command must be overloaded.
- *The lack of message level security enforcement:* SOAP services have several security standards including WS-Security (OASIS, 2002) and WS-SecureConversation (OASIS, 2007) that allow parts of messages to be encrypted, while leaving a few sections in plain text to facilitate content-based routing. There is no REST equivalent to support advanced security features; the only security mechanism available to REST services are those of the transport level (SSL/TLS) (IETF, 2008).

Because of the popularity and wide spread use of REST, there have been numerous advances that attempt to address these limitations with technologies that facilitate interoperability and reuse, that simplify development of REST services, and that provide REST API management and governance. In addition, the HTTP community is currently pursuing HTTP extensions to better support REST and emerging research efforts target these limitations to help ensure that REST adoption and Web service development continue. We provide an overview of these advances in the following sections.

Facilitating Service Interoperability and Reuse without Standardization

The limitations that result from the lack of standardization in REST are mitigated by design conventions and best practices that facilitate service interoperability and reuse. Some of the key conventions and practices include:

- *Minimal Resource Exposure:* It is important to identify the minimal set of resources that should be exposed by the system. Exposing more resources than necessary can lead to a large and complex API that is difficult to make interoperable.
- *Data Abstractions as Attributes:* The data abstractions that are used to implement resources should be represented as resource attributes.
- *URLs define Application-visible Names:* URLs should be used to implement name hierarchies that are meaningful to specific application functions and data.
- *Appropriate HTTP Method Usage:* While the HTTP POST operation is general-purpose, interoperability is enhanced by intuitive choices of HTTP operations (e.g. a POST should not be used to delete a resource instead of a DELETE).

- *Self-contained Message:* Each message should carry all the control information, data and metadata relevant to the target REST operation so that clients and servers do not need to infer critical information. In particular, HTTP status codes and headers should be used to disambiguate messages.
- *Hypermedia drives State Transitions:* Services should respond with hypermedia content (e.g. hyperlinks) that triggers state transitions in the client application. Using hypermedia-based state transitions in the client is part of a paradigm known as Hypermedia as the Engine of Application State (HATEOAS). HATEOAS precludes the need for a fixed interface to integrate services and clients, and replaces interface descriptions language (IDL) semantics in REST with the browser paradigm so that a client (emulating the functionality of a browser) "navigates" the interface using URLs and hyperlinks.

In addition, developers can employ the Richardson Maturity Model (Richardson, 2008; Parastatidis, Webber, & Robinson, 2010; Fowler, 2010) to determine how well a Web service adheres to REST principles. The Richardson Maturity Model categorizes a Web service into four levels, numbered 0 through 3. Each higher level corresponds to a more complete adherence to REST principles than the lower levels. Also, each level contains all the properties of the level below it.

- Level-0 services use "Plain Old XML" (POX) over HTTP to communicate. Each service has exactly one URL (endpoint) and all the requests use the HTTP POST method. Clients invoke different operations on the service by sending different XML payloads. This approach is similar to the traditional RPC model. Thus, we can categorize SOAP-style Web services as level-0 services.
- Level-1 services use resources to represent code and data abstractions. Developers identify each resource as a separate URL; the service consists of multiple URLs (endpoints). However, like level-0 services, these services also rely on a single HTTP method (usually POST). Different operations on the same resource are invoked by sending different payloads.
- Level-2 services use different HTTP methods to invoke different operations on the same resource. The goal of level-2 is for the service to adhere more closely to the underlying semantics of HTTP in terms of its operations and responses.
- Level-3 services send hypermedia content in response messages that identify the options a client can take for the subsequent operation. Client applications make local state transitions based on the received hypermedia content and navigate their way through the REST API. More specifically, level-3 services make effective use of HATEOAS.

Interestingly, most Web services in production today that are considered RESTful fall into either level-0 or level-1 of the Richardson Maturity Model (Richardson, 2008; Parastatidis, Webber, & Robinson, 2010; Fowler, 2010).

Frameworks in Support of REST API/Service Development

To make up for the lack of middleware enforcement of standards, many server-side programming frameworks have emerged to facilitate and simplify implementation of RESTful services. For example, the LAMP stack (Linux, Apache HTTPD, MySQL, PHP) is a popular open-source software stack. Developers increasingly employ LAMP stacks to implement RESTful services in PHP (PHP, 1995) that are deployed over an Apache HTTPD Web server (Apache, 1995). Depending on the requests received by the Web service, the PHP program interacts with a MySQL database (MySQL, 1995) to retrieve and modify the corresponding data entries (resources).

The Java J2EE (JavaEE, 1999) standard defines a collection of Java libraries to facilitate RESTful service development. Using this framework, developers program a RESTful service as a Java servlet and deploy it via a J2EE-compliant application server such as Apache Tomcat (Apache, 1999) or JBoss (JBoss, 2012). Typically, J2EE services interact with a database using the JDBC standard (JDBC, 1997). Java developers also use the popular Spring framework to implement and deploy RESTful services.

The Java platform also defines the "Java API for RESTful Web Services" standard (JAX-RS or JSR311) (JAX-RS, 2008) that facilitates development of RESTful Web services. JAX-RS takes a similar approach to the SOAP standard JSR224 (also called JAX-WS) (JAX-WS, 2006) but focuses entirely on exposing Java code through RESTful APIs. Similar frameworks are available for different language technologies, including NodeJS (for JavaScript) (NodeJS, 2009), Sinatra (Sinatra, 2007) (for Ruby) and Django (Django, 2005) (for Python).

Popular cloud platform as-a-service (PaaS) technologies such as Google App Engine (Google, 2008), Microsoft Azure (Microsoft, 2009), and AppScale (Krintz, 2013) simplify development, deployment, and interoperation of RESTful services across programming languages using highly scalable and available Cloud computing technologies. Such platforms scale and tolerate faults in RESTful services by replicating them across server resources and employing modern load balancing techniques to distribute workloads. The stateless nature of REST and its core principle of caching enables platforms such as these to simplify management of RESTful services by automating fault tolerance, elasticity and scaling, as well as configuration and deployment so that developers can focus on their innovation.

REST APIs and API Management

RESTful services identify resources via unique HTTP URLs and invoke operations using HTTP methods. RESTful services also use HTTP to identify a content exchange format to use in requests and responses, depending on the type of client accessing the service (e.g. browser-based, mobile app, other). Ideally, Web services use HTTP headers and hypermedia content to exchange service metadata and to trigger application state transitions in HTTP messages. As a result of this coupling between REST and HTTP, we can describe the API of a RESTful Web service in terms of how the API employs HTTP features. That is, we can fully specify the API of a REST service using the following HTTP properties of the API:

- HTTP URLs (or URL patterns) that can be invoked
- HTTP methods that can be invoked
- Structure and media types (content exchange formats) of request payloads
- Structure and media types (content exchange formats) of response payloads
- HTTP headers and status codes of requests
- HTTP headers and status codes of responses
- Hypermedia content used in responses

Exposing an existing system via a REST API thus reduces to exposing the system via HTTP while adhering to the principles of the REST model. As a result, we are able to construct a machine-readable description of a REST API by documenting these properties using structured markup language, such as XML or JSON.

Note that for many Web services it is not possible to document or even enumerate all the URLs associated with a REST API. Systems often manage vast numbers of resource instances, each potentially with a unique URL. To solve this problem, developers assign URLs to resources hierarchically to establish an unambiguous pattern that describes the structure of the URLs supported by the API. By doing so, it becomes possible to use regular expressions to describe the URLs associated with an API or resource.

The Java servlet specification (Java, 2009) uses this idea to allocate URLs to different servlet implementations. The servlet specification identifies three types of URL patterns.

- Exact patterns: Specifies the full URL path exactly and explicitly. (e.g. /my/path)
- Extension patterns: Specifies the URLs in terms of the extensions of the files/documents to which they point. (e.g. *.jsp, *.do)
- Path patterns: Specifies the URLs as a combination of path fragments and wildcards. (e.g. /my/path/*)

The recently adopted URI template standard (RFC6570) (IETF, 2012a) provides a more generic and language neutral approach for describing URLs of REST APIs. URI templates make use of template variables and several different wildcard symbols to describe URLs. The JAX-RS Java standard uses a subset of the URI templates standard to assign URLs to different resources.

Content Exchange Formats

Programs that rely on RESTful services interact with REST APIs by transferring resource state representations. A resource state representation is a machine-readable serialization of the current state of a resource that can be correctly transferred over a network. There are many content exchange formats (media types) that can be used for this purpose. XML and JSON are the most widely used formats for representing structured data. Content formats like HTML, plain text and CSV are also popular. APIs that communicate binary objects such as images and videos require other content types tailored to these object types (e.g. those aware of resolution and frame rate metrics).

In most cases, the efficiency of a RESTful service is directly related to the efficiency of its content exchange. Since all data communicated by each service request or response must be serialized into a binary format for network transfer (and deserialized for programmatic access), content exchange imposes overhead for (un-)marshalling of data and consumes bandwidth for data transfer. Efficient content exchange is key for wide spread use of any service, but is particularly important for services that are consumed by resource constrained devices (e.g. mobile apps). Such devices often operate with limited computing capability, battery power, and stringent networking resources.

A content exchange format thus must be compact and facilitate low overhead (un-)marshalling. Since compression ratio directly impacts (un-)marshalling overhead, content exchange formats must balance the two metrics while preserving ease of use through tooling. JSON, for example, is increasingly used as an XML replacement in RESTful services since it provides a more compact representation and is supported by a vast diversity of programming frameworks. Recently, Apache Thrift (Apache, 2010) and Google Protocol Buffers (Google, 2010) have emerged as cross-platform tools that provide more efficient binary content exchange formats (overhead and bandwidth) than JSON and XML. The respective vendors provide libraries for incorporating these content formats into RESTful service development frameworks. In addition to third-party software, the compression mechanisms (e.g. gzip) and encoding schemes (e.g. chunking) of HTTP also improve performance and bandwidth utilization of content exchange for RESTful services.

API Security and Access Control

To compensate for the lack of standards for API security and access control, REST APIs require support for user authentication. In 2010, Maleshkova et al. (Maleshkova, M., Pedrinaci, C., & Domingue, J., 2010) showed that approximately 80% of the Web APIs found on the Internet use some form of user authentication. The two most widely used techniques for authenticating REST API users are HTTP basic access authentication (BasicAuth) (IETF, 1999) and OAuth (IETF, 2012a).

With BasicAuth, clients send authentication credentials (username and password) to the remote API as a part of the HTTP header. Because BasicAuth does not utilize any encryption or signing techniques, it does not provide any confidentiality. As such, simple interception algorithms can compromise user credentials and services. For this reason, BasicAuth is generally used in conjunction with HTTPS, which provides the necessary encryption and transport level security.

OAuth is an authorization protocol for Web APIs that has also garnered widespread adoption recently. OAuth provides a mechanism for clients to access server-side resources on behalf of a resource owner. OAuth supports authorizing third-party users without having to share user credentials across different service domains. OAuth powers the "Login with Facebook" option that is currently employed by many Web applications. The OAuth 2.0 framework (RFC6749) (IETF, 2012b) and the related Bearer Token Usage (RFC6750) (IETF, 2012c) have been adopted by some of the largest Web API providers in the world including Google, Facebook and Twitter. An OAuth bearer token is a special access token (key) with which a user can access a particular Web API or an application. Clients send tokens to a remote API prefixed by the string "Bearer" as the value of the HTTP "Authorization" header. Many API providers that require their clients to use API keys to access the APIs use OAuth 2.0 and bearer tokens to handle API security.

API providers also require the ability to limit the number of requests received by their APIs. This is accomplished by restricting the number of requests that each client can send over time. APIs without such rate limiting capabilities are highly vulnerable to denial of service (DoS) attacks. Rate limiting is easily implemented by associating each API key (access credential) with a policy that defines the maximum number of requests the API key holder (credential owner) may send during a fixed time interval (e.g. an hour or a day). The service must track the credentials and client use rates. When the number of requests from the same user exceeds some limit defined in the policy, the service drops (throttles) the requests.

API Management Platforms

Despite the benefits that are possible from exposing critical data and applications as APIs, doing so can also have negative side effects. Some of the possible issues are:

- *Security violations:* Each API exposed to the Internet is a potential opening for a malicious client to gain access to confidential information or disrupt the service by causing errors and outages.
- *Client and use tracking:* As on-line transactions are automated, it becomes increasingly challenging for organizations to track service users.
- *Intellectual property control:* Web APIs allow third parties to develop applications and potentially to profit from their use.

To overcome these issues API providers often adopt an API management platform. An API management platform is a software system that is managed by a trusted third party or the service owner that simplifies and unifies management of APIs and their service implementations. Advanced API management automates API availability advertising and subscription, API subscription authentication, authorization, and revocation, monitoring and analysis of API use, and support for service-level agreements (SLAs).

SLAs define the client's rights and responsibilities as an API client. For instance, an SLA may define attributes such as:

- Maximum number of requests that can be sent by a client (rate limit)
- Cost formula (cost per request)

- Service availability guarantee from the API provider (this is usually specified as a percentage – e.g. 99.999%)
- Security guarantee from the API provider

A comprehensive and well thought out SLA provides a defense against possible security violations (e.g. DoS attacks) and creates opportunities to monetize APIs. In addition to the SLAs, it is also common practice for API providers to have an additional license or a set of terms for the developers who develop applications or services using the API. These legal documents provide additional defense against potential API abuse or theft.

Most information technology service providers employ some form of advanced API management. Organizations such as Google, Yahoo, Netflix, eBay and Twitter expose their systems to the Web as managed API platforms. App developers register with these organizations, agree to their licensing terms and obtain API keys prior to developing applications that make use of their APIs. In some cases, a few non-critical APIs are exposed via unmanaged APIs. For instance, parts of the YouTube API and Twitter search API are exposed as unmanaged APIs which can be accessed without any pre-registration or API keys. However, all business-critical APIs require user authentication.

API Versioning and Lifecycle

Most programming APIs have a lifecycle. When an API is introduced, it typically is in a rapidly evolving state. API providers often change the API during this stage to test initial assumptions about service use and to investigate alternative implementations and functionality. Eventually the API evolves into a more mature and stable state where it can be made widely available (released) to clients. Successful APIs change only minimally, if at all, once released. After some time, the API providers introduce new APIs or new versions of old APIs that replace or subsume an existing stable API.

In an attempt to persuade the clients to switch to this new API, API providers deprecate old APIs. Deprecation is a courtesy notice from the API provider to a client to stop using the old API and migrate to the new API. Some time (usually about a year) after deprecation, the old API is decommissioned and retired (made unavailable to clients). REST APIs undergo a similar lifecycle, as shown in Figure 4. API lifecycle management and the associated technical concerns are related to SOA governance (Schepers, Iacob, & Van Eck, 2008).

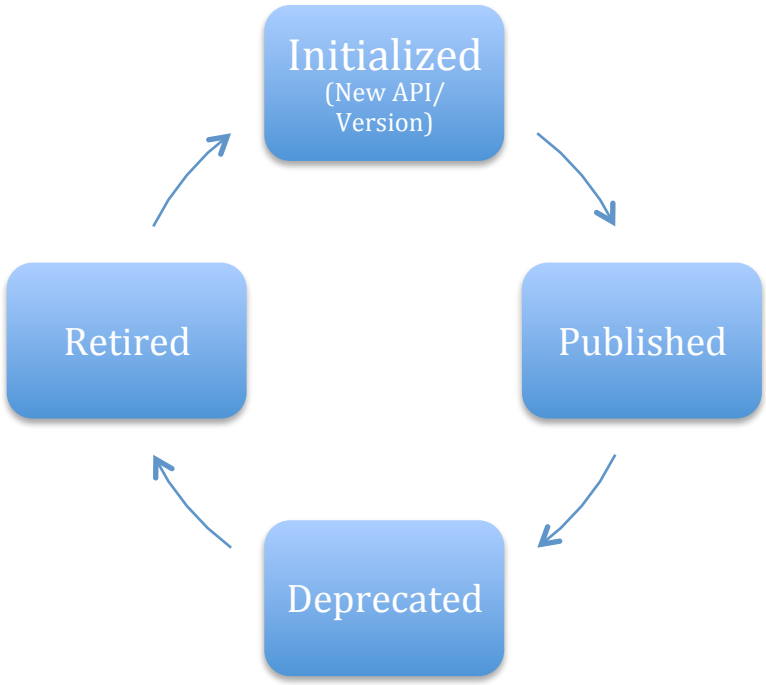

Figure 4: API Lifecycle

Often API providers have multiple versions of the same API in production. For instance there could be a situation where version 3.0 of API MYSERVICE is in the published state, and version 1.0 and 2.0 are in the deprecated state waiting for retirement. All three versions of the API are exposed to the clients and hence can be invoked. This raises the issue of differentiating traffic to different versions of the same API. For instance a client might want to invoke the latest version of MYSERVICE in application P and invoke an older version of MYSERVICE in application Q. API providers in this case can support access to different versions of the same API. There are several approaches that can be taken to solve this problem. The two most effective alternatives are:

- *Differentiate based on URLs:* In this approach, the API provider defines a resource for each version of the API identified by a URL. Path based routes are trivial to configure at the server-side which makes it easier for the API provider to implement this scheme. Moreover, an unversioned URL can be used as a pointer (redirection) to the URL of the latest API, so that client applications that do not care about different versions can transparently migrate to the latest version of the API.
- *Differentiate based on a query parameter:* Each URL takes the version number as a query parameter.

RECENT TRENDS AND ONGOING RESEARCH

As organizations become increasingly reliant on RESTful Web services, new technologies are emerging to aid Web service developers make APIs and services more reliable, interoperable, capable, and easier to implement. In this section, we discuss emerging trends and ongoing research in this important area of computer science. Over the years RESTful Web services research has branched out into many areas such as service description, discovery, formal methods, quality of service, quality of experience, performance, API management, governance and more. Of these we will focus on the recent advancements in the areas

of Web service description, discovery and formalisms as these three areas are crucial for simplifying the processes by which Web services are implemented and consumed.

Machine-Readable Descriptions for Web APIs

As described previously, HTTP can be used to describe the properties of a RESTful service. Such descriptions can simplify development and enable developers to document and communicate API functionality and use to clients. Descriptions, if machine-readable, also make it possible to automate API search and generation of client programs that use the APIs.

Unfortunately, to date there are no widely accepted API description languages for REST. As a result, applications must discover the required directions on the fly and navigate the APIs on their own. HATEOAS can help a client achieve this, but even then some kind of a machine-readable description language is required so that a client application can interpret the semantics of various resources and hypermedia items. To simplify the service development and to take advantage of the multiple benefits that machine-readable descriptions enable, several organizations and research groups are investigating REST API description languages.

Web Application Description Language (WADL) (W3C, 2009) is one of the earliest efforts to develop a description language for RESTful Web services. WADL was developed by Sun Microsystems with the goal of developing a description language for REST that provides the same functionality that the Web Service Description Language (WSDL) does for SOAP. Similarly to WSDL, WADL is based on XML and XML schema. WADL describes the resources that constitute a REST API. Each resource is described by specifying its HTTP method, request parameters, response parameters, media types and status codes. Message payloads are described using XML schema. While WADL was submitted to W3C in year 2009, W3C has not disclosed plans to standardize it to date.

Swagger (Swagger, 2010) is a more recent API description language from Wordnik (Wordnik, 2009). Swagger uses a simple JSON syntax to describe REST APIs in terms of the supported HTTP methods, data types, media types and status codes. Wordnik provides tools for auto-generating Swagger documents for RESTful services. They also provide tools for processing Swagger files to auto-generate API documentation and client stubs. These tools are available in a number of languages including Java and JavaScript. Multiple API providers have incorporated Swagger into their product/middleware offerings.

hRESTS (Kopecký, Gomadam, & Vitvar, 2008) is an HTML micro-format for "decorating" existing HTML-based API documentation with machine-readable data. Most REST APIs today have some API documentation associated with them. These documentation files are typically developed in a human-readable language (e.g. English), and are published on the Web as HTML files. hRESTS can be used to embed certain machine-readable information fragments in these HTML files. More specifically, hRESTS supports embedding information on services, operations, HTTP methods, input parameters and output parameters in Web pages. An hRESTS parser may then crawl the HTML API docs and extract the embedded information to construct a machine-readable data model such as Resource Description Framework (RDF). The machine-readable data model can be used in a number of scenarios involving service discovery and search. hRESTS has also been further extended to a more powerful micro-format named SA-REST and a more semantic oriented micro-format named MicroWSMO. MicroWSMO allows hRESTS annotations to make references to semantic Web service descriptions and semantic ontologies thereby describing each element of the service description semantically.

RESTDesc (Verborgh, Steiner, Van Deursen, Coppens, Gabarró, & Van de Walle, 2012) is a powerful description language for describing the semantics of hyperlinks. Using RESTDesc, it is possible to comprehensively describe how to invoke a given hyperlink and what to expect in return. RESTDesc is

based on the Notation3 (N3) language (W3C, 2011) syntax for encoding data semantics. A machine or a program that receives a RESTDesc description is able to construct an HTTP request to invoke the URL and to process the response returned by the URL.

SEREDASJ (Lanthaler & Gutl, 2011) is a JSON-based description language that describes REST APIs in terms of their URLs, data types and media types. SEREDASJ also supports embedding references to external semantic descriptions and ontologies in the JSON description itself. The referenced semantic descriptions can be in any suitable format including RDF, FOAF (FOAF, 2010), OWL (W3C, 2004a) and N3.

Languages like SA-REST and SEREDASJ represent a class of machine-readable description languages that use semantic annotations. Such languages enable describing Web services using a structured language such as XML or JSON, but allow embedding references to external semantic data models in the description. Semantic-unaware applications can also consume these descriptions by simply ignoring the semantic annotations. A semantic-aware application may choose to follow the references, load the external data models and interpret the semantics of the service in question. Several researchers have realized the value of this approach and have studied its implications in great detail (Cretella & Martino, 2012; Maleshkova, Pedrinaci, Li, Kopecky & Domingue, 2011). However, the complexity of semantic data models and ontologies inhibit the widespread adoption of these technologies. It has been shown that constructing comprehensive semantic ontologies is a highly complex task that requires a lot of time and manual labor (Klein & Bernstein, 2004).

While many syntactic and semantic description languages have been proposed to tackle the problem of describing RESTful APIs, none have yet to be adopted as a standard (de facto or otherwise). Thus, describing APIs via a machine-readable language remains an open research problem.

API Discovery and Search

The Universal Description, Discovery and Integration (UDDI) (UDDI, 2004) standard governs how SOAP services are published and discovered. UDDI introduces the concept of a Web services registry where service providers can register machine-readable descriptions (WSDL) of SOAP services. Interested clients may search and query the UDDI registry to locate the required services. S-RAMP and ebXML Registry are two other repository standards for cataloging, searching and discovering SOAP services. WS-Discovery is a standard that facilitates dynamic discovery of SOAP services and their endpoints either via a centralized agent (discovery proxy) or via multicast communication among servers and clients. However, there are no similar widely accepted conventions for RESTful services.

Instead, clients of the RESTful services and developers that use REST APIs manually search the Web to discover the available APIs. These search operations are generally performed using commonly available Web search engines (Google, Bing etc.), which are optimized for searching for textual Web content. Therefore the task of discovering REST APIs is usually difficult, time consuming, and error prone.

One approach that is employed increasingly to discover REST APIs and associated resources is the HTTP OPTIONS method. As per the HTTP specification (RFC2616) (W3C, 1999), a client may issue an HTTP OPTIONS request to get communication options available on the request/response chain identified by the request URL. The server may respond to an OPTIONS request by sending an HTTP message with the "Allow" header, which indicates the other HTTP methods (operations) allowed on the same URL. The server may also optionally include a payload in the response, further describing the resource. If the API or the resource can be described in some machine-readable language, the server may return a machine-readable description of the resource as the response payload of the response to the OPTIONS request.

RESTDesc uses the above approach to discover semantic descriptions of a given URL. For instance if invoking a particular REST resource returns the hyperlink http://api.example.com/next in the response, the client can make a HTTP OPTIONS request on http://api.example.com/next to obtain its RESTDesc description. The returned description specifies how the client invokes the URL and handles the response.

Public repositories such as ProgrammableWeb (ProgrammableWeb, 2013) have also emerged as on-line registries for Web API discovery. API providers publish or register their APIs with these repositories, which provides clients and developers a centralized location for searching and discovering APIs. However, the process of publishing and searching are still manual for the most part, and the search results are often far from optimal. Nevertheless, these public repositories are excellent sources of information regarding Web APIs and their usage. Many researchers use these repositories to obtain real-world statistical and social information regarding APIs.

Other research groups are investigating different technologies for classifying and searching for Web APIs. Gomadam et al. (Gomadam, Ranabahu, Nagarajan, Sheth, & Verma, 2008) have developed a taxonomy to categorize Web APIs based on factors including functionality, message format (XML, JSON etc.), and communication protocol. They use this taxonomy to model Web APIs as vectors in a multi-dimensional space and then use vector cosine distance methods to search for APIs. They further use social information to rank the results obtained from the vector-based search algorithm. That is, they rank and sort search results based on public user ratings and Alexa ratings available for the Web APIs. The authors introduce a new ranking system called the serviut rank (short for service utilization rank), which aggregates social information obtained from several sources to compute the final social score assigned to each API.

Torres, Tapia and Astudillo have introduced two metrics for ranking Web APIs based on available social information (Torres, Tapia, & Astudillo, 2011). The first, called Web API Rank (WAR), assigns a higher ranking for APIs that are widely used. This can be computed based on the publicly available usage information such as the number of applications and Mashups (collections of integrated, yet independent Web services) developed using a particular API. This information can be extracted from an on-line repository such as ProgrammableWeb. The WAR indicator is used to augment API search results obtained from another source such as traditional Web search. The second metric, called Co-utilization API Rank (CAR), facilitates searching for a set of APIs with which an application or a mashup can be developed. This parameter is similar to WAR but ranks API sets based on how often they are used together in an application. Therefore depending on the set of APIs the developer is already using, CAR can help find other APIs that may work well with the already selected APIs.

Other discovery techniques that have been proposed by researchers include policy-centric search (De Paoli, Palmonari, Comerio, & Maurino, 2008) that search for APIs based on their policies, SLAs and other non-functional properties, and HTML micro-format based discovery (John & Rajasree, 2012) Micro-format based discovery extracts APIs/resources from HTML metadata as users browse the WWW.

Modeling and Reasoning about APIs

In addition to documenting and searching for APIs, comparing, evaluating, and verifying REST APIs is also a time consuming, manual task. Automation of these tasks attempts to capture the functionality and behavior of APIs. Using these models, it is possible to develop tools that can automatically evaluate, verify and reason about APIs. Model building and tools to reason about APIs is one of the active research areas.

In particular, Klein and Namjoshi have developed a formal model for RESTful applications using temporal logic (Klein & Namjoshi, 2011). This model exploits the statelessness and hypermedia-driven behavior (HATEOAS) of REST. It also formally describes the behavior and semantics of different HTTP

methods. More specifically, it captures the safety and idempotence of certain operations used in RESTful services. An operation is considered safe if it does not modify the resource being acted upon (e.g. GET). An operation is considered idempotent if repeated execution of the operation does not modify the resource being acted upon (e.g. PUT).

Using this formal model, Klein and Namjoshi have attempted to answer fundamental questions such as whether it is possible to automatically verify whether a given application is stateless or whether a given application adheres to its functional specification within the boundaries imposed by HATEOAS. They have outlined an automaton (a state machine) which detects violations of the hypermedia constraint and have proved that this particular verification problem is hard.

Bianchini, Antonellis and Melchiori have developed a simple tuple-based model to formally describe Web APIs (Bianchini, De Antonellis, & Melchiori, 2011). According to this model, an API is a 3-tuple of the form CAT, OP, EV, where CAT is a set of categories associated with the API, OP is a set of operations available in the API, and EV is a set of events that may occur in the API. Each operation and event can be further broken down into tuples. Based on this theoretical foundation, Bianchini et al., have defined the notion of API selection patterns. These patterns indicate how application developers typically use and engage with APIs.

Researchers also use methods that make use of semantic ontologies (Kim, Kim, & Jung, 2013), social information (Melchiori, 2011) and various other types of data to build more and more powerful models for Web APIs. Unfortunately, powerful models are very complex and difficult to process. Therefore, it is important to find the right balance between the level of detail (power) and complexity of models to develop something that is useful in practice. To date, no single method has been widely adopted, and tooling support for automated verification and evaluation of APIs is still in its infancy.

CONCLUSION

Web services are enjoying a renaissance of utility as a result of the increasing value that can be extracted from exposing digital assets (computation and data) via the Internet by a vast diversity of clients and devices. As a result, developers are increasingly employing the Web service paradigm, tooling, and the service-oriented architecture (SOA) for their distributed program development.

In this chapter, we described the composition of a Web service as architecture, abstraction, implementation, and API; and provided an overview of the two most commonly used styles for architecting Web services, SOAP and REST, and articulated how these styles impact the logical and physical design and implementation of Web service software. We discussed the strengths and weaknesses of these styles and provided evidence and reasoning behind the emergence of REST as predominant for the development of APIs and services. To compensate for the limitations of REST, key technologies have emerged to facilitate interoperability and reuse, and simplify development and API management and governance, among other features. We overviewed these technologies and described potentially high impact research efforts and open research questions associated with the next generation of REST-based service technologies.

REFERENCES

Amazon. (2013). Amazon Elastic Compute Cloud (Amazon EC2). Retrieved from http://aws.amazon.com/ec2/

Apache. (1995). The Apache HTTP Server Project. Retrieved from http://httpd.apache.org

Apache. (1999). Apache Tomcat. Retrieved from http://tomcat.apache.org

Apache. (2010). Apache Thrift. Retrieved from http://thrift.apache.org

Beyer, D., Chakrabarti, A., & Henzinger, T.A. (2005). Web service interfaces. In *Proceedings of the 14th International Conference on World Wide Web*, Chiba, Japan, pp. 148-159.

Bianchini, D., De Antonellis, V., & Melchiori, M. (2011). Semantics enabled Web APIs selection patterns. In *Proceedings of the 15th Symposium on International Database Engineering*, New York, NY, pp. 204–208.

Chavda, K.F. (2004). Anatomy of a Web service. *Journal of Computer Sciences in Colleges*, vol. 19, issue 3, pp. 124-134.

Cretella, G., & Martino, B.D. (2012). Semantic Web annotation and representation of cloud APIs. In *Proceedings of the 3rd International Conference on Emerging Intelligent Data and Web Technologies*, pp. 31-37.

Dan, A., Johnson, R.D., & Carrato, T. (2008). SOA service reuse by design. In *Proceedings of the 2nd International Workshop on System Development in SOA Environments*, Leipzig, Germany, pp. 25-28.

De Paoli, F., Palmonari, M., Comerio, M., & Maurino, A. (2008). A meta-model for non-functional property descriptions of Web services. In *Proceedings of the IEEE International Conference on Web Services*, pp. 393–400.

Django. (2005). Django. Retrieved from https://www.djangoproject.com

Fielding, R. (2000). Architectural styles and the design of network-based software architectures. *PhD thesis*. University of California, Irvine.

FOAF. (2010). FOAF Vocabulary Specification. Retrieved from http://xmlns.com/foaf/spec

Fowler, M. (2010). Richardson Maturity Model. Retrieved from http://martinfowler.com/articles/richardsonMaturityModel.html

Gomadam, K., Ranabahu, A., Nagarajan, M., Sheth, A.P., & Verma, K. (2008). A faceted classification based approach to search and rank Web APIs. In *Proceedings of the IEEE International Conference on Web Services*, pp. 177–184.

Google. (2008). Google App Engine - Google Developers. Retrieved from https://developers.google.com/appengine

Google. (2010). Protocol Buffers - Google Developers. Retrieved from https://developers.google.com/protocol-buffers

Haines, M., & Haseman, W. (2009). Service-oriented architecture adoption patterns. In *Proceedings of 42nd Hawaii International Conference on System Sciences*, pp. 1-9.

IBM. (2013). REST in the cloud. Retrieved from http://www.ibm.com/developerworks/cloud/library/cl-RESTfulAPIsincloud/

IETF. (1999). HTTP Authentication: Basic and Digest Access Authentication. Retrieved from http://www.ietf.org/rfc/rfc2617.txt

IETF. (2008). RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. Retrieved from http://tools.ietf.org/html/rfc5246

IETF. (2012a). RFC 6570 - URI Template. Retrieved from http://tools.ietf.org/html/rfc6570

IETF. (2012b). RFC 6749 - The OAuth 2.0 Authorization Framework. Retrieved from http://tools.ietf.org/html/rfc6749

IETF. (2012c). RFC 6750 - The OAuth 2.0 Authorization Framework: Bearer Token Usage. Retrieved from http://tools.ietf.org/html/rfc6750

Java. (2009). JSR-000315 Java Servlet 3.0 - Final Release. Retrieved from http://jcp.org/aboutJava/communityprocess/final/jsr315

JavaEE. (1999). Java Platform, Enterprise Edition. Retrieved from http://www.oracle.com/technetwork/java/javaee/overview/index.html

JAX-RS. (2008). JSR-000311 JAX-RS: The Java API for RESTful Web Services. Retrieved from http://jcp.org/aboutJava/communityprocess/final/jsr311

JAX-WS. (2006). JSR-000224 Java API for XML-Based Web Services 2.0. Retrieved from http://jcp.org/ aboutJava/communityprocess/final/jsr224

JBoss. (2012). JBoss Application Server 7. Retrieved from https://www.jboss.org/jbossas

JDBC. (1997). JDBC Overview. Retrieved from http://www.oracle.com/technetwork/java/ overview-141217.html

John, D., & Rajasree, M. S. (2012). A framework for the description, discovery and composition of Restful semantic Web services. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, New York, NY, pp. 88–93.

Kim, H.S., Kim, S., & Jung, W. (2013). Ontology modeling for rest open APIs and Web service Mashup method. In *Proceedings of the 2013 International Conference on Information Networking*, Washington DC, USA, pages 523–528.

Klein, M., & Bernstein, A. (2004). Toward high-precision service retrieval. *IEEE Internet Computing*, vol. 8, no. 1, pp. 30-36.

Klein, U., & Namjoshi, K.S. (2011). Formalization and automated verification of restful behavior. In *Proceedings of the 23rd international conference on Computer aided verification*, Berlin, Heidelberg, pp. 541–556. Springer-Verlag.

Kopecký, J., Gomadam, K., & Vitvar, T. (2008). hrests: An html microformat for describing restful Web services. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, Washington DC, USA, pp. 619–625.

Krintz, C. (2013). The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment. *IEEE Internet Computing*, 17(2), 72-75.

Lanthaler, M., & Gutl, C. (2011). A semantic description language for restful data services to combat semaphobia. In *Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference*, pp. 47–53.

Maleshkova, M., Pedrinaci, C., & Domingue, J. (2010). Investigating Web APIs on the world wide Web. In *Proceedings of the Eighth IEEE European Conference on Web Services*, Washington DC, USA pp. 107–114.

Maleshkova, M., Pedrinaci, C., Li, N., Kopecky, J., & Domingue, J. (2011). Lightweight semantics for automating the invocation of Web APIs. In Proceedings of 2011 IEEE International Conference on Service-oriented Computing and Applications, pp. 1-4.

Melchiori, M. (2011). Hybrid techniques for Web APIs recommendation. In *Proceedings of the 1st International Workshop on Linked Web Data Management*, New York, NY, pp. 17–23.

Microsoft. (2004). An Introduction to the Web Services Architecture and Its Specifications. Retrieved from http://msdn.microsoft.com/en-us/library/ms996441.aspx

Microsoft. (2009). Windows Azure Cloud Platform. Retrieved from http://www.windowsazure.com/en-us

Mulligan, G., & Gracanin, D. (2009). A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. *Winter Simulation Conference*, December 13-16, 2009, Austin, Texas, pp. 1423-1432.

MySQL. (1995). MySQL: The world's most popular open source database. Retrieved from http://www.mysql.com

NodeJS. (2009). node.js. Retrieved from http://nodejs.org

OASIS. (1998). OASIS: Advanced open standards for the information society. Retrieved from https://www.oasis-open.org

OASIS. (2002). OASIS Web Service Security (WSS) TC. Retrieved from https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

OASIS. (2007). WS-SecureConversation 1.3. Retrieved from http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html

Offermann, P., Hoffmann, M., & Bub, U. (2009). Benefits of SOA: Evaluation of an implementation scenario against alternative architectures. In *Proceedings of the 13th Enterprise Distributed Object Computing Conference Workshops*, pp. 352-359.

Pautasso, C., Zimmermann, O., & Leymann, F. (2008). Restful web services vs. "big" web services: making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, Beijing, China, pp. 805-814.

PHP. (1995). PHP: Hypertext Preprocessor. Retrieved from http://php.net

Parastatidis, S., Webber, J., & Robinson, I. (2010). *REST in Practice: Hypermedia and Systems Architecture.* O'Reilly Media.

Phan, K.A., Tari, Z., & Bertok, P. (2006). A benchmark for SOAP's transport protocols performance for mobile applications. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, Dijon, France, pp. 1139-1144.

Potti, P. K. (2011). On the design of web services: SOAP vs. REST. *UNF Theses and Dissertations*, paper 138.

ProgrammableWeb. (2013). ProgrammableWeb - Mashups, APIs, and the Web as a Platform. Retrieved from http: //www.programmableWeb.com

Rackspace. (2013). App, Email, SharePoint & Web Hosting on Cloud & Bare Metal. Retrieved from http://www.rackspace.com

Richardson, L. (2008). The Maturity Heuristic. Retrieved from http://www.crummy.com/writing/speaking/2008-QCon/act3.html

Schepers, T.G.J., Iacob, M.E., & Van Eck, P.A.T., (2008). A lifecycle approach to soa governance. In *Proceedings of the ACM Symposium on Applied Computing*, New York, NY, pp. 1055–1061.

Sinatra. (2007). Sinatra. Retrieved from http://www.sinatrarb.com

Swagger. (2010). A simple, open standard for describing REST APIs with JSON. Retrieved from https://developers.helloreverb.com/swagger

Torres, R., Tapia, B., & Astudillo, H. (2011). Improving Web API discovery by leveraging social information. In IEEE International Conference on Web Services, pp. 744–745.

UDDI. (2004). UDDI Version 3.0.2. Retrieved from http://www.uddi.org/pubs/uddi_v3.htm

Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Gabarró, J., & Van de Walle, R. (2012). Functional descriptions as the bridge between hypermedia APIs and the semantic Web. In Proceedings of the Third International Workshop on RESTful Design, New York, NY, pp. 33–40.

W3C. (1994). World Wide Web Consortium (W3C). Retrieved from http://www.w3.org

W3C. (1999). Hypertext Transfer Protocol – HTTP 1.1. Retrieved from http://www.w3.org/ Protocols/rfc2616/rfc2616.html

W3C, (2001). W3C Technical Architecture Group (TAG). Retrieved from http://www.w3.org/2001/ tag

W3C. (2004a). OWL Web Ontology Language Overview. Retrieved from http://www.w3.org/TR/owl-features

W3C, (2004b). Web Services Architecture. Retrieved from http://www.w3.org/TR/ws-arch

W3C. (2009). Web Application Description Language. Retrieved from http://www.w3.org/ Submission/wadl

W3C. (2011). Notation3 (N3): A readable RDF syntax. Retrieved from http://www.w3.org/ TeamSubmission/n3

Wordnik. (2009). Retrieved from http://www.wordnik.com

ADDITIONAL READING

Davis, A., & Zhang, D. (2002). A comparative study of dcom and soap. In *Proceedings of the Fourth IEEE International Symposium on Multimedia Software Engineering*, Washington DC, USA.

Dzone. (2012). Different SOAP encoding styles - RPC, RPC literal and document literal. Retrieved from http://java.dzone.com/articles/ different-soap-encoding-styles

Dissanaike, S., Wijkman, P., & Wijkman, M. (2004). Utilizing xml-rpc or soap on an embedded system. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pp. 438–440.

Looker, N., & Xu, J. (2003). Assessing the dependability of soap rpc-based Web services by fault injection. In, 2003. In *Proceedings of the Ninth IEEE International Workshop on WORDS: Object-Oriented Real-Time Dependable Systems*, pp. 163–170.