

Service interoperability in the Internet of Things

José C. Delgado

Abstract The main service interoperability models (SOA and REST) are currently implemented in the Web with text based technologies (XML, JSON, HTTP), conceived for large grained hypermedia documents. Their extension to the Internet of Things context, involving devices with constrained capabilities and unreliable wireless network protocols, implies using a subset of the features of those technologies and adapting the network and message level protocols. This chapter starts by establishing a layered interoperability framework, from the organizational down to the network protocol levels. Then, it assesses the constraints and limitations of current technologies, establishing goals to solve these problems. Finally, a new interoperability technology is presented, based on a distributed programming language (and its execution platform) that combines platform independence and self-description capabilities, which current data description languages exhibit, with behavior description (not just data), elimination of the need of a separate language for schema or interface description, complete separation of data and metadata (optimizing message transactions) and native support for binary data (eliminating the need for encoding or compression).

Introduction

Aside older technologies, the main solutions currently available for service and resource interoperability are the SOA (with Web Services) and REST styles, both usually over HTTP. These are the product of an evolutionary line stemming directly from the early days of the Web, where the distinction between a client and a server was clear, stateless browsing and scalability were primary objectives and the hypermedia document was the interaction unit. That was the original Web, or the *Web of Documents* [1]. Today, the world is rather different:

- People evolved from mere browsing and information retrieval to first class players, either by actively participating in business workflows (*Business Web*) or engaging in leisure and social activities (*Social Web*);
- The service paradigm became widespread, in which each resource (electronic or human) can both consume and provide services, interacting in a global *Service Web*;
- Platforms are now virtualized, dynamic, elastic and adaptable, cloud-style, with mobility and resource migration playing a role with ever increasing relevance;

- The distributed system landscape has expanded dramatically, both in scale (with a massive number of interconnected nodes) and granularity (nodes are now very small embedded computers), paving the way for the *Internet of Things* (IoT) [2, 3].

The usual meaning of *Web* implies **HTTP based systems**, whereas *Internet* means **IP based systems**. In this chapter, we use *Internet* in a more general sense of *network interconnecting networks*, even if these are not IP based, to cater for the low level devices (such as sensors) and the networks that interconnect them.

Figure 1 establishes two main approaches to achieve the IoT:

- **Top-down**, by extending current HTTP based technologies to a level as low as the devices' capabilities allow (which is known as *Web of Things* [4], or WoT), with gateways that represent the functionality provided by lower level devices, not HTTP or not even TCP/IP enabled. This is the mainstream solution today;
- **Bottom-up**, by rethinking the interoperability problem in the envisaged scenario of IoT, not just the Web, and deriving a solution that works in the entire IoT, not just the WoT. The goal is to seamlessly integrate distributed platforms, applications and services, covering the ground from higher down to lower level devices and small footprint applications, as well as providing native support for binary data and asynchronous event processing, without the need to use and interconnect complex technologies. This is the solution proposed by this chapter.

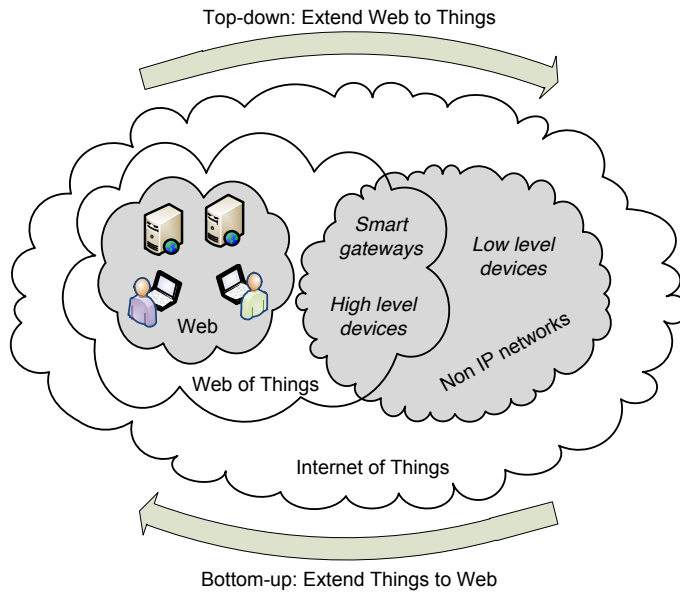


Fig. 1. Approaches to integrate low level devices with people and computers.

This chapter is organized as follows:

- First, an interoperability framework is proposed, catering for several levels of interoperability and identifying those dealt with by this chapter;
- Next, the current interoperability technologies at these levels are assessed and their limitations discussed;
- Finally, with the goal of solving the problems previously identified, a language (SIL – Service Implementation Language) and its development and execution environments are proposed.

Background

Pervasive computing [5] has been made possible by the ever decreasing cost of microcontrollers. Many of these need to be connected in a network, to generate events that control many applications, as it happens in the case of sensors or readers of RFID tags [6]. Since many applications today are web based, it is only natural to conceive that low level devices can interact directly with conventional web actors, such as people and applications running in servers. All these interconnected entities form what is usually known as the Internet of Things (IoT) [3], as expressed in Figure 1, with the Web of Things (WoT) [4] as the subset that uses HTTP as the underlying message protocol.

The conventional HTTP based solutions for distributed interoperability in the context of Web are the SOA (with Web Services) [7] and REST [8] architectural styles. These are technologies conceived for reliable TCP/IP networks and a reasonable large granularity, with hypermedia documents as the interacting unit and a synchronous, stateless protocol (HTTP).

There is an evident mismatch between this setting and the IoT environment, in which:

- The granularity can be much smaller, with low level devices and very simple messages;
- Communication is more peer to peer (with wireless oriented protocols) than many clients to one server (with omnipresent connectivity);
- The limitations of devices, in terms on power consumption, processing power and memory capacity, have a decisive influence.

Nevertheless, the tendency has been to adapt existing technologies to the IoT context, mainly in the following fronts:

- At the service level in the Web, there is a debate on whether to use SOA or REST [9, 10, 11]. REST is clearly simpler to use as long as applications are not complex, and thus a better match for the very simple APIs found in IoT applications [4, 12, 13, 14]. Nevertheless, there are also SOA applications for the IoT [15, 16]. The DPWS (Devices Profile for Web Services) standard supports

a lightweight version of the Web Services stack on resource-constrained devices [17];

- The serialization formats used in the Web (e.g., XML, JSON) are text-based and thus verbose and costly in communications. Technologies have been developed to compress text-based documents [18], such as EXI (Efficient XML Interchange) and others [19]. However, these are compression technologies, which need text parsing after decompression. Recent native binary serialization formats, such as Protocol Buffers and Thrift [20], do not have this problem;
- There are several efforts targeted at endowing simple devices with IP capability, in particular in the wireless realm and even in IPv6 [21, 22]. The IETF 6lowpan working group has produced a specification (currently a standard proposal) for IPv6 support on Low power Wireless Personal Area Networks (6LoWPAN) [23], such as those based on the IEEE 802.15.4 standard [24].

Simplicity and efficiency (which translates to low processing requirements and promotes scalability, when needed) seem to be the driving forces today, behind not only the IoT and the WoT but even the Web in general as well. This has fueled the generalized adoption of the REST style [9], except for complex, enterprise class of applications, under the assumption that it reduces coupling with respect to the RPC style that is common in SOA based applications [8]. However, interoperability needs to be considered at several levels, as discussed in [25, 26], which is the basis for the interoperability framework described in this chapter and the reasoning that leads to a different conclusion.

Building on the simplicity of REST, the CoAP (Constrained Application Protocol) [27] is a specification of the IETF working group CoRE, which deals with constrained RESTful environments. CoAP includes only a subset of the features of HTTP, but adds asynchronous messages, binary headers and UDP binding.

Compatibility is both a bonus and a curse. All the adaptations towards supporting Web technologies in lower level devices and wireless networks specify subsets of and changes to those technologies. In the end, it is questionable what is gained by this partial compatibility (which ends up demanding changes to middleware and applications) and what is lost by not designing a model that contemplates from scratch (by design) the needs of the modern distributed applications, including the IoT. This is the underlying thought that constitutes the background scenario and motivation for this chapter.

An interoperability framework

In a distributed world, there is no direct intervention from one resource on another. The only way for a resource to interact with another one is to send it a message, with the requirement that there is a channel interconnecting them. This channel can involve one or more networks, as illustrated by Fig. 1.



In this context, we make the following informal definitions:

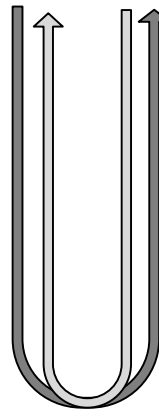
- A *resource* is an entity of any nature (material, virtual, conceptual, noun, action, and so on) that embodies a meaningful, complete and discrete concept, making sense by itself while being distinguishable from, and able to interact with, other entities. This means that each resource must have a unique way to identify it (such as a URI);
- A *service* is the interface of a resource, expressed by the set of messages that can accept and react to. A resource implements its service. Resources are the entities that actually exchange messages. However, since a service is the visible face of a resource, saying that a service sends a message to another (or mentioning *service interoperability*) is a common and acceptable language liberty;
- A *transaction*, in this context, is the set of actions and effects resulting from:
 1. Sending a request message;
 2. Reaction from the resource that receives it;
 3. Sending back a response message;
 4. Reaction to that response by the resource that has sent the request.

The resources involved in a transaction perform the roles of *consumer* (sends the request) and *provider* (executes the request and sends the response). In another transaction, the roles can be reversed.

Achieving interoperability between two resources is much more than sending a set of bytes as a message. The interacting resources need to agree on compatible protocols, formats, meanings and purposes, so that a request message sent by one resource produces the intended effects on the other resource and a suitable response message is returned to the former resource.

Table 1 establishes a framework for interoperability by defining a set of layers of conceptual abstraction of interoperability, from very high level (the strategies of both resources need to be aligned so that each message fits a common purpose) down to very low level (the message must physically reach the other resource).

Table 1. Levels of interoperability in a simple transaction.  - Request.  - Response.

Concept	Consumer	Channel	Provider	Interoperability
Alignment	Strategy		Strategy	Organizational (purpose)
Cooperation	Partnership		Partnership	
Outsourcing	Value chain		Value chain	
Ontology	Domain		Domain	Semantic (meaning)
Knowledge	Rules		Rules	
Contract	Choreography		Choreography	
Interface	Service		Service	Syntactic (notation)
Structure	Schema		Schema	
Serialization	Message format		Message format	
Interaction	Message protocol		Message protocol	Connectivity (protocol)
Routing	Gateway		Gateway	
Communication	Network protocol		Network protocol	

A consumer or a provider can be a very complex resource, such as an enterprise, or a very simple one, such as a temperature sensor. Naturally, the higher interoperability levels are more relevant for the more complex resources, but even a mere sensor has a purpose and implements a (very simple) strategy that must fit and be aligned with the strategy of the system that uses it.

We should also bear in mind that the IoT entails not really just interconnecting physical devices but rather fitting them into the overall picture, up to the organizational level. A simple RFID tag can be fundamental in supporting the automation of business processes.

The interoperability levels considered in Table 1 result from a refinement and adaptation to the IoT context of the frameworks described in [25, 26] and can be organized, from top to bottom, in the following categories:

- **Organizational.** Each request from a consumer to a provider must **obey a master plan** (the strategy). The consumer must know why it is placing that request to that provider, which must be willing to accept and to honor it. Both strategies need to be aligned and implemented by a cooperation supported by complementary activities (outsourcing), as part of a value chain. A key concept at these levels is Enterprise Architecture [28];
- **Semantic.** Both interacting resources must be able to **understand the meaning of the data exchanged and the reaction of the other resource to each message sent**. This means compatibility in ontology, rules and workflows, so that these resources can participate with matching roles in some choreography;
- **Syntactic.** This category deals mainly with form, rather than content. An interface needs to be defined for each resource, exposing some structure, both at the data and behavior levels, according to some schema. WSDL and REST APIs

[29] are examples. The contents of messages need to be serialized to be sent over the channel and the format to do so (such as XML or JSON) also belongs to this category;

- **Connectivity.** The main objective in this category is to transfer the message's content from one resource to the other. This usually involves enclosing that content in another message with control information, implementing a message protocol (such as HTTP), which is then sent over a communications network, according to its own protocol (such as TCP/IP or ZigBee [30]). When different networks (eventually, with different protocols) or network segments are involved or proxies are needed, gateways are used to implement routing and/or protocol adaptation [31].

Each transaction must satisfy interoperability at all levels, both in the request and the response. In Table 1, interoperability at each level (horizontally, in both consumer and producer) abstracts lower levels and ignores higher levels. For example, interoperability at the service level assumes that the correct service is targeted, the correct operation is invoked with adequate parameters and that operation produces the expected result. This ignores the semantics and purpose of the transaction (which must be dealt with at higher levels) and relies on compatible formats and protocols for communication (ensured by lower levels).

In practice, most existing systems specify and implement interoperability in the syntactic and connectivity categories in detail, dealing with the organizational and semantic categories essentially at the documentation level. This is particularly true in the context of the IoT, in which many of the interconnected resources are low level devices, with limited capabilities and intelligence, and consumers and providers can reside in different networks, with different protocols, as shown in Fig. 1. For this reason, and to limit the complexity involved, this chapter deals essentially with the syntactic and connectivity categories, from the service interoperability level downwards.

Fig. 2 illustrates the main steps involved when a consumer in one network performs a transaction with a provider in another network, with a gateway to bridge the two. This could be, for example, the case of a computer application establishing a dialog with a smart phone, reading information from a sensor or controlling some device in a Bluetooth or ZigBee network.

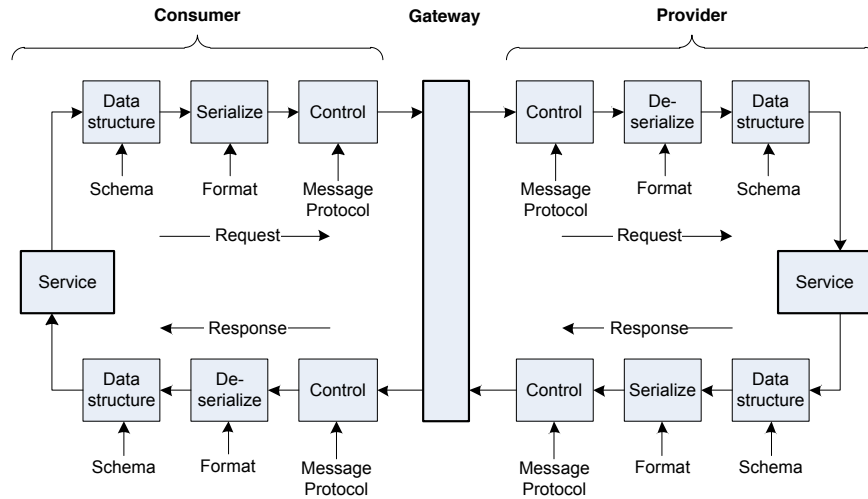


Fig. 2. Main steps involved in a transaction between two services.

Sending a request message from the consumer to the provider (top part of Figure 2) entails the following main steps:

- At the consumer:
 - Build the request data structure, according to the request schema;
 - Serialize it, using a format (e.g., XML) that supports that schema;
 - Enclose it as a payload in a suitable message level protocol (e.g., HTTP);
 - Send the complete message;
- At the gateway:
 - Convert protocol information (if needed);
 - Convert the message payload (if needed);
 - Route the message to the provider's network;
- At the provider:
 - Recover the message payload, using the provider's protocol;
 - Rebuild the request data structure, using the provider's schema;
 - Check if the message payload can be accepted;

These steps correspond to the U shape in Table 1 and show how to make services interoperable we need to ensure compatibility at the schema, serialization format and message and network protocols. The trivial solution is to use the same specification at each level (for example, XML reader and writer must use the same schema), but this translates to coupling, an undesirable constraint in distributed systems. This chapter shows how to alleviate this problem.

The third part of the transaction, sending a response to the consumer, is similar to sending the request to the provider, but now the message flows the other way

(bottom part of Figure 2). Note that schemas, formats and protocols may not be the same or use different rules with respect to the first part of the transaction (sending the request).

Assuming that the gateway performs its role as a bridge between network protocols, if needed, making the consumer and provider interoperable in syntactic and connectivity terms means solving the interoperability problem at the levels of service and below, as shown in Table 1. These are the levels used in this chapter to discuss interoperability.



Assessing current interoperability technologies

We will now use the interoperability framework described above to discuss the suitability and limitations of the main existing solutions to support the IoT. We do not assess the levels below message protocol because we aim at providing a solution that works across heterogeneous networks, with different network protocols.

Service interoperability

Naturally, resources are free to use whatever conventions they see fit to ensure interoperability at the service (interface) level. However, application specific solutions should be avoided. In the context of IoT, the two most used models at this level are SOA and REST. Although these are abstract models, in practice only the two most common instantiations, SOA with Web Services and REST with HTTP, are considered. Therefore, the SOA and REST acronyms in this chapter refer to these instantiations.

We will use an example to make the differences between SOA and REST clearer. Figure 3 describes a typical application in the IoT, involving electronic commerce and logistics [6], with a set of processes involved in purchasing and delivering a product. The customer orders and pays (with an option to cancel) a product at the web site of the seller, which sends the product to a distributor via a transport company. The customer can track the evolution of the product during transport, thanks to information from an RFID tag in the product's parcel. When the parcel passes by a RFID sensor (in a truck or distribution station), a signal is sent to the business process at the seller company.

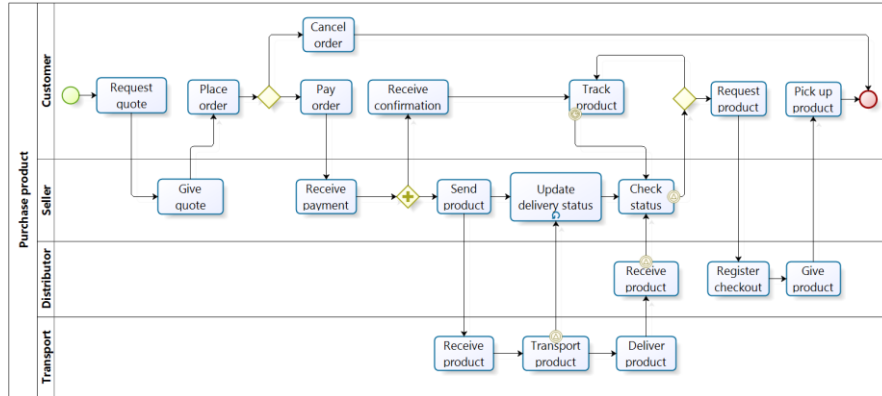


Fig. 3. An example of an application in the IoT.

The essence of SOA

SOA leads to an architectural style that is an evolution of the RPC (Remote Procedure Call) style, by declaring and invoking operations in a standard and language independent way. Compared to RPC, coupling has been reduced and interoperability increased. However, the following principles still apply:

- The resources that model the problem domain are essentially those that correspond to nouns;
- Web Services model each of these resources, exposing a set of operations (its interface) that implement its behavior;
- Only behavior structure (operations) is exposed by the service. State structure (data) is private, implementation dependent and not externally accessible;
- Resources are peers, in the sense that each resource can both offer a service and invoke other resource's service;
- To use a resource, its service description (i.e., WSDL document) must be obtained first, so that a contract can be established between that resource (the provider) and the resource that uses it (the consumer). That contract is static (design time).

Figure 4 partially illustrates the SOA solution to the problem of Figure 3, by describing the application from the viewpoint of the customer (how it progresses and interacts with the seller and the distributor companies). Each of the interacting entities is modeled as a resource, defining a Web Service with operations as needed to support the corresponding functionality. The resulting processes will perform a choreography, as shown in the sequence diagram.

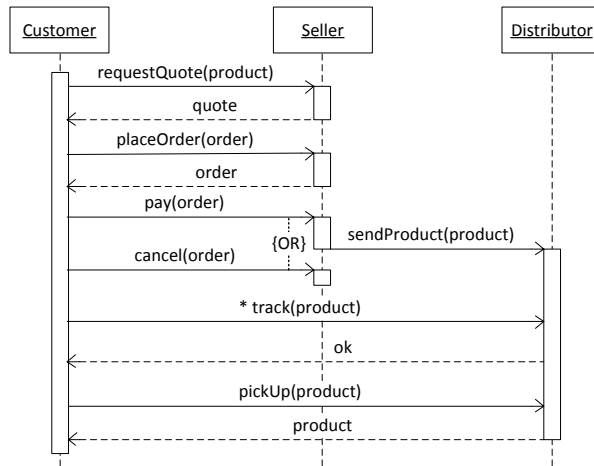


Fig. 4. Interaction of the customer with the seller and distributor, SOA style.

The essence of REST

A contract between resources, which is an expression of their coupling, is seen as a strong disadvantage by REST proponents, which contend that a more dynamic approach reduces complexity and coupling, while promoting changeability, scalability and performance [9, 32].

REST is an architectural style that essentially defines a set of best practices of HTTP usage. The most relevant principles, expressed as architectural constraints [33], are the following:

- **Client-server dichotomy.** There is a clear distinction between client and server roles, with the implicit assumptions that clients are the only ones allowed to take the initiative of sending requests and usually greatly outnumber servers. A response from a server resource is considered a representation of that resource and not a resource in itself;
- **Stateless interactions.** Each request must include all the information needed for the server to process it. Servers do not store session state, which is only maintained by the client, which means that, in each response to a request, the server must return all the information that may be needed for the next request;
- **Uniform interface.** This means having the same set of operations (with GET, PUT, POST and DELETE as the most common) for all resources, albeit the behavior of operations can differ from resource to resource. Therefore, at the syntactic level, all resources in REST implement the same service (interface);

- **Explicit cacheability.** All responses by a server must define whether they can be cached at the client or not. If yes, they can be reused in future equivalent requests, which improves performance and scalability.

The REST style has been inspired by the class of applications that motivated the conception of the Web, in which typically there are many clients accessing a server and scalability is of paramount importance. This justifies the stateless interaction and cacheability constraints, since the load on the server becomes less dependent on the number of clients.

However, the most distinguishing constraint of REST is the uniform interface, which breaks the dichotomy between nouns (objects) and verbs (operations) that is typical of SOA modeling, which stems from object-oriented modeling (usually expressed in UML), in which a specific set of verbs is used to describe a noun concept. In REST, there is only one structuring dimension (resource composition) and any operation that does not fit the semantics of the four basic HTTP verbs is modeled as a resource defined inside another resource.

The rationale for the uniform interface is a logical consequence of the stateless interaction. If the state of the interaction is exchanged between the client and the server in each request-response, what to do next by the client depends on that state and the possible state transition paths, leading to a state machine processing style to model behavior. Each server response is equivalent to a *closure* [34] that includes the necessary information (including links to other resources) to change to the next application state.

This means that a client should not rely on previous knowledge about the server's resource structure, so that changes in the application can automatically be used by the client without changing it. The client needs only to know how to change state, by pursuing links and using the uniform interface. It is the resource representations retrieved from the server that present the states to which the client can go next.

This is what is usually known as HATEOAS (Hypermedia As The Engine Of Application State) [33, 35], in which the client analyzes the server's response and typically proceeds by sending a request through one of the links contained in it, leading to a state diagram traversal which overall corresponds to a process, as Figure 5 illustrates.

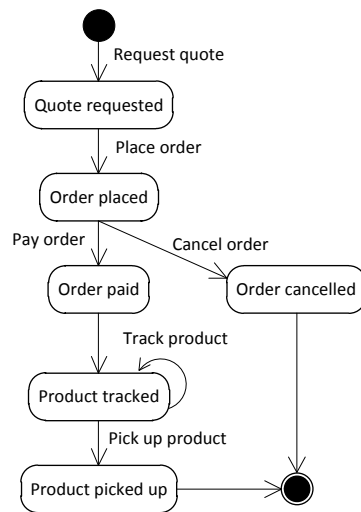


Fig. 5. Interaction of the customer with the seller and distributor, REST style.

This is the REST solution to the customer interaction with the other actors of the system in Figure 3, equivalent to the SOA solution of Figure 4. Resources corresponding to actions (such as placing a purchase order) are created at the server and links to them (such as <http://company.com/order/product123>) are returned in a response to the client. Following these links has the effect of executing the corresponding actions. Such resources are created dynamically, as the state machine progresses. The different actions in Figure 5 are mapped onto a unique set of operations executed on different resources.

Assessing SOA and REST

The main idea behind the uniform interface in REST is to separate the mechanism of traversing a state graph (such as the one in Figure 5) from the individual processing actions of each state. If all states were externally identical, then all state transitions would obey the same rules (HTTP verbs) and the HATEOAS at the client would automatically adapt to changes in the application stored at the server. In other words, the contract binding client and server would not exist or, better said, would be universal and not application specific, with a much smaller coupling than a Web Service contract.

The client-server interaction, however, does not always work in this way and REST cannot ensure this minimal coupling. In practice, the client needs to understand the purpose, meaning and notation (Table 1) of all the responses the server may return, so that it can determine which state to go to next. This means that:

- A generic media type such as XML or JSON is not enough. A concrete schema, with the names used, must be specified and be compatible on both sides, otherwise a representation returned by a resource, for example, will not be able to be analyzed and understood. This is why REST favors standardized media types. However, this hampers variability, which is particularly relevant in the IoT context, given the plethora of interconnected devices. What happens frequently is that a generic media type is used with out-of-band knowledge about the specific schema (implicit information, which can break in case of changes);
- The decision of which link to choose to go to the next state depends on the higher interoperability levels. A person using a browser resorts to additional page information (text, pictures, forms, and so on) to choose a link to click and can even backtrack to follow a different path, if needed. Client applications are less intelligent. In any case, out-of-band information is always needed. The idea that in REST the design-time client-server contract is limited to the data level semantics provided by standardized media types is a mere illusion. This is a direct consequence of dealing with complexity by specifying explicitly only some of the lower interoperability aspects and implicitly assuming the others through names and documentation (informally used to express meaning, subject to misinterpretations);
- REST has a lower resource granularity than SOA, given that internal data and operations in SOA are converted into resources in REST (behavior structure is converted into resource structure). These are simpler and have smaller contracts, but the number of different resource types is higher and the structure of links needs to be known to some degree. The overall application contract (client to server coupling) cannot be simpler than the problem itself requires, independently of using a SOA or REST solution.

SOA exhibits a small modeling semantic gap, since interacting entities at the problem level are mapped one-to-one onto resources, but data resource structure is not supported. Interaction contracts are static (design time) and as complex as the functionality of each service requires.

REST has a greater semantic gap, since resources are at a lower level, but supports data resource structure and contracts are simpler and dynamic (although more numerous).

In fact, SOA and REST have dual models, constituting two complementary ways of solving a given problem. Comparing Figures 4 and 5, we can notice that, apart from process specific details, the flow of activities of the client in one figure is dual of the other's. SOA uses activity based flow, with activities in the nodes and state in between, whereas REST makes transitions from state to state, implemented by the activities in between.

SOA is guided by behavior and REST by (representation of) state. In UML terminology, SOA uses a static class diagram as a first approach, to specify which resource types are used and to establish the set of operations provided by each, whereas REST starts with a state diagram, without relevant concern about distinguishing which state belongs to which resource. In the end, they perform the same

activities and go through the same states. This should not be a surprise since, after all, the original problem, described by Figure 3, is the same.

There are IoT applications based on SOA [15, 16], but the RESTful approach seems to be more popular, in particular in the lower range (applications with simple devices) [12, 13, 14]. The main reason for this preference is not really caching for performance (data is constantly changing), nor stateless interaction for scalability (interaction tends to be more local and peer to peer than many clients accessing a single device), nor even a preference for state diagrams over processes for modeling (many devices do not even support multistate interaction, only isolated requests). It seems that the preference for REST is based on plain simplicity. Using basic HTTP is much simpler than using Web Services and even link usage is kept to a minimum. In fact, in typical systems such as those described in [4, 12], the only representations returned with links are the list of devices. In other words, there is only application data structure exposed, not behavior. This is essentially a CRUD (Create, Read, Update and Delete) application [35], rudimentary REST at best.

SOA and REST have different advantages, tradeoffs and constraints, summarized in Table 2, which makes them more suited to different classes of applications. REST is a better match for Web style applications (many clients, stateless interactions) that provide a simpler interface. This is why all the major Internet application providers, including cloud computing service providers, now have REST interfaces. SOA can be a better choice for functionally complex, enterprise level distributed applications, in which the semantic gap becomes prominent.

The ideal would be not having to choose between one or the other, by combining the best of both. Support for portable code would also be desirable, to support code migration. Currently, services need to be implemented in general programming languages or BPEL [38]. The desirable features are expressed in the *Wish list (SIL)* column, which serves as the requirements for a language (SIL) that combines the best of SOA and REST, reaping the benefits while avoiding the limitations inherent to each technology. SIL is described below, in section *Rethinking interoperability*.

Table 2. Comparing characteristics at the service level

Characteristic	SOA	REST	Wish list (SIL)
Semantic gap	Low	Higher	Low
User defined interface	Yes	No	Yes
Contract	Static	Dynamic	Both
Changeability	Low	High	High
Design time support	High	Low	High
Granularity	High	Low	Variable
Complexity	High	Low	Low
Best for applications	Complex, event based	Simple, scalable	All
Exposed structure	Behavior	State and behavior	State and behavior

Platform agnostic code	No	No	Yes
Execution paradigm	Workflow	State machine	Both
Links as closures	No	Yes	Yes

Schema interoperability

Resources send requests to each other to invoke a given operation, be it with a SOA or REST approach. These requests and their responses usually contain data, which are serialized, sent and reconstructed upon reception of the corresponding message (Figure 2). The sender and receiver need to interpret those data in a compatible way, which means interoperability at the schema level between the corresponding data structures at both ends.

In many cases, the schema information is limited to the indication of a standardized media type, with specific data component names as out-of-band information. This is particularly true in the IoT, in which JSON is a very popular serialization format, as a simpler alternative to XML. JSON Schema is currently just an IETF draft [36], but is raising interest as a simpler alternative to XML Schema.

XML Schema and JSON Schema share many of their goals and characteristics, as expressed by Table 3. The fact is that even JSON Schema can be too much for the IoT, since usually RESTful APIs [29] just specify JSON as the media type, assuming that concrete component names have been agreed between client and server. This provides little support for interface verification when designing services. We need a better way of specifying schema level interoperability, particularly in the context of the IoT. The requirements to do so are expressed by the *Wish list (SIL)* column.

Table 3. Comparing characteristics at the schema level

Characteristic	XML Schema	JSON Schema	Wish list (SIL)
Separate document	Yes	Yes	Yes
Separate specification	Yes	Yes	No
Same syntax as data language	Yes	Yes	Yes
Compatibility	Sharing	Sharing	Compliance
Compatibility cache	No	No	Yes
Schema compiler	No	No	Yes
Validation	Yes	Yes	Yes
Data binding	Yes	Yes	Yes
Complexity	High	Low	Low
Verbosity/size	High	Medium	Low
Reference format	URI string	URI string	Agnostic
Dynamic link construction	Yes	Yes	Yes

There are several ways to improve the schema level interoperability, by solving the main limitations of XML Schema and JSON Schema, such as:

- Do not require both writer and reader of a document to use the same schema, because this requires interoperability for all the documents that satisfy the schema, instead of just the documents that need to be exchanged. Use *structural interoperability (compliance and conformance)* [37] instead, as a way to reduce coupling and to improve adaptability. The basic idea is to check whether a specific document (not all the documents satisfying a schema) is complies with the schema requirements of the service that will read that document. This is done structurally and recursively, component by component. If it does, the document (a message) can be accepted by the receiver. This is different from what XML does, which requires that both interlocutors use the same schema;
- A schema document, separate from the data document, is always a huge source of complexity. Although simpler, JSON Schema seems to be following XML Schema's footsteps, which is very revealing of the need for better design time support. However, we can automatically generate a document's schema from its data, which, in consonance with structural interoperability, allow us to get rid of specific schema languages altogether;
- XML Schema and JSON Schema are valid XML and JSON documents, respectively. This means that they suffer from the same verbosity and parsing inefficiencies that text based serialization formats exhibit. It is better to use a schema compiler [20] than a mere compression mechanism [18, 19]. A schema compiler produces binary information that can be used to implement compliance efficiently, both in terms of processing time and size of exchanged information;
- Use a cache-based mechanism to avoid repeating the compatibility checks in every message, if there are no schema changes;
- Do not restrict resource references to URIs, to encompass non TCP/IP based networks, in which nodes are not identified by URIs. A reference should include a link as a primitive data type, with a format adequate to the network of the target resource. Multi-network references should be supported (including ZigBee addresses, for example).

Serialization format

A schema is used to transform the internal data structures into serial messages and vice-versa, as shown in Figure 2. Two of the most common serialization formats are XML and JSON. Both are text based and support schemas. XML retains the look and feel of HTML, with text markup based on start and end tags, whereas JSON delimits data with syntax tokens, with a simple grammar that bears some similarity with data structures in C-like languages. XML has targeted flexibility

and generality, whereas JSON has emphasized simplicity, which is after all the secret of its popularity.

In spite of the differences, both suffer from drawbacks and limitations that are particularly relevant in the context of IOT:

- They are text based, which means inefficient parsing and data traversal (all characters of a component need to be parsed to reach the next one), high memory consumption, relevant message transmission times and poor support for binary data. The serialization format should be as close as possible to the data structure, in Figure 2, to minimize the conversion effort in serialization and deserialization. Text has been touted as human readable and therefore advantageous over binary, but this is true only for very simple documents, especially when using XML. Binary compression mechanisms exist [18, 19], but this does not reduce the parsing time, since text is recovered. It would be better to follow the ancient example of programming languages, by using a source format for humans, a binary format for computers and a compiler to synchronize them;
- Metadata is partially interspersed with data, in the form of element/attribute names and tags (in XML). This is redundant with respect to schema information and adds overheads to parsing and transmission times. Data and metadata should be completely separate, much like HTML evolved into XML by separating content from format. This allows to optimize recurring messages, with the same metadata, by just sending the metadata once, caching it with some ID and using that ID in subsequent messages;
- Only data types can be serialized. There is no support for serializing behavior elements (instructions and operations). Describing these in a XML based syntax yields a very cumbersome syntax, such as in the case of BPEL [38]. Incorporating a set of primitive behavior elements and their structuring constructs, at the image of what happens with data elements, is a simple and clear solution of completing message semantics and supporting code migration.

Table 4 summarizes the basic characteristics of XML and JSON as serialization formats, as well as the improvements sought, in the *Wish list (SIL)* column.

Table 4. Comparing characteristics at the serialization format level

Characteristic	XML	JSON	Wish list (SIL)
Data compiler	No	No	Yes
Baseline data	Text	Text	Text, binary
Data delimiters	Markup	Grammar	Grammar, binary tags
Metadata/data separation	Partial	Partial	Complete
Compliance optimization	No	No	Yes
Native binary support	No	No	Yes
Code element types	No	No	Yes

Message protocol

In Figure 2, serialized messages need to be sent to the interlocutor, using a message level protocol that adds control information to the message's content.

RESTful applications simply use HTTP, although this protocol has grown to include features all the way up to the service level. It is not a generic message interoperability protocol, since it has been developed specifically for the Web context and with the REST principles (namely, client-server dichotomy, stateless interactions and the uniform interface) as the support for client scalability. This is not the best match for the IoT scenario, which is more based on asynchronous event processing and peer interactions.

HTTP lacks extensibility, headers simply follow a text format instead of a structured format (e.g., XML or JSON), does not support binary data (only text encoded), asynchronous messages, session based interaction for recurring messages, server side initiated requests or client notifications, and follow a strict pattern in message interactions, with a fixed set of operations and of response status codes. HTTP is truly specific for the Web, but with a crucial importance that stems directly from its simplicity, the ubiquity of the Web and the *de facto* generalized friendliness of firewalls.

SOAP, used in conjunction with Web Services, does not incur some of the problems of HTTP but introduces some of its own, namely complexity. Being XML Schema based, is inefficient in every message without benefiting from the flexibility and variability that a schema would allow. SOAP is a standard and does not change frequently. The most common case is to have SOAP over HTTP, but mainly as a transport level protocol, which is another source of inefficiencies.

The message protocol is another level in which substantial improvements can be made. This is expressed in the *Wish list (SIL)* column of Table 5, which also summarizes the main characteristics of HTTP and SOAP.

Table 5. Characteristics of message level protocols

Characteristic	HTTP	SOAP	Wish list (SIL)
Baseline format	Text	Text	Binary
Message is	Character stream	Character stream	Byte stream
Structured headers	No	Yes	Yes
Layered headers	No	Yes	Yes
Schema based	No	Yes	No
Generic operation call	No	Yes	Yes
Generic response status	No	Yes	Yes
Message transaction	Synchronous	Synchronous	Asynchronous
Reaction messages	No	No	Yes
Heterogeneous network support	No	No	Yes

We would like to ally the simplicity of HTTP with the capabilities of SOAP and to throw in some additional features, such as:

- Native support for binary data. This implies using a byte stream as the rock bottom transport format, not a character stream (text). A bit stream is also possible and more compact [19, 42], but requires more processing effort to encode and decode, which is relevant in small IoT devices with low processing power;
- Native support for asynchronous messages and responses, with additional information at the MEP (Message Exchange Pattern) level;
- Reaction messages, sent to a resource without specifying any operation and letting that resource react as it sees fit, by automatically selecting the appropriate operation, based on the type of the message;
- Support for messages spanning heterogeneous networks, with different protocols such as TCP/IP, ZigBee or Bluetooth, without requiring message format conversions in gateways. Essentially, this involves not being dependent on the protocol, even at the level of identification of the resources that messages are addressed to. A link used to identify a resource identification may be no longer a single URI, but rather an inter-network path (a list of URIs or even of other identifiers, such as a ZigBee address).

Rethinking interoperability

An IoT case study scenario

We envisage a typical scenario of the IoT, depicted in Figure 6a, in which a client application accesses sensors coordinated by a controller and interconnected by a sensor network. In a logical view, the sensors are components of the controller, as shown in Figure 6b, assuming that could be other controllers, coordinating their own set of sensors.

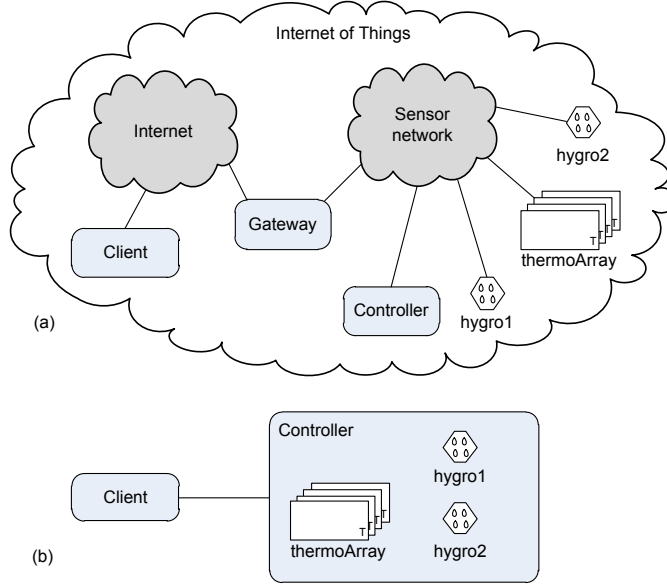


Fig. 6. A client accessing devices in a sensor network. (a) Network view. (b) Logical view.

The usual solutions to implement this scenario would be:

1. To provide an API (REST or SOA) in the Controller, which makes a mashup of the functionality of the sensors [39], hiding them from the client;
2. To endow sensors with TCP/IP and HTTP capability and to implement a REST or SOA service in them [40], so that they can be used directly by the client.

Both solutions have drawbacks:

1. The controller needs to reproduce the functionality of sensors and needs to be changed whenever unforeseen changes are made at the sensors or their configuration;
2. The sensors need to support HTTP and REST or SOA, which requires more capabilities in each sensor, both to support message processing and to overcome the natural mismatch between TCP/IP and the wireless protocol.

Solution 2 seems to be the most popular given that sensors are becoming equipped with better hardware and REST and SOA are tried technologies with implementations at this low level that are starting to appear [12, 16]. However, the fact is that these technologies are not native solutions for IoT (were conceived for text based hypermedia rather than this level of granularity, usually binary based) and they rely on the simplicity of the application.

Another way to put the issue is to say that applications cannot get more complex than the technology allows. History shows that, when the technology evolves

and gets better and more efficient, new applications immediately exploit those improvements. That is our motivation.

We will use the *Wish list (SIL)* column in Tables 2 through 5 as a set of requirements to design SIL (Service Implementation Language), a new interoperability technology that considers the IoT range of applications right from the start. The basic tenet is to reconsider the interoperability problem and to derive the solution that most closely matches the context of IoT, without the constraint of compatibility with previous technologies. We continue to use the interoperability framework described in Table 1.

Service interoperability

The basic problem is to make two resources able to interact by making their services compatible at the syntactic interface level (assuming higher interoperability levels have been dealt with). According to Table 2, we want to be able to combine the design time support and low semantic gap of SOA (resources with a fixed set of generic operations) with the linked and dynamic resource structure of REST, which includes operations as resources.

To make this possible, we have defined a language to specify and implement resources and their services (**SIL – Service Implementation Language**), which is able to describe not only data but also behavior (operations). In fact, it is a distributed programming language, supporting both the RPC and resource interaction styles. Not only can each resource implement any number of operations, each with any type of input and output, but it can also be structured, recursively composed of other resources, with a path to identify component resources. Operations are (behavior) resources and can be sent messages.

SIL allows the specification of both platform independent code (SIL instructions) and platform dependent code, with primitive operations implemented in a programming language such as Java or C#. In each distributed node, a SIL server is needed, to host resources and to provide an execution platform, to run SIL code and to interface other programming languages through adapters. This interface can be done statically, with compile time generation of a class or resource description (in a similar way to binding WSDL and object-oriented programming languages), or dynamically, by using reflection.

SIL nodes can use any protocol that provides transport for binary messages. A SIL server can support several protocols and heterogeneous networks if network gateways are available.

Program 1 illustrates some of the features of SIL with a simple description and partial implementation of the *Controller* resource of Figure 6. Like JSON, structured resources are defined between curly brackets and named components with a colon. SIL uses the same mechanism to define named operations (with the keyword **operation**), which can be primitive (implemented in another language, with the keyword **primitive**). Each operation can have only one input and out-

put parameter (but which can be structured), separated by the token `->`. These can be accessed in the body of the operation with the predefined identifiers **in** and **out**, respectively.

```
{
// definitions
define humidityValue as [0..100]; // integer range
define thermometer as {
  networkID: integer; // ID in the sensor network
  primitive (-> number); // unnamed operation (get temperature)
  setAlarm: primitive ({&any; number}); // structured parameter
};

// state components
hygro1: {
  networkID: integer; // ID in the sensor network
  primitive (-> humidityValue); // unnamed (get humidity)
  setPeriod: primitive ([1..60]); // sets history sampling period
  history: primitive (-> array humidityValue); // get values stored
};
hygro2: hygro1; // replicate resource
thermoArray: array thermometer; // array can grow

// operation components
init: operation (array integer){ // array has IDs of thermometers
  for i (0..in.size)
    addThermometer <-- in[i];
};

addThermometer: operation (integer) {
  th: thermometer;
  th.networkID = in;
  thermoArray.add <-- th; // add an array element
};

getAverageTemperature: operation (-> number) {
  total: number; // initialized to 0.0
  if (thermoArray.size == 0)
    reply 0; // default value
  for i (0..thermoArray.size)
    total += thermoArray[i];
  reply total/thermoArray.size; // compute average
};
};
```

Prog. 1. Specification of the *Controller* resource in SIL. Reserved words are in bold.

Definitions, at the beginning, are only auxiliary and generate no components until used. Components can also be declared inline, such as shown by `hygro1`.

Operations are behavior resources and are executed (invoked) by sending them a message, with the `<--` operator. The message to send can be omitted, if there is no input parameter. This is the same mechanism used to send messages to non operation resources, such as `hygro1`. In this case, an operation without name with matching input parameter will be invoked. This usually corresponds to a GET operation in REST.

The *Controller* has three operations (`init`, `addThermometer` and `getAverageTemperature`), all non-primitive. This means that they will be executed by the SIL execution platform, in a portable way. The compiler transforms their instructions into *silcodes* (equivalent to Java's bytecodes), which are executed by a virtual machine (an interpreter). Resources that have no primitive operations can be suspended, migrated from one server to another and have executed resumed at the new server.

Once created and registered in a server's directory (which includes a resource name server), the *Controller* can be configured with a set of sensors by sending its `init` operation a message with the `networkIDs` of several sensors:

```
controller.init <-- {107; 129; 114};
```

This resource path, ending in an operation, is typical of the REST style, but in SIL it blends seamlessly with the SOA style, since we are not limited to an universal set of operations.

The *Client* resource could have a definition such as shown in Program 2. Note that instructions can be interspersed with state resource declarations. The entire set will be executed once, upon resource creation, becoming ready to receive messages afterwards. In effect, the declaration of a resource is its constructor.

```
{
  t: number;
  h: integer;
  t = controller.thermoArray[1] <||; // asynchronous message
  h = controller.hygro1 <--; // synchronous message
  controller.thermoArray[2].setAlarm(&alarm; 30);
  someOtherResource <-- {temperature: t; humidity: h};

  alarm: operation (integer){ // networkID (no name for flexibility)
    ... // deal with alarm
  };
};
```

Prog. 2. Specification of the *Client* resource in SIL.

An asynchronous message (with the `<||` operator) invokes the unnamed operation of `thermometer 1` in the *Controller*'s array but returns a *future* immediately (stored in `t`), so that processing can proceed concurrently with the message request. When the reply value arrives, it will automatically replace the future. If the value of `t` is used before that (in the message to `someOtherResource`), execution of the *Client* will be blocked until the value is available. The message to sensor `hygro1` is synchronous and waits until the value is replied.

The *Client* has an operation `alarm`, to be invoked whenever a given sensor exceeds some temperature. The operation `setAlarm` in `thermometer 2` is sent a message composed of a reference to the *Client*'s `alarm` operation (obtained with the `&` operator) and a threshold temperature. The first component of the `setAlarm` operation's parameter is declared also as a reference (again, using the `&` operator),

with the predefined value `any`. All resources comply with `any`, which means that any resource can be used to receive that alarm.

SIL is a distributed programming language, in which references are global (such as URIs), not local (pointers). For this reason, references must be specified explicitly with the `&` operator, as in this example, and, unlike typical programming languages, assignments have structural copy semantics. This means that, in an assignment to a structured resource, only the components of the value to assign that *comply* with the target resource are actually assigned [41]. This is similar to what an XML processor does, by processing only the tags it recognizes and ignoring the rest.

In these examples, resources are declared and referenced with static pathnames, which allows design time support and verification from the SIL compiler. However, it is also possible to create and delete resources dynamically and specify resource pathnames as structured references, with an array of links, which is the basic support for heterogeneous networks, with different protocols and resource identification mechanisms (not shown here for simplicity).

Schema interoperability

There are **no type declarations in SIL**, only values with an associated variability. For instance, `integer` in the declaration `networkID` in `hygro1` (Program 1) is not a type but the predefined value `zero` with an associated variability identical to the allowed integer range. The `networkID` component gets both the value and the variability. The value can change, but not the variability. In this way, any valid integer value can be assigned to `networkID`. On the other hand, `humidityValue` is defined with a smaller range of variability (ranges verified by the compiler in assignments).

The hygrometer `hygro1` is defined directly, without a previous definition, and `hygro2`, another identical sensor, is simply defined by replication. There is no type instantiation. Actually, type compatibility in SIL is done not by a shared type declaration but by *structural interoperability* (*compliance* and *conformance*) [37], in which only the used components are required. For example, if we need to invoke an operation `X`, any resource that implements it (with compatible input and output parameters) can be used, independently of all the other components it may have. The reason for this is the distribution context, in which the lifecycles of resources are independent and common named type declarations become meaningless.

SIL resources and messages are completely self-describing, with a schema mechanism that differs from that of usual schema languages in two fundamental ways:

- The schema is specific of a given resource (document, message or service implementation), not of a set of documents. Instead of ensuring compatibility be-

tween resources by sharing a common schema, structural interoperability is used to check compatibility whenever needed. This reduces coupling and widens compatibility [37], since now interoperability is checked on a message by message basis, not on a full set of possible messages;

- There is no separate schema language. SIL itself, the declaration of resources themselves, fills this role. This is possible because the schema pertains to one resource only and there are no types, only values with an associated variability. The SPID (SIL Public Interface Descriptor) is the equivalent of WSDL or WADL and simply consists of all the public features of a resource declaration such as those of Programs 1 and 2. It suffices to remove the body of non-primitive operations and any resources declared as `private` (feature not illustrated here, for simplicity), which the compiler does easily. Design time compatibility checking is ensured by structural interoperability, which the compiler supports.

Serialization format

SIL has a source text format, adequate for people and illustrated by Programs 1 and 2, and a binary format, used for computer processing, either to be stored in a file or sent in a message. For programming, the source text is the input and the binary format is derived by the compiler. For runtime generated messages, only binary is used, although a decompiler can generate a source text format. The binary format usually includes metadata that supports self-description.

SIL uses the concept of *binary stream*, a sequence of bits or bytes with content's meaning and format known to both sender and receiver, encoded in a modified version of the TLV (Tag, Length and Value) scheme used by ASN.1 [42]. Streams can be composed of other streams, which allows for instance XML or JSON strings (a string is also a binary stream, with some encoding) to be part of a SIL resource. Each serialized SIL resource can have up to three streams:

- A source stream (a string such as Program 1 or 2);
- A compiled stream, composed of streams corresponding to primitive resources and streams corresponding to structured resources, composed of other streams. No metadata, aside from the semantics resulting from the streams' tags and structuring, is included here;
- A metadata stream, which includes information on component indices (relative position in the resource), names and value variability.

A resource can use the following combination of streams:

- Source only: mostly for programming and documentation (including SPIDs, the equivalent to WSDL and WADL), but can also be sent at runtime and compiled on the fly by the receiver;

- Compiled + metadata: complete information as well (aside source comments), but more efficient than source only;
- Compiled only: the most efficient, but does not support runtime interoperability checks, unless an optimization mechanism is devised such as the one described below;
- Metadata only: used essentially to represent SPIDs more efficiently, in binary format;
- All three: all the information available on a resource.

Using only the compiled stream looks like a return to the old specific binary formats, but it is in fact one of the distinguishing features of the support of SIL to the IoT applications. In this context, messages from or to low level devices typically follow the same schema until the device suffers some change. Sending metadata in each message is rather inefficient. The compiled stream can include a token returned by the server on the first message request. On reception of subsequent requests, the server checks this token and skips interoperability checking if the token matches its own or returns an error (the client then repeats the request with compiled + metadata). This is not a security feature, but a mere optimization mechanism. Its robustness depends on the number of bits of the token and its pseudorandom non-repeating evolution algorithm. Each resource maintains a cache of the tokens returned by the resources it has sent messages to. This mechanism bears similarities with the Etag header of HTTP.

Separating data from metadata is also done in the binary formats Protocol Buffers and Thrift [20], precisely for efficiency reasons. Schemas can be much larger than actual data, a problem identified but not solved in [19]. The solution in SIL is to be able to send a message without metadata, with a mechanism that still ensures interoperability with the help of design-time checks by the compiler.

Message protocol

A message in SIL is a resource, just like any other, and can include operations. The message protocol is the simplest possible to allow a message request and reply, independently of message content. All the rest is extensible and uses the envelope approach, in which a message is encapsulated with further control information into another (the envelope) at the sender and retrieved from that envelope at the receiver. Security, for instance, can use this mechanism. There are no specific purpose headers.

The message protocol supports asynchronous messages (with futures that can be cancelled if the client gives up waiting or decides otherwise) and application faults (exceptions). Each SIL server maintains a message ID generator, based on a non-repeatable, large sequence pseudorandom number generator. The message ID is used to correlate a response to the original message sender (which may be blocked, waiting for that response, if the message was asynchronous).

This ID mechanism is similar to the Token option in COAP (Constrained Application Protocol) [27], but more efficient since it is part of the basic protocol and handled in binary. In fact, the SIL message protocol contemplates the most relevant features of CoAP, namely asynchronous transactions and binary control information (headers). In fact, these features should be available in HTTP itself.

The main message types of the SIL message protocol are described in Table 6.

Table 6. Main message types of the SIL message protocol.

Message category/type	Description
<i>Request</i>	<i>Initial request in each transaction</i>
React	React to message, no answer expected
React & respond	React to message and answer/notify
Assimilate	Merge the message into the receiver resource, subject to structural interoperability (only compatible components are replaced). It bears similarities with PUT in REST.
<i>Amendment</i>	<i>Further information on an already sent request</i>
Cancel	Cancel the execution of the request
<i>Response</i>	<i>Response to the request</i>
Answer	A (structured) value returned by the <code>reply</code> instruction
Resource fault	A (structured) value returned as the result of an exception
Protocol fault	A status code resulting from a predefined protocol error
<i>Notification</i>	<i>Information of completion status</i>
Done	Request completed but has no value to reply
Cancelled	Request has been cancelled

The message protocol includes the addresses of both sender and receiver, so that the request can be addressed and the receiver can address the response back. These addresses, which correspond to resource names and pathnames in SIL, build on the following assumptions:

- Pathnames can span several networks, with different addressing schemes (e.g., TCP/IP and ZigBee);
- A name server exists for each network, so that a pathname can be converted into a list of network addresses, one for each network;
- Gateways inspect the address list in each message and route it accordingly.

The SIL message protocol includes either an address or a list of addresses for both sender and receiver. These are encoding using the same TLV format of the serialization protocol. This means that SIL resources can directly address others in different networks.

Assessing the new approach

Contrasting SIL and related technologies

History has taken its course and evolved from two main technologies, HTTP and HTML, in a context of many clients for each server and essentially retrieval of multimedia documents. Together with the corresponding execution platforms, the server and the browser, they constituted a well matched technology set that fostered the exponential development of the Web.

The mismatches begun when these technologies started being used and extended for all sorts of application domains and scenarios, from large business systems down to small IoT applications, from large scale server accesses to peer level interaction, from synchronous to asynchronous transactions, and so on. Table 7 tries to shed some light into the picture, by expressing the depth span in the interoperability ladder of some of the most relevant technologies.

Table 7. Levels of interoperability tackled by some existing technologies. Lighter gray in the right column means future work.

Concept	Interoperability level	HTTP	XML JSON	SOAP	WSDL	BPEL	REST	SIL
Alignment	Strategy							
Cooperation	Partnership							
Outsourcing	Value chain							
Ontology	Domain							
Knowledge	Rules							
Contract	Choreography							
Interface	Service							
Structure	Schema							
Serialization	Message format							
Interaction	Message protocol							
Routing	Gateway							
Communication	Network protocol							

HTTP has been enriched with all the features needed to support interaction in the original Web context. It is actually a service level protocol, with a fixed set of operations (such as GET and POST). It does almost everything, including control data description and type of payload data (with Internet Media Types).

XML has generalized HTML, separating data from formatting and introducing self-description with a schema, but retained much of its look and feel, still with a

data document nature (just data, instead of a more complete service nature, by including code) and text with markup (lacks native binary support).

Web Services appeared to fill in the service gap, removing the HTTP's restriction of a fixed interface. But, as Table 7 shows, there is a great overlap in interoperability levels between HTTP, SOAP and WSDL. The latter can be bound to protocols other than SOAP and SOAP can be bound to protocols other than HTTP. In practice, the most common situation by far is to have WSDL with SOAP over HTTP, which means that all this generality is seldom used but has permanent costs. SOAP treats HTTP essentially at transport level, ignoring many of its features. This is a sign of mismatch in implementing generic technologies. One sits, as is, on top of another, without separating the components that match from those that don't. This increases complexity and decreases performance.

Another sign of over generality is the universal use as XML as the underlying serialization format, with specific schemas, such as it happens in SOAP, WSDL and BPEL. This is a good thing, in principle, but these schemas evolve rarely and are standardized, which means users cannot change them. Therefore, all the power and flexibility of XML becomes of limited usefulness, at the same time that its verbosity and complexity are always present. A classic example is the assignment instruction in BPEL, with a baroque syntax instead of a mere equal sign.

These problems are particularly stringent in lower level applications, such as those in the IoT context, and justify the increasing popularity of REST and JSON. REST is essentially HTTP, as Table 7 expresses, or a set of best practices on how to use it. JSON is much simpler than XML and a good match to REST. Code still needs to be provided by another technology, typically a generic programming language. This is fine for simple applications but is limited for more complex ones.

In summary, the main problems are:

- The basic technologies, HTTP and XML (or JSON) are already immediately below the application level (the reason why REST is so simple to use). This causes mismatches, duplications and inefficiencies when subsequent technologies, such as Web Services, have to map onto them;
- The use of a single serialization language, for both people and computers, means that it becomes hard to read, inefficient to process and awkward to support binary data.

The purpose of SIL is to cleanup this scenario, learning from previous technologies and showing that a single language can replace many of the existing technologies. The simple description and examples used in this chapter are not enough to fully show how this can be accomplished, but the most fundamental ideas are:

- Move from client-server to peer dialog, in a service oriented paradigm;
- Base interoperability in compliance and conformance, not schema sharing;
- Derive the schema directly and automatically from the document, instead of having a separate document with different rules;
- Support and describe both data and code, as well as service and resource architectural styles;

- Use one serialization format (text based) for people and another (binary based) for computers, but derive the second from the first automatically;
- Use a simple protocol as the underlying communication mechanism.

SIL may be seen as a return to the RPC (Remote Procedure Call) programming model. Up to a certain point this is true, but with fundamental differences:

- Message marshalling (data serialization) has its own rules, which include self-description. This means that different languages can interact. Actually, coupling is even lower than in XML based systems, since compliance and conformance are used instead of schema sharing [37];
- The target of a message is identified by a distributed reference (such as a URI), in a server based interaction setting;
- Asynchronous transactions are readily supported;
- The interaction is not merely data based. Full resources (data + code) can be exchanged, which means that the basic mechanism is not the operation call but in fact the migration of a resource (the message).

To the best of our knowledge, there is no other proposal with such a wide range of objectives, while maintaining the decoupling and distributed interoperability that characterize current Web applications and technologies. Web Services or HTTP+XML/JSON are usually the rock bottom of existing proposals in Internet based contexts. There are, however, some attempts to change parts of the global scenario.

Web Sockets [44] are fundamental in the efficient support for binary data. They use the protocol upgrade feature of HTTP and provide a substantial degree of compatibility with existing systems. Part of the HTML5 world, servers and firewalls are increasingly supporting them.

The textual nature of markup languages has been recognized as inefficient. Using text as the serialization format requires textual parsing at the message receiver and a heavier effort to generate the corresponding memory data structures than when using a binary format [47]. Proposals such as EXI (Efficient XML Interchange) involve compression and decompression for transmission or storage purposes [18], but text must be recovered and therefore parsing time is not reduced. Native binary serialization formats, such as Protocol Buffers and Thrift [20], aim to solve this issue but deal only with data. Binary is also the path taken by SIL, but with support for code. Since SIL has two serialization formats (text and binary), text is better (for flexibility) when a priori knowledge of the interlocutors is low, and binary is better (for performance) when messages repetitively use the same schema and the compiler can be used. Changing between the two formats can be automated by using a token, as described in the two previous sections.

Distributed applications can be programmed in generic programming languages, using XML or JSON for data level interoperability, or higher level languages, such as BPEL. This is most common in complex enterprise applications, typically SOA and XML based, but has been shown to also fit with the REST style [38]. BPEL provides the support for behavior (code) that Web Services lack, but

constitute a separate technology with a different paradigm (processes) and a cumbersome XML based syntax that becomes usable only by resorting to visual programming tools.

SIL has a classical syntax and supports not only processes but also services and resources, in an integrated way. Other proposals also favor the resource based approach. In [48], an information centric process model is proposed, centering the resource concept on business entities instead of instances of workflow activity. Others propose to represent and transfer not only data but behavior as well, such as a method to expose process fragments (described declaratively as reusable workflow patterns) as resources and to map business process concepts onto the usual HTTP-style of CRUD operations [49]. Going a step further, in Computational REST (CREST) [50] the basic entities are computational resources, in the form of continuations [51], providing a base model for code mobility. The client is no longer a mere interface to the user but a computational engine, capable of executing these resources. State transfer is a side effect of this execution.

This is precisely what SIL offers, showing that it is possible to combine all these features under a single model and syntax. Tables 2 through 5 compare SIL and other technologies in further detail.

Table 7 also expresses that SIL will be extended to encompass the two topmost semantic levels, initially by using SIL instead of XML to serialize RDF and OWL documents and afterwards by incorporating constructs to express knowledge and ontologies directly in SIL. However, this is future work and outside the scope of this chapter.

The organizational levels, above semantics (Table 1), are better expressed by frameworks and development methods [28] than by languages with descriptive or execution semantics.

Implementation

We have developed a compiler based on ANTLR [43], which converts source to instructions and data in a binary format, according to the streams described above. An interpreter then executes the binary code (*silcodes*, equivalent to Java's bytecodes). The current implementation, in pure Java, is not optimized and has a performance roughly 50 times slower than a Java Virtual Machine (JVM). However, much of that time is spent just on method dispatch, the mechanism used to execute the various *silcodes*. A C based interpreter, for example, would be much faster, although harder to develop. To maintain flexibility and control of implementation, we did not use a JVM and bytecodes.

Support for distribution is implemented with a Jetty application server, but any other server will do. In fact, we only need a protocol handler, which can be much simpler in the case of simpler network protocols. For message exchange, we require only a transport level protocol. We currently use Web Sockets [44], with a cache for automatic connection management, but again any lower level message

transport protocol will be enough, provided that it implements the level of reliability required by the application or the application provides that itself.

The Jetty server connects to a SIL server (to handle the SIL message level protocol) that hosts a resource directory for service discovery. This is a regular service, just like any other, that contains references to the SPIDs of the resources registered in it. This directory can be searched for a suitable service by supplying keywords and/or a SPID as required by the client. The directory then searches for these keywords in the registered SPIDs and performs a structural interoperability check to ensure that the returned references to SPIDs are conformant to the SPID used in the search. Similarity ranking [45] is not supported at the moment.

Figure 2 shows the basic message path in a transaction between two services, each implemented by a resource. Figure 7 shows the basic architecture of a SIL node, capable of hosting SIL resources. This is the unit of distribution in the SIL realm. A reference to a SIL resource is made of two parts:

- The network level address of the SIL node. This is network dependent. With TCP/IP, this is the IP address of the SIL node;
- The path, within that node, from the directory (the root of the resource tree) to that resource. This is network agnostic and depends only on the structure of the resources.

URIs join the two parts in a single string. SIL has a primitive reference type that maintains the two parts separate, so that these references can be used in non IP networks.

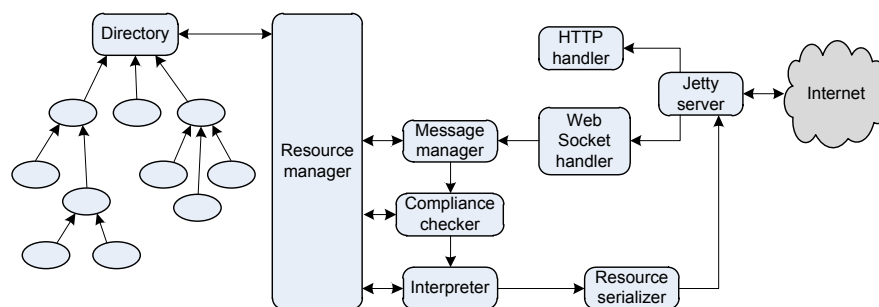


Fig. 7. Basic architecture of a SIL node.

The architecture of a SIL node can be described as follows:

- The application server (Jetty, in this case) is the interface to the network. It supports several message handlers, which means that it can deal with several message level protocols. We have only implemented two, HTTP and the message protocol of SIL. Only the steps ensuing the latter are described here. There is a list of handlers to be invoked and each checks whether it recognizes the

message format. The first one to do so gets the message for further processing. All SIL messages begin with “SIL”, encoded in UTF-8;

- A message received is handled next by the Message Manager, which determines the recipient of the message, the type of the message (Table 6), which streams are present in the message and, for some message types, whether a compliance token is present (described in section Serialization Format, above);
- The Resource Manager implements the access to the structured resources registered in the Directory, obtaining internal references (indices in a resource table) to resources targeted by messages or by distributed references (URIs, for example);
- The Directory is the root resource and implements several operations, such as searching for a resource which conforms to a given SPID. The resource tree depicted in Figure 7 shows only containment relationships. Any resource can have a distributed reference to another, but only if it is registered as globally accessible in a Directory. This means that resources can be locally reached from others during execution of a SIL program, but only registered resources, directly in the Directory, can be addressed by a global, distributed reference;
- The Compliance Checker performs type compliance between the message and the addressed operation or resource. It can do so in text or binary formats, since each has all the information needed, as long as the metadata stream is also present. Naturally, this is faster when done in binary. Messages that include a compliance token, obtained in a previous checking, can skip this step, as mentioned before in the section Serialization Format;
- If the resource targeted by the message is not an operation, the Compliance Checker must go through the various operations defined in that resource, to find one which the message complies with. A Protocol Fault (Table 6) is returned if none is found;
- Once the target operation has been identified, a thread is created in the silcode interpreter to execute that operation’s code, produced previously by the SIL compiler before the resource has been registered in the Directory. That code can access other resources, according to what has been programmed in the operation;
- If the operation needs to send a message to another resource, as illustrated by Program 2, it has to pass the resource to send as a message to the Resource Serializer, which is done by the send operators (<-- or <| |, according to whether the message is synchronous or asynchronous, respectively).

Migration path

SIL does not require a big bang migration path. An incremental e evolutionary approach can be achieved by using coexistence of several interoperability technologies and protocols.

SIL is application server and protocol agnostic and can coexist with SOA and REST applications. For example, the Jetty server used in the implementation of the SIL platform maintains normal HTTP capability, which means that it can also deal with SOA and REST based applications, by automatically choosing handlers based on the format of incoming messages.

The SIL server itself is able to deal with XML and JSON media types, through the stream concept. When a message is received, a set of available handlers are invoked to check whether they can process that message. The SIL message handler can be first and quick to recognize that the message is not SIL (lack of the right preamble), in which case it can invoke other handlers.

Conclusions and future work

Simplicity is the key concept in the IoT, in particular in what concerns the lower level devices. This has been the driving force behind the popularity of REST as service interoperability model, JSON as a serialization format and plain HTTP as a message level protocol. Web Services, SOA and XML can be too complex for simple applications, but offer the design time support that lacks in REST, JSON and HTTP, especially with simple devices that cannot cater for higher levels of interoperability. Application dynamicity is important, but so is verifiability and developer support.

REST APIs are simpler, but lack this design time support. It is also a fact that REST APIs in the context of IoT applications are extremely simple, with almost no states involved. This simplicity stems from applications, not from the technology, which was not conceived for the IoT, but rather for the Web, and has limitations, namely regarding binary support and asynchronous event processing.

Our opinion is that current Web technologies need to be reevaluated, taking into account the smaller granularity, higher constraints and even higher massive scale of networked devices. This is already happening, with adaptations of IPv6 to low power devices and sensor networks [22], as well as adaptations of HTTP for constrained resources [27]. A high legacy load is present, without achieving transparent compatibility. The design of a fresh solution, incorporating lessons learned and without being hampered by compatibility tradeoffs, has been the basic motivation for the design of the alternative solution that we have described in this chapter, which meant adopting the following main principles:

- Do not use text with markup for the resource serialization format, which is complex for human reading and inefficient for machine processing. Use two formats instead: programming language style for humans and binary for machine processing, with a compiler to link the two;
- Use complete separation of data and metadata (which text markup does not allow). This supports automatic use of metadata, only when needed. When the

schema does not change, send only binary data, with design time checks done by the compiler;

- Use a message protocol with native support for binary data and asynchronous messages;
- Do not base schema level interoperability on schema sharing (as usually done in XML documents) or implied schemas (often, the case of JSON data). Use structural interoperability [37] instead, which decreases coupling;
- Do not use a separate language to describe schemas. Derive them automatically from the textual resource descriptions;
- Support a variable number of operations for each resource but support external access to resource structure as well (both measures contribute to a low modeling semantic gap);
- Support multinetwork resource references, so that heterogeneous networks can be seamlessly integrated.

A preliminary implementation of these principles exists, with a compiler and an execution platform for a language that implements these principles, SIL, but much remains to be done. In particular, the following aspects are already being tackled:

- Completion and optimization of the SIL platform;
- Extension of SIL to the upper semantic levels;
- A study comparing quantitative aspects (execution time and memory requirements) and qualitative aspects (ease of programming and of changing, advantages and perils of platform independent code) between SOA, REST and SIL based solutions, especially in the low level granularity context of IoT;
- An assessment of scenarios of application. One particularly interesting concerns joining a SIL server and a conventional browser, working in tight cooperation, something we call the *browserver* [46] and that has been conceived to replace the browser as a Web access device (at the user's laptop, tablet or smart phone). This has the great advantage of turning the user into a first class Web citizen, improving the interactivity experienced by the user and allowing him to automatically offer services (including private information for personalization, context awareness, ambient intelligence, authentication, gathering statistics and information on usage patterns, direct browserver to browserver interaction, which can be used for group or collective intelligence, and so on).

References

1. Berners-Lee T (1999) Weaving the web: the original design and ultimate destiny of the World Wide Web by its inventor. HarperCollins Publishers, New York
2. Luigi A, Iera A, Morabito G (2010) The Internet of Things: A survey. *Comput Netw* 54:2787–2805
3. Sundmaeker H, Guillemin P, Friess P, Woelfflé S (2010) Vision and challenges for realising the Internet of Things. Publications Office of the European Union, Luxemburg

4. Guinard D, Trifa V, Mattern F, Wilde E (2011) From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices. In: Uckelmann D, Harrison M, Michahelles F (eds.) *Architecting the Internet of Things*, Springer, Berlin
5. Abawajy J (2009) Advances in pervasive computing. *International Journal of Pervasive Computing* 5(1):4–8
6. Guinard D, Mueller M, Pasquier-Rocha J (2010) Giving RFID a REST: Building a Web-Enabled EPCIS. *Proc. Second International Internet of Things Conference* (pp. 1-8) doi: 10.1109/IOT.2010.5678447
7. Earl T (2005) *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River
8. Richardson L, Ruby S (2007) *RESTful Web Services*. O'Reilly Media, Sebastopol, CA
9. Pautasso C, Zimmermann O, Leymann F (2008) Restful web services vs. "big" web services: making the right architectural decision. *Proc. International conf. on World Wide Web* (pp. 805-814) ACM Press
10. Mulligan G, Gracanin D (2009) A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. *Proc. Winter Simulation Conf.* (pp. 1423-1432) IEEE Computer Society Press.
11. Becker J, Matzner M, Müller O (2010) Comparing Architectural Styles for Service-Oriented Architectures - a REST vs. SOAP Case Study. In: Papadopoulos G et al (eds.) *Information Systems Development* (pp. 207-215) Springer-Verlag US.
12. Gupta V, Udupi P, Poursohi A (2010) Early lessons from building Sensor.Network: an open data exchange for the web of things. *Proc. Conf. on Pervasive Computing and Communications Workshops* (pp. 738-744) doi: 10.1109/PERCOMW.2010.5470530
13. Taherkordi A, Eliassen F, Romero D, Rouvoy R (2011) RESTful Service Development for Resource-Constrained Environments. In: Wilde E, Pautasso C (eds.), *REST: From Research to Practice*, Springer Science+Business Media, New York
14. Guinard D, Trifa V, Wilde E (2010) A resource oriented architecture for the Web of Things. *Proc. Second International Internet of Things Conf.* (pp. 1-8) doi: 10.1109/IOT.2010.5678452
15. Priyantha N, Kansal A, Goraczko M, Zhao F (2008) Tiny web services: design and implementation of interoperable and evolvable sensor networks. *Proc. 6th ACM conf. on Embedded network sensor systems* (pp. 253-266), doi: 10.1145/1460412.1460438
16. Akribopoulos O, Chatzigiannakis I, Koninis C, Theodoridis E (2010) A Web Services-oriented Architecture for Integrating Small Programmable Objects in the Web of Things. *Proc. Developments in E-systems Engineering Conf.* (pp. 70-75), doi: 10.1109/DeSE.2010.19
17. Jammes F, Mensch A, Smit H (2005) Service-Oriented Device Communications using the Devices Profile for Web Services. *Proc. 3rd international workshop on Middleware for pervasive and ad-hoc computing* (pp. 1-8) doi: 10.1145/1101480.1101496
18. Sakr S (2009) XML compression techniques: A survey and comparison. *J Comput Syst Sci* 75(5): 303-322
19. Moritz G, Timmermann D, Stoll R, Golatowski F (2010) Encoding and Compression for the Devices Profile for Web Services. *Proc. 24th International Conf. on Advanced Information Networking and Applications Workshops* (pp. 514-519) doi: 10.1109/WAINA.2010.91
20. Sumaray A, Makki S (2012) A comparison of data serialization formats for optimal efficiency on a mobile platform. *Proc. 6th International Conf. on Ubiquitous Information Management and Communication*, doi: 10.1145/2184751.2184810
21. Hui J, Culler D (2010) IPv6 in Low-Power Wireless Networks. *Proc IEEE*, 98(11):1865-1878
22. Jacobsen R, Toftegaard T, Kjærgaard J (2012) IP Connected Low Power Wireless Personal Area Networks in the Future Internet. In: Vidyarthi D (ed.) *Technologies and Protocols for the Future of Internet Design: Reinventing the Web*, IGI Global, Hershey PA
23. Hui J, Thubert P (2011), Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. Internet Engineering Task Force (IETF) RFC 6282. <http://tools.ietf.org/html/rfc6282>. Accessed 30 April 2012
24. Shelby Z, Bormann C (2009) *6LoWPAN: The Wireless Embedded Internet*. Wiley UK

25. Lewis G, Morris E, Simanta S, Wraga L (2008) Why Standards Are Not Enough To Guarantee End-to-End Interoperability. Proc. Seventh International Conf. on Composition-Based Software Systems (pp. 164-173) doi: 10.1109/ICCBSS.2008.25
26. Diallo S, Tolk A, Graff J, Barraco A (2011) Using the levels of conceptual interoperability model and model-based data engineering to develop a modular interoperability framework. Proc. Winter Simulation Conf. (pp. 2571-2581) doi: 10.1109/WSC.2011.6147965
27. Castellani A, Gheda M, Bui N, Rossi M, Zorzi M (2011) Web Services for the Internet of Things through CoAP and EXI. Proc. International Conf. Communications Workshops (pp. 1-6), doi: 10.1109/iccw.2011.5963563
28. Minoli D (2008) Enterprise Architecture A to Z. Auerbach Publications, Boca Raton, FL.
29. Masse M (2011) REST API Design Rulebook. O'Reilly Media, Sebastopol, CA
30. Gislason D (2008) Zigbee Wireless Networking. Elsevier, UK
31. Trifa V, Wiel S, Guinard D, Bohnert T (2009) Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices. Proc. 2nd International Workshop on Sensor Network Engineering
32. Fielding R, Taylor R (2002) Principled Design of the Modern Web Architecture. ACM Trans Internet Technol, 2(2):115-150
33. Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California at Irvine. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf. Accessed 30 April 2012
34. Appel A, Jim T (1989) Continuation-passing, closure-passing style. Proc. Symp. Princ. Program. Lang. (pp. 293-302) doi: 10.1.1.134.7735
35. Webber J, Parastatidis S, Robinson I (2010) REST in Practice. O'Reilly Media, Sebastopol, CA
36. Zyp K, Court G (2011) A JSON Media Type for Describing the Structure and Meaning of JSON Documents. Internet Engineering Task Force (IETF) draft-zyp-json-schema-03. <http://tools.ietf.org/html/draft-zyp-json-schema-03>. Accessed 30 April 2012
37. Delgado J (2012) Structural interoperability as a basis for service adaptability. In: Ortiz G, Cubo J (eds.) Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solutions, IGI Global, Hershey PA
38. Pautasso C (2009) RESTful Web service composition with BPEL for REST, Data Knowl Eng 68(9):851-866
39. Guinard D, Trifa V, Pham T, Liechti O (2009) Towards physical mashups in the Web of Things. Proc. Sixth International Conf. Networked Sensing Systems (pp. 1-4) doi: 10.1109/INSS.2009.5409925
40. Castellani A, Bui N, Casari P, Rossi M, Shelby Z, M (2010) Architecture and Protocols for the Internet of Things: A Case Study. Proc. International Conf. Pervasive Computing and Communications Workshops (pp. 678-683), doi: 10.1109/PERCOMW.2010.5470520
41. Delgado J (2012) Bridging the SOA and REST architectural styles. In: Ionita A, Litoiu M, Lewis G (eds.) Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments, IGI Global, Hershey PA
42. Dubuisson O (2000) ASN.1 Communication Between Heterogeneous Systems. Academic Press, San Diego, CA
43. Parr T (2007) The Definitive ANTLR Reference, The Pragmatic Bookshelf, Raleigh, NC
44. Lubbers P, Albers B, Salim F (2010) Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development, Apress, New York, NY.
45. Formica A (2007) Similarity of XML-Schema Elements: A Structural and Information Content Approach, Comp J, 51(2):240-254
46. Delgado J (2012) The User as a Service. In: Vidyarthi D (ed.) Technologies and Protocols for the Future of Internet Design: Reinventing the Web, IGI Global, Hershey PA
47. Maeda K (2011) Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats, Proc. Second International Conference on Digital Information and

- Communication Technology and its Applications (pp. 177 - 182) doi: 10.1109/DICTAP.2012.6215346
48. Kumaran S et al (2008) A RESTful Architecture for Service-Oriented Business Process Execution. In: International Conference on e-Business Engineering (pp. 197-204), IEEE Computer Society Press.
 49. Xu X, Zhu L, Kannengiesser U and Liu Y (2010) An Architectural Style for Process-Intensive Web Information Systems, in Web Information Systems Engineering, Lecture Notes in Computer Science, 6488 (pp. 534-547), Springer-Verlag Berlin Heidelberg.
 50. Erenkrantz J, Gorlick M, Suryanarayana G and Taylor R (2007) From representations to computations: the evolution of web architectures. In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 255-264), ACM Press.
 51. Queinnec C (2003) Inverting back the inversion of control or, continuations versus page-centric programming, ACM SIGPLAN Not, 38(2), 57-64.

<i>binary stream</i> , 26	<i>service</i> , 5
BPEL, 15	Service Implementation Language, 22
<i>browser</i> , 36	Service interoperability, 9, 22
choreography, 10	<i>Service Web</i> , 1
<i>compliance</i> , 17	SIL, 22
<i>conformance</i> , 17	SIL Public Interface Descriptor, 26
distributed programming language, 25	<i>silcodes</i> , 32
<i>Internet of Things</i> , 2	SOA, 10
interoperability framework, 4	SPID, 26, 33
Message protocol, 19, 27	<i>structural interoperability</i> , 17, 26
<i>resource</i> , 5	<i>transaction</i> , 5
REST, 11	<i>Web of Documents</i> , 1
Schema interoperability, 16, 25	<i>Web of Things</i> , 2
Serialization format, 17, 26	Web Sockets, 31