19 Improving data and service interoperability with structure, compliance, conformance and context awareness

José C. Delgado

Department of Computer Science and Engineering

Instituto Superior Técnico, Universidade de Lisboa

Av. Prof. Cavaco Silva, Taguspark, 2744-016 Porto Salvo, Portugal

jose.delgado@tecnico.ulisboa.pt

Abstract: The Web has been continuously growing, in number of connected nodes and quantity of information exchanged. The Internet of Things (IoT), with devices that largely outnumber Internet users and have little computing power, has contributed to show the limitations of current Web technologies (namely, HTTP) for the integration of small systems. New protocols, such as CoAP and WebSockets, alleviate some of the problems but do not solve them at the origin. This chapter revisits the interoperability problem in the IoT context and proposes an architectural style, Structural Services, which combines the service modelling capabilities of SOA with the flexibility of structured resources in REST. The simplicity of JSON is combined with operation descriptions in a service-oriented distributed programming language that provides design time self-description without the need for a separate schema language. Interoperability, at the data and service levels, is enhanced with partial interoperability (by using compliance and conformance) and a context model that contemplates forward and backward contextual information. Efficiency and lower computing power requirements, aspects highly relevant for IoT systems, are supported by using a binary message format resulting from compilation of the source program and that uses self-description information only when needed.

Keywords: Internet of Things, Interoperability, Service, Resource, Compliance, Conformance, Context awareness

19.1 Introduction

Usually, Big Data is characterized by three main properties, referred to as the three Vs [1]:

- Volume (data size and number);
- Velocity (the rate at which data are generated or need to be processed);
- Variety (heterogeneity in content and/or form).

Essentially, *Big* means too complex, too much and too many to apply conventional techniques, technologies or systems, since their capabilities are not enough to handle such extraordinary requirements.

In the context of the Internet of Things (IoT), Big Data also means Big Services, in the sense of the three Vs: too many services interacting and exchanging too many data, at too high rates in a too heterogeneous environment. A service can be a complex application in a server or a very simple functionality provided by a small sensor. Two additional Vs (Value and Veracity) have been proposed by [2], but are less relevant to the scope of this chapter.

Service interaction implies interoperability. The main integration technologies available today are based on the SOA and REST architectural styles, with Web Services and REST on HTTP as the most used implementations. Web Services are a closer modelling match for more complex applications, since they allow a service to expose an arbitrary interface. In turn, they lack structure (all services are at the same level) and are complex to use. REST is certainly simpler, since essentially it entails a set of best practices to use resources in a generic way, as heralded by HTTP. Although REST provides structure, it supports a single syntax interface and only a set of previously agreed data types. Also, its apparent simplicity hides the functionality that the interlocutors need to be endowed with. In addition, both SOA and REST rely, in practice and in most cases, on HTTP and text based data (XML or JSON).

These technologies stem from the epoch in which people were the main Internet drivers, text was king, document granularity was high, the distinction between client and server was clear, scalability was measured by the number of clients and the acceptable application reaction times could be as high as one second (human timescale).

Today, the scenario is much different. Cisco estimates [3] indicate that, in the end of 2012, the number of devices connected to the Internet was around 9 billion devices, versus around 2.4 billion Internet users in June 2012, according to the Internet World Stats site (http://www.internetworldstats.com/stats.htm). The European Commission estimates that the number of Internet-capable devices will be on the order of 50-100 billion devices by 2020 [4], with projections that at that time the ratio of mobile machine sessions over mobile person sessions will be on the order of 30. The number of Internet-enabled devices grows faster than the number of Internet users, since the worldwide population is estimated by the United Nations to be on the order of 8 billion by 2020 [5]. This means that the Internet is no longer dominated by human users, but rather by smart devices that are small computers and require technologies suitable to them, rather than those mostly adequate to fully-fledged servers.

These technologies include native support for binary data, efficient and full duplex protocols, machine-level data and service interoperability and context awareness for dynamic and mobile environments, such as those found in smart cities [6]. Today, these features are simulated on top of Web Services, RESTful libraries, HTTP, XML, JSON and other technologies, but the problem needs to be revisited to minimize the limitations at the source, instead of just hiding them with abstraction layers that add complexity and subtract performance.

The main goals of this chapter are:

- To rethink the interoperability problem in the context of data intensive applications and taking into account the IoT and smart environment requirements, regarding the characteristics of devices (low computing capabilities, power constraints, mobility, dynamicity, heterogeneity, and so on);
- To show that the good characteristics of the SOA and REST models can be combined by using a better designed interoperability model, based on active resources instead of passive document descriptions;
- To illustrate how the three Vs of Big Data (and Big Services) can be better supported by this interoperability model, namely in terms of:
  - Performance (Volume and Velocity), by using binary data and cached data interoperability mappings;
  - Heterogeneity (Variety), by using compliance and conformance (less restrictive than schema sharing) and context awareness (which increases the range of applicability).

The chapter is organized as follows. Section 19.2 discusses interoperability in the IoT context and presents some of the current interoperability technologies. Section 19.3 provides a generic model of resource interaction and a layered interoperability framework. Section 19.4 establishes the main requirements to improve interoperability in IoT environments. Section 19.5 proposes and describes a new architectural style, *Structural Services*. Sections 19.6 and 19.7 show how interoperability can be improved with structural typing and context awareness, respectively. Section 19.8 discusses the rationale for the work presented in this chapter, section 19.9 present guidelines for future work and section 19.10 draws some conclusions and summarizes the scientific contributions of this chapter.

19.2 Background

IoT scenarios should be implemented by a complex ecosystem of distributed applications, with nodes interacting seamlessly with each other, independently of platform, programming language, network protocol and node capabilities (from a fully-fledged computer, connected to a wide, fast and reliable network, down to a tiny sensor, connected to a local, slow and *ad hoc* wireless network).

This is not what we have today. Distributed programming always faced practical difficulties, caused by the lack of a generic and widely available way to exchange data. The RPC (Remote Procedure Call) approach, such as Java RMI (Remote Method Invocation) and CORBA [7] provided a solution, but usually language or system dependent due to differences in protocols and message formats. In any case, one fundamental problem was the attempt to mimic a local environment (namely, with pointers) on top of a distributed one, instead of designing a system with a distributed mind-set from the start.

The advent of the Web changed this, by providing standardized mechanisms:

- To retrieve data from a remote node (a server), first in HTML and then by a format evolution towards XML and JSON;
- To remotely invoke behaviour in that node, first with the CGI (Common Gateway Interface) [8] and then with several technologies, such as Web Services [9] and REST [10], the most common today.

However, these mechanisms are supported on HTTP, a protocol conceived for the original Web problem [11], browsing remote hypermedia documents with text as the main media type. The main goal was information retrieval, from humans to humans, with client-server dichotomy and scalability as the main tenets. This justified many decisions taken at that time. Given the explosive success of the Web, these are still conditioning current technologies.

The IoT has different goals and needs from those of the Web, since the emphasis is now on machine to machine interactions, rather than human to human relationships.

Even the current Web, at human level, is very different from the original Web. Instead of a static Web of documents, we now have a Web of services, or at least a dynamic Web of documents, generated on the fly from databases or under control of server side applications. The classic client-server paradigm is also taking a turn with one of the latest developments, real-time communications in the Web [12] directly between browsers, by using WebRTC [13] in a peer-to-peer architecture.

This is being done in an evolutionary way, with essentially JavaScript and WebSockets [14], to circumvent some of the limitations of HTTP. The most important issue is that finally humans are also becoming direct Web producers and not merely clients of servers.

This is also the philosophy underlying IoT, in which all nodes, down right to very simple sensors, should be able to initiate transactions and to produce information. Since the IoT encompasses not only machines but also humans in a complete environment, a common architecture (or at least a common way to integrate architectures) should be the way to implement the vision of the IoT [4].

Typical implementations of the IoT use Web technologies for global interoperability. There are IoT applications based on SOA [15, 16], but the RESTful approach seems to be more popular [17, 18, 19], since it is simpler and more adequate to applications with smaller devices. 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 [17, 20], the only resource representations returned with links are the list of devices. This means that there is only application data structure exposed, not behaviour, which is essentially just a CRUD (Create, Read, Update and Delete) application [10].

In both SOA and REST, interoperability is achieved by using common data types (usually structured), either by sharing a schema (i.e., WSDL files) or by using a previously agreed data type (typical in RESTful applications). There is usually no support for partial interoperability and polymorphism in distributed systems. Compliance [21] and conformance [22] are concepts that support this, but they are not used in the SOA and REST contexts.

Context awareness [23] is very important to many IoT applications, but there is typically no support from current integration technologies for context-oriented services [24].

IoT devices can be integrated with the Web in the following ways:

- The device itself has Web capability (an HTTP server using SOA or REST) and can interact with the rest of the world directly. This implies connection to the Internet and support for the TCP/IP stack;
- Through a gateway, which connects to the Internet on one side and to the device's non-IP network (such as a ZigBee [25] wireless sensor network) on the other. The gateway needs to mediate not only network protocols but also the application level messages.

The protocols underlying the Web (TCP/IP and HTTP) are demanding in terms of the IoT. Hardware capabilities are increasingly better and efforts exist to reduce the requirements, such as IPv6 support on Low power Wireless Personal Area Networks (6LoWPAN) [26, 27] and the CoAP (Constrained Application Protocol) [28] to deal with constrained RESTful environments. However, the fact is that new applications, such as those based on nanotechnology [29, 30] and vehicular networks [31], prevent the universal adoption of an Internet backbone approach.

19.3 The interoperability problem

IoT scenarios and applications entail a complex ecosystem of resources interacting with each other in a big data environment, with many and heterogeneous data exchanged between many and heterogeneous resources.

To be valuable and meaningful, interactions require *interoperability*. This is one of the most fundamental characteristics of distributed systems. Although there is no universally accepted definition of interoperability, since its meaning can vary accordingly to the perspective, context and domain under consideration, probably the most cited definition is provided by the 24765 standard [32]. According to it,

interoperability is "the ability of two or more systems or components to exchange information and to use the information that has been exchanged". In other words, merely exchanging information is not enough. Resources must also be able to understand it and to react to it according to each other's expectations. To detail what this means, we need to analyse the interoperability problem, starting by establishing a model of resources and of their interactions.

19.3.1 A generic model of resource interaction

We define *resource* as an entity of any nature (material, virtual, conceptual, noun, action, and so on) that embodies a meaningful, complete and discrete concept, which makes sense by itself and can be distinguished from, although able to interact with, other entities.

Resources are the main modelling artefacts and should mimic, as closely as possible, the entities in the problem domain. Physical resources range from simple sensors up to people, although the latter are represented in the computer-based realm by user interface devices. Non-physical resources typically correspond to software-based modules, endowed with interaction capabilities.

Resources are discrete and distinct entities, atomic (an indivisible whole) or structured (recursively composed of other resources, its components, with respect to which it performs the role of container). Each component can only have one direct container (yielding a strong composition model, in a tree shaped resource structure). Resources can also exhibit a weak composition model, by referring to each other by *references*. The reference itself is an atomic resource, a component of the resource that holds it (its container), but not the referenced resource. References support the existence of a directed graph of resources, superimposed on the resource tree.

Atomic resources can have state, either immutable or changeable. The state of a structured resource is, recursively, the set of the states of its components. Resources can migrate (move) from one container to another, changing the system's structure.

Resources are self-contained and interact exclusively by sending *messages*, i.e., resources moved from the sender to the receiver. Each resource implements a *service*, defined as the set of *operations* supported by that resource and that together define its behaviour (the set of reactions to messages that the resource exhibits). We also assume that resources are typically distributed and messages are exchanged through a *channel*, a resource specialized in relaying messages between interacting resources.

One resource, playing the role of *consumer*, sends a request message (over the channel) to another resource, playing the role of *provider*, which executes the request and eventually sends a response message back to the consumer. This basic interaction patterns constitutes a *transaction*. More complex interactions can be achieved by composing transactions, either temporally (the consumer invokes the provider several times) or spatially (the consumer *X* invokes the provider *Y*, which in turn invokes another provider, *Z*, and so on).

A *process* is a graph of all transactions that are allowed to occur, starting with a transaction initiated at some resource and ending at a transaction that neither provides a response nor initiates new transactions. A process corresponds to a use case of a resource and usually involves other resources, as transactions flow (including loops and recursion, eventually).

Figure 19.1 depicts this conceptual model in UML. Association classes describe relationships (such as Transaction and the message Protocol) or roles, such as Message, which describes the roles (specialized by Request and Response) performed by resources when they are sent as messages. The interaction between resources has been represented as interaction between services, since the service represents the active part (behaviour) of a resource and exposes its operations. When we mention resource interaction, we are actually referring to the interaction between the services that they implement.

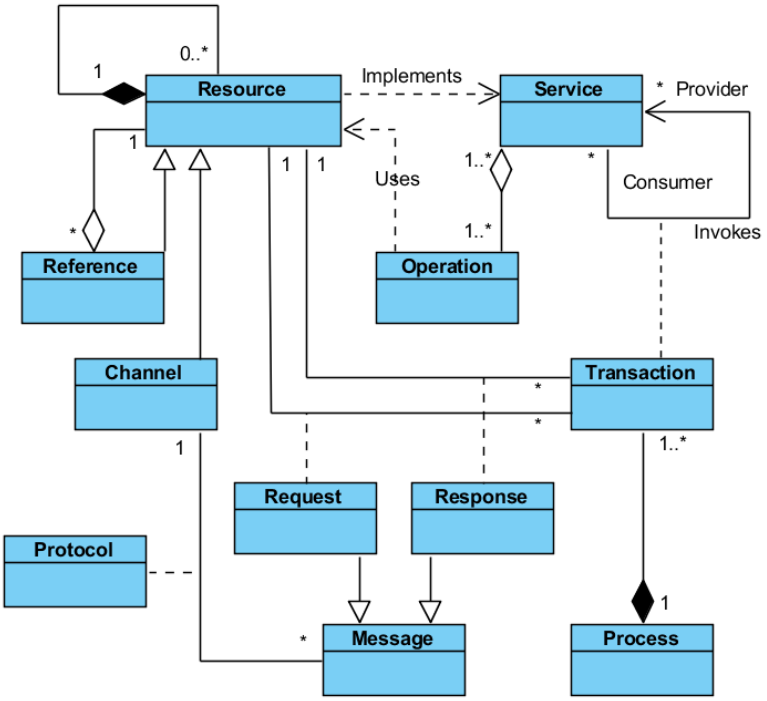

Fig. 19.1. A generic model of resource interaction

Therefore:

- Resources entail structure, state and behaviour;
- Services refer only to behaviour, without implying a specific implementation;

- Processes are a view of the behaviour sequencing and flow along services, which are implemented by resources.

This is a model inspired in real life, conceived to encompass not only computer-based entities but also humans (themselves resources) and any other type of physical resources, such as sensors.

19.3.2 The meaning of resource interaction

Figure 19.2 depicts the basic resource interaction scenario in a distributed environment, between a consumer *X* and a provider *Y*, and allows us to detail what happens during a transaction, with a request and response sent through some channel.

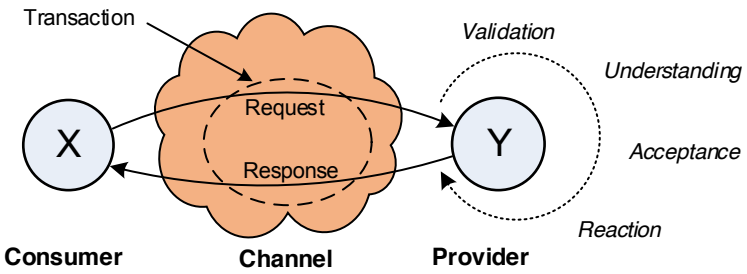

Fig. 19.2. Basic transaction between the services of two interacting resources

Figure 19.2 also details some of the phases which the request goes through until the response is sent. The corresponding phases in the consumer are omitted for simplicity but, in general, the goal of achieving such a simple transaction can be decomposed into the following objectives:

1. The request message reaches *Y* through an adequate channel, such as a network;
2. *Y* validates the request, according to its requirements for requests;
3. *Y* understands what *X* is requesting;
4. *Y* accepts to honour the request;
5. The reaction of *Y* to the request, including the corresponding effects, is consistent with the expectations of *X* regarding that reaction;
6. The response message reaches *X* (usually through the same channel as the request message);
7. *X* validates the response, according to its requirements for the response;
8. *X* understands what *Y* is responding;
9. *X* accepts the response;

10. *X* reacts appropriately, fulfilling the role of the response in a consistent manner with the overall transaction.

Therefore, just sending a request to a provider, or a response to the consumer, hoping that everything works, is not enough. We need to ensure that each message is validated and understood by the resource that receives it. In general, meaningfully sending a message (request or response) entails the following aspects:

- *Willingness* (objectives 4 and 9). Both sender and receiver are interacting resources and, by default, their services are willing to accept requests and responses. However, non-functional aspects such as security can impose constraints;
- *Intent* (objectives 3 and 8). Sending the message must have a given intent, inherent to the transaction it belongs to and related to the motivation to interact and the goals to achieve with that interaction;
- *Content* (objectives 2 and 7). This concerns the generation of the content of a message by the sender, expressed by some representation, in such a way that the receiver is also able to validate and interpret it;
- *Transfer* (objectives 1 and 6).The message content needs to be successfully transferred from the context of the sender to the context of the receiver;
- *Reaction* (objectives 5 and 10). This concerns the reaction of the receiver upon reception of a message, which should produce effects according to the expectations of the sender.

19.3.3 Abstraction levels and other aspects of interoperability

Each of these aspects corresponds to a different category of interoperability abstraction levels, which can be further refined. For example, the transfer aspects are usually the easiest to implement and lie at the lowest level. Transferring a message from the context of the sender to the context of the receiver requires *connectivity* over a channel, which can be seen at several levels, such as:

- Message protocol, which includes control information (for example, message type: request, response, etc.) and message payload (structured or just serialized as a byte stream);
- Routing, if required, with intermediate gateways that forward messages;
- Communication, the basic transfer of data with a network protocol;
- Physics, which includes communication equipment and physical level protocols.

Table 19.1 expresses a range of interoperability abstraction levels, ranging from very high level, governance management and coordination, to very low level, physical network protocols.

Table 19.1. Interoperability abstraction levels

Category	Level	Description
Symbiotic (intent)	Coordination Alignment Collaboration Cooperation	Purpose and intent that motivate the interaction, with varying levels of mutual knowledge of governance, strategy and goals. Can range from a coordinated governance to mere outsourcing arrangement (cooperation).
Pragmatic (reaction and effects)	Contract Workflow Interface	Management of the effects of transactions at the levels of choreography (contract), process (workflow) and service (interface).
Semantic (meaning of content)	Inference Knowledge Ontology	Interpretation of a message in context, at the levels of rule (inference), relations (knowledge) and definition of concepts (ontology).
Syntactic (notation of content)	Structure Predefined type Serialization	Representation of resources, in terms of composition (structure), primitive resource (predefined types) and their serialization format in messages.
Connective (transfer protocol)	Messaging Routing Communication Physics	Lower level formats and network protocols involved in transferring a message from the context of the sender to that of the receiver.

In practice, the higher levels tend to be dealt with *tacitly*, by assuming that what is missing has been somehow previously agreed or is described in the documentation available to the developer. Inferring intent and semantics from documentation, or undocumented behaviour stemming from programs, constitute examples of this.

In the same way, the lower levels tend to be tackled *empirically*, by resorting to some tool or technology that deals with what is necessary to make interoperability work. An example is using some data interoperability middleware, without caring for the details of how it works.

The syntactic category is the most common, with the pragmatic category used to express service interfaces or business processes. Semantics is lagging, normally dealt with tacitly by documentation, although in recent years the so called Semantic Web [33] has increased its relevance and visibility.

All these interoperability levels constitute an expression of resource coupling. On one hand, two uncoupled resources (no interaction between them) can evolve freely and independently, which favours adaptability, changeability and even

reliability. On the other hand, resources need to interact to cooperate towards common or complementary goals and some degree of previously agreed knowledge is indispensable. The more they share with the other, the more integrated they are and the easier interoperability becomes, but the greater coupling gets.

Therefore, one of the main problems of interoperability is to ensure that each resource knows enough to be able to interoperate but no more than that, to avoid unnecessary dependencies and constraints. This is an instance of the principle of least knowledge, also called the Law of Demeter [34].

Another important aspect is non-functional interoperability. It is not just a question of invoking the right operation with the right parameters. Adequate service levels, context awareness, security and other non-functional issues must be considered when resources interact, otherwise interoperability will not be possible or at least ineffective.

19.4 Improving on current solutions: from the Web to the Mesh

The Web is a rather homogeneous (due to HTTP), reliable (due to TCP/IP and to the Internet backbone) and static (due to DNS) environment. The nature of IoT is much more heterogeneous (due to lower-level networks, such as sensor and vehicular networks), unreliable (in particular, with mobile devices) and dynamic (networks are frequently *ad hoc* and reconfigurable in nature). This means that the Web, although offering a good and global infrastructure, is not a good match for the IoT.

The existence of a single, universal network and protocol is unrealistic. What we can do is to simplify the interoperability requirements so that the microcomputer level (such as in very small sensors) can be supported in a lighter and easier way than with current solutions (such as TCP/IP, HTTP, XML, JSON, SOA and REST). We simply cannot get rid of network gateways, but these should be transparent to applications.

Our approach is to revisit the interoperability problem in the IoT context and in the light of what we know today, both from the perspective of what the technology can offer and of what the applications require. Backward compatibility is important, but simply building more specifications on top of existing ones entails constraints that prevent using the most adequate solutions.

What we need is a global network of interconnected resources, but with interoperability technologies more adequate to humans, machines and dynamic networks. We designate this as the *Mesh*, to emphasize heterogeneity and dynamicity. We believe that it constitutes a natural match for the IoT, reflecting many of the characteristics of *mesh networks* [35].

Therefore, we propose to improve the IoT-on-Web scenario by adopting some basic principles that lead to an IoT-on-Mesh scenario, such as:

- *Structured references*. An URI is a mere string. References to resources should be structured data and include several addresses, so that a resource can reference another across various networks. In addition, a reference should not necessarily be limited to addresses, but should also be able to designate channels, directions (in mesh topologies) or even ID tokens, all in the context of the respective target networks or gateways, so that routing and indirect addressing are supported. The system should deal with the network in a virtual (or logical) way, with information on the actual network contained inside the structured references;
- *Efficiency*. Messages should be as short as possible and optimized for machine to machine interaction. This entails the possibility of sending only binary information, without names or other source information, whenever agreed, but maintaining the possibility of self-description information, whenever needed. Ideally, the level of source information sent should be automatically adjusted;
- *Human adequacy*. Description of resources need also to be adequate for people, the developers. JSON and REST are simpler than XML and SOA but less equipped for design-time support. The ideal would be to combine the simplicity of the former with the design-time support provided by the latter;
- *Layered protocol*. Protocols such as HTTP and SOAP include everything in one specification, placing the protocol at a relatively high level. The IoT is highly variable in this respect, which means that there should be a very simple protocol at the bottom, unreliable and asynchronous, with higher-level mechanisms separate and offered optionally or implemented by the application, in conjunction with exception handling;
- *Context-oriented approach*. This means that every message can have two components, functional (arguments to operations) and non-functional (context), corresponding to a separation between data and control planes in what could be designated a *software-defined IoT*. For example, context information can be used to specify the type of protocol to use when sending a message (reliable or unreliable) and also to get non-functional information in a response message;
- *Decoupling*. Web interaction at the service level is based on passive data resources (e.g., XML and JSON), with interoperability based on schema sharing (SOA, with WSDL) or pre-agreed media types (REST). This implies interoperability for all possible variants of service interaction and entails a higher level of coupling than required. We propose to use *partial interoperability*, based on the concepts of *compliance* and *conformance* (defined below).

These principles apply not only to machine-level but also to human-level interaction, although browsers would have to be adapted. The following sections describe the most relevant issues in further detail, establishing a relationship with currently available technologies.

19.5 The Structural Services style

19.5.1 Getting the best of SOA and REST

The main architectural styles [36] used today in distributed systems are SOA [9] and REST [10]. An architectural style can be defined as a set of constraints on the concepts and on their relationships [37].

Both SOA and REST have advantages and limitations, and our goal is to derive an architectural style that emphasizes the advantages and reduces the limitations:

- The SOA style allows any number of operations for each resource, whereas REST specifies that all resources must have the same set of operations. Real world entities (problem domain) have different functionality and capabilities. REST deals with this in the solution domain by decomposing each entity in basic components, all with the same set of operations, and placing the emphasis on structure. This simplifies access to components, making it uniform, but it also makes the abstraction models of entities harder to understand (hidden by the decomposition into basic components) and increases the semantic gap between the application's entities and REST resources. This is not adequate for all problems, namely those involving complex entities. We want SOA's flexibility in service variability, or the ability to model resources as close as possible to real world entities;
- The REST style allows structured resources, whereas SOA does not. Real world entities are usually structured and in many applications their components must be externally accessible. Just using a black-box approach (yielding a flat, one-level resource structure) limits the options for the architectural solution. We want REST's flexibility in resource structure.

Therefore, we propose the *Structural Services architectural style*, in which a resource has both structured state (resources as components) and service behaviour (non-fixed set of operations), all defined by the resource's designer. This means that a resource becomes similar to an object in object-oriented programming, but now in a distributed environment, with the same level of data interoperability that XML and JSON provide.

Table 19.2 summarizes the main characteristics of these architectural styles. The Structural Services style is a solution to several of the principles enunciated above, in particular the principles of human adequacy and structured references. The following sections illustrate how this style can be implemented.

Table 19.2. Main characteristics of architectural styles

Style	Brief description	Examples of characteristics or constraints
SOA	Style similar to Object-Oriented, but with Distributed constraints	Resources have no structure; Services are application specific (variable set of operations); No polymorphism; Integration based on common schemas and ontologies.
REST	Resources have structure but a fixed service	Client and server roles distinction; Services have a fixed set of operations; Stateless interactions.
Structural Services	Structured resources, variable services, minimal coupling, context awareness	Resources are structured (like REST); Services have variable interface (like SOA); Interoperability based on structural compliance and conformance (new feature); Functional and context-oriented interactions (new feature).

19.5.2 The need for new implementation mechanisms

A distributed system is, by definition, a system in which the lifecycles of the modules are not synchronized and, therefore, can evolve independently, at any time. Distribution does not imply geographical dispersion, although this is the usual case. Pointers become meaningless and distributed references, such as URIs, must be used. Assignment (such as in argument passing to operations) must have a copy semantics, since the reference semantics, common in modern programming languages, is not adequate for distributed systems.

Data description languages, such as XML and JSON, merely describe data and their structure. If we want to describe services, we can use WSDL (a set of conventions in XML usage), but the resulting verbosity and complexity has progressively turned away developers in favour of something much simpler, namely REST. If we want a programming language suitable for distributed environments we can use BPEL [38], but again with an unwieldy XML-based syntax that forces programmers to use visual programming tools that generate BPEL and increase the complexity stack. For example, a simple variable assignment, which in most programming languages would be represented as `x=y`, requires the following in BPEL:

```
<assign>
    <copy>
        <from variable="y" />
        <to variable="x" />
    </copy>
</assign>
```

JSON is much simpler than XML and, thanks to that, its popularity has been constantly increasing [39]. The evolution of the dynamic nature of the Web, as shown by JavaScript and HTML5 [14], hints that data description is not enough anymore and distributed programming is a basic requirement. In the IoT, with machine to machine interactions now much more frequent that human interaction, this is of even greater importance.

Current Web-level interoperability technologies are greatly constrained by the initial decision of basing Web interoperability on data (not services) and text markup as the main description and representation mechanism. This has had profound consequences in the way technologies have evolved, with disadvantages such as:

- Textual representation leads to parsing overheads (binary compression benefits communication but not processing overheads) and poor support for binary data;
- The flexibility provided by self-description (schemas) did not live up to its promise. XML-based specifications (WSDL, SOAP, and so on), although having a schema, are fixed by standards and do not evolve frequently. This means that schema overheads (in performance and verbosity of representation) are there but without the corresponding benefits;
- The self-description provided by schemas does not encompass all the interoperability levels described in Table 19.1, being limited mostly to the syntactic and in some cases semantic levels. Adequate handlers still need to be designed specifically for each specification, which means that data description languages take care of just some aspects and are a lot less general than they might seem at first sight;
- Interoperability is based on data schema sharing (at runtime or with previously standardized or agreed upon internet media types), entailing coupling for the entire range of data documents described by the shared schema, which is usually much more than what is actually used;
- Data description languages have no support for the pragmatic (service description and context awareness) and semantic (rules and ontologies) interoperability categories in Table 19.1, which must be built on top of them with yet other languages;
- The underlying protocol, usually HTTP, is optimized for scalability under the client-server model, whereas the IoT requires a peer to peer model with light and frequent messages.

Therefore, it is our opinion that different implementation mechanisms are needed to support the Structural Services style natively and efficiently in its various slants (instead of emulating them on top of a data-level description language such as it happens with WSDL, for example) and in various environments (such as the Web, IoT and non-IP networks).

19.5.3 Textual representation of resources

With respect to XML, JSON points the way in data representation simplicity, but it too has a compatibility load, in this case with JavaScript. Like XML, it is limited to data (no support for operations) and its schema is a regular JSON document with conventions that enable to describe the original JSON document. With both XML and JSON, a schema is a specification separate from the data (documents) it describes. A schema expresses the associated variability, i.e., the range of documents that are validated by that schema.

The basis of data interoperability with XML and JSON is schema sharing. Both the producer and consumer (reader) of a document should use the same schema, to ensure that any document produced (with that schema) can be read by the consumer. This means that both producer and consumer will be coupled by this schema.

This may be fine for document sharing, but not for services (in which the service description and the data sent as arguments are the documents), since typically a service provider should be able to receive requests from several consumers and each consumer must be able to invoke more than one provider.

Therefore, we use a different approach to represent resources, which can be described in the following way:

- The representation includes data, behaviour, context and self-description, in a format suitable and familiar for humans. In fact, it is a textual distributed programming language (SIL – Service Implementation Language), with an object-oriented look and feel;
- Self-description is done within the resource description itself (no separate schemas), through a design-time variability range. In fact, each resource has its own exclusive schema. At any time, each resource has a structured value, which can vary at runtime but must lie within the variability range, otherwise an error is generated;
- Interoperability is based on *structural typing* [40], in which a consumer can read a document, as long as it has a structure recursively interoperable (until primitive resources are reached) with what it expects. Section 19.6 details this aspect, including not only data but also service interoperability;
- Unlike XML and JSON, SIL has a dual representation scheme, source text and compiled binary, using TLV (Tag, Length and Value) binary markup. This is not mere data compression, but actual compilation by a compiler, which automatically synchronizes binary with source. The binary representation provides native support for binary data, has a smaller length and is faster to parse (the Length field enables breadth-first parsing), all very important for the small devices that pervade IoT applications.

Listing 19.1 shows a simple example of resources described in SIL. It includes a temperature controller (`tempController`), which is composed of a list of

references to temperature sensors with history (`tempSensorStats`), each of which has a reference to a remote temperature sensor (`tempSensor`). The lines at the bottom illustrate how to use these resources and should be included in some resource that uses them.

```
tempSensor: spid {  // descriptor of a temperature sensor
   getTemp: operation (-> [-50.0 .. +60.0]);
};

tempSensorStats: { // temperature sensor with statistics
   sensor: @tempSensor; // reference to sensor (can be remote)
   temp: list float;    // temperature history
   startStats <||;      // spawn temperature measurements
   getTemp: operation (-> float) {
      reply sensor@getTemp <--; // forward request to sensor
   };
   getAverageTemp: operation ([1 .. 24] -> float) {
      for (j: [temp.size .. temp.size-(in-1)])
         out += temp[j];
      reply out/in; // in = number of hours
   };
   private startStats: operation () {  // private operation
      while (true) {
         temp.add <-- (getTemp <--); // register temperature
         wait 3600;  // wait 1 hour and measure again
      }
   }
};

tempController: { // controller of several temperature sensors
   sensors: list @tempSensorStats;  //list of references to sensors
   addSensor: operation (@tempSensor) {
      t: tempSensorStats;  // creates a tempSensorStats resource
      t.sensor = in;       // register link to tempSensor
      sensors.add <-- @t;  // add sensor to list
   };
   getStats: operation (-> {min: float; max: float; average: float})
   {
      out.min = sensors[0]@getTemp <--;
      out.max = out.min;
      total: float := out.min;   // initial value
      for (i: [1 .. sensors.length-1) {   // sensor 0 is done
         t: sensors[i]@getTemp <--; // dereference sensor i
         if (t < out.min) out.min = t;
         if (t > out.max) out.max = t;
```

```
            total += t;
        };
        out.average = total/sensors.length;
        reply;   // nothing specified, returns out
    }
};

// Using the resources
// tc contains a reference to tempController
tc@addSensor <-- ts1; // reference to a tempSensor resource
tc@addSensor <-- ts2; // reference to a tempSensor resource
x: tc@sensors[0]@getAverageTemp <-- 10;   // average last 10 hours
```

Listing 19.1. Describing and using resources, using SIL (Service Implementation Language)

This listing can be briefly described in the following way:

- The temperature sensor (`tempSensor`) is remote and all we have is its SPID (SIL Public Interface Descriptor). This corresponds to a Web Service's WSDL, but much more compact and able to describe both structure and operations. It is automatically derived from the resource description by including only the public parts (public component resources and operation headings). The SPID of `tempSensorStats`, for example, is expressed by the following lines:

```
tempSensorStats: spid { // temperature sensor with statistics
    sensor: @tempSensor; // reference to sensor (can be remote)
    temp: list float;    // temperature history
    getTemp: operation (-> float);
    getAverageTemp: operation ([1 .. 24] -> float);
}
```

- Resources and their components are declared by a name, a colon and a resource type, which can be primitive (such as `integer`, a range, e.g., [10 .. 50], or `float`), or user defined (enclosed in curly brackets, i.e., "{. . .}"). Overall, there are some resemblances to JSON, but component names are not strings and operations are first class resources (in the line of REST, actually);
- The definition of a resource acts like a constructor, being executed only once, when a resource of this type is created, and can include statements. This is illustrated by the statement "`startStats <||`" in `tempSensorStats`. This is actually an asynchronous invocation ("`<||`") of private operation `startStats`, which is an infinite loop registering temperature measurements every hour. Synchronous invocation of operations is done with "`<--`", followed by the argument, if any;
- Operations have at most one argument, but it can be structured (with "{. . .}"). The same happens with the operation's reply value, such as illustrated by

operation `getStats`. Inside operations, the default names of the argument and the value to return are `in` and `out`, respectively. The heading of operations specifies the type of the argument and of the reply value (inside parentheses, separated by "`->`");

- References to resources (indicated by the symbol "`@`") are not necessarily URIs, not even strings. They can be structured and include several addresses, so that a resource in a network (e.g., the Internet) can reference another in a different network (e.g., a ZigBee network). It is up to the middleware in the network nodes that support these protocols to interpret these addresses, so that transparent routing can be achieved, if needed;
- Resource paths, to access resource components, use dot notation, except if the path traverses a reference, in which case a "`@`" is used. Note, for example, the path used in the last line of Listing 19.1, which computes the average temperature, along the last 10 hours, in sensor 0 of the controller.

19.5.4 Binary representation of resources

Listing 19.1 is what the programmer sees, a text program in a distributed programming language, with a syntax similar to usual and familiar object-oriented programming languages, instead of a verbose and cumbersome syntax of a data description language such as XML.

However, this is not necessarily what is used by the computing platform. A compiler transforms Listing 19.1 into binary code, like a Java compiler transforms Java source into bytecodes. This binary code is then interpreted, which guarantees its universality, as long as the interpreter is implemented in interacting computing nodes.

The binary representation uses a modified version of the TLV format (Tag, Length and Value) used by ASN.1 [41]. This not only supports the direct integration of binary information but also facilitates parsing, since each resource, primitive or structured, can be stepped over in a breadth-first traversal, thanks to the *Length* field (the resource size in bytes).

Resources (including messages) can be represented in three levels:

1. Binary, no variable names (components are referred to by their position index in the resource);
2. Variable names, in a dictionary that maps names to position indices;
3. Source text, when available. Typically, runtime messages are generated directly in binary, without source text. This is used mostly in design-time resources.

Levels 1 and 2 include the same information as level 3, apart from comments and formatting. Level 1 can be used without level 2 and is the most efficient and the most adequate for smaller devices in IoT. However, it requires a previous agreement

on the types of the resources involved in the interaction. Hardwired solutions, typical of old binary systems, should be avoided and therefore it must be possible to establish a mapping between corresponding components (with the same name) of interacting resources, even if they are in different order or not all are present in both resources. This is structural type checking [40] and needs to be done in distributed systems, in which the lifecycle of resources are not synchronized and a simple name is not enough to define a type.

To use just level 1 without losing type checking, both levels 1 and 2 should be sent in the first message, verify interoperability by checking resource types structurally (which takes time), establish a mapping between names and position indices and cache it, assigning an ID token to it. In subsequent messages with the same structure, this token can be used instead of performing a structural resource analysis again. This exploits the fact that many IoT devices, once installed, repeatedly use the same message structure for quite a long time. The interaction protocol must support this optimization and be able to recover from situations in which the token has changed, by signalling an error that forces retransmission of a message, now with components names, and generation of a new token. This mechanism bears resemblance to the Etag header of HTTP.

Table 19.3 gives an idea of the size of resource representations in various situations. The example refers to a data-only resource, so that XML and JSON can be used. The two lines refer to the same resource, but with longer and shorter component names. That is why the sizes in the last column are the same and the size reduction is greater when names are longer. Naturally, the sizes presented vary a lot with the concrete example, but this gives an idea of the values involved.

Table 19.3. Sizes (in bytes) of several resource representations

	XML	JSON	SIL Source	SIL binary with names	SIL binary without names
Longer names	2472	1491	1317	927	358
Shorter names	1857	1153	979	589	358

Compressed versions of XML and JSON [42] also accomplish relevant reductions in size, but time is needed to compress and to decompress messages, on top of which text parsing still needs to be done at the receiver. The compilation in SIL takes longer than a simple compression, but it does not need to be done at runtime. Compilation is a design time feature and messages sent are generated in binary directly. Messages received are parsed directly in binary, much faster than text parsing due to the navigable structure provided by the TLV scheme.

The interface between SIL and other languages is done in the same way as XML-based solutions, but with structure as the guiding mechanism. There is a mapping between each primitive data type in SIL and the corresponding one in another language. Structured resources in SIL are mapped onto classes. A dynamic load mechanism must be available to dynamically register new resources into a resource directory, so that their services become available in the network they connect to.

19.5.5 Message protocol

The typical reliability of Web messaging provided by TCP is a luxury in many IoT applications, in terms of network characteristics and node processing capabilities. The CoAP [28] addresses this issue by assuming unreliability and resorting to UDP and lighter messages. CoAP separates the basic message transport protocol, which can be unreliable, from the message level protocol, which deals with requests and responses.

SIL also follows the layered protocol approach, with a basic binary message transport on top of which several types of messages can be defined. The control part of the protocol also uses the TLV scheme. There are no assumptions about the transport level, apart from addressability and capability of transferring the message from the sender to the receiver. It can be unreliable and unsecure and it can use any protocol, IP or non-IP based. Resource references are not necessarily URIs. Anything can be used, as long as the messaging middleware is able to understand it. If reliability or security are needed but not provided by the transport protocol, these must be implemented by the middleware or the application.

The main message types of the SIL message protocol are described in Table 19.4. The context aspect is discussed in section 19.7.

Table 19.4. Main message types of the SIL message protocol.

Message category/type	Description
Request	*Initial request in each transaction*
React	React to message, no answer expected
React & respond	React to message and answer/notify
React with context	React to message and context, no answer expected
React with context & respond	React to message and context and answer/notify
Assimilate	Merge the message into the receiver resource, subject to structural interoperability (only compatible components are replaced). It bears similarities to PUT in REST.
Response	*Response to the request*
Answer	A (structured) value returned by a `reply` instruction
Answer with weak context	A (structured) value returned with context, handler at receiver optional
Answer with strong context	A (structured) value returned with context requiring a handler (includes exceptions)
Protocol fault	A status code resulting from a predefined protocol error (includes interoperability token mismatch)
Done	Request completed but has no value to reply

19.6 Improving interoperability with compliance and conformance

Although not an exclusive issue of the Structural Services style, the goal of reducing as much as possible the coupling between two interacting resources, while still providing the level of interaction required by the application, greatly benefits from a resource representation language such as SIL, which has native support for structural interoperability, based on *compliance* and *conformance*.

In a distributed system, any resource can change the name of the types it uses at any time, which means that name-based type checking to ensure interoperability requires lifecycle synchronization and hampers changeability. Yet, this is precisely what happens in current Web technologies, namely Web Services and RESTful applications:

- Schemas must be shared between interacting Web Services, establishing coupling for all the possible values satisfying each schema, even if they are not actually used. In this case, a reference to a schema acts like its name;
- REST also requires that data types (usually called media types) must have been previously agreed, either standardized or application specific;
- Searching for an interoperable Web Service is usually done by schema matching with similarity algorithms [43] and ontology matching and mapping [44]. This does not ensure interoperability and manual adaptations are usually inevitable.

The interoperability notion, as defined in this chapter, introduces a different perspective, stronger than similarity but weaker than commonality (sharing). The trick is to allow partial (instead of full) interoperability, by considering only the intersection between what the consumer needs and what the provider offers. If the latter subsumes the former, the degree of interoperability required by the consumer is ensured, regardless of whether the provider supports additional capabilities or not.

Interoperability of a consumer with a provider entails the following properties:

- *Compliance* [21]. The consumer must satisfy (*comply with*) the requirements established by the provider to accept requests sent to it, without which these cannot be honoured;
- *Conformance* [22, 45]. The provider must fulfil the expectations of the consumer regarding the effects of a request (including eventual responses), therefore being able to take the form of (*conform to*) whatever the consumer expects it to be.

In full interoperability, the consumer can use all the provider's capabilities. In partial interoperability, the consumer uses only a subset of those capabilities, which means that compliance and conformance need only be ensured for that subset.

These properties are not commutative. If *X* complies with *Y*, *Y* does not necessarily comply with *X*. However, they are transitive. For example, if *X* complies with *Y* and *Y* complies with *Z*, then *X* complies with *Z*.

Figure 19.3 illustrates this model. A resource *A*, in the role of consumer, has been designed for full interoperability with resource *B*, in the role of provider. *A* uses only the capabilities that *B* offers and *B* offers only the capabilities that *A* uses. Now, let us consider that we want to change the provider of *A* to resource *D*, which has been designed for full interoperability with resource *C*, in the role of consumer. The problem is that *A* was designed to interact with a provider *B* and *D* was designed to expect a consumer *C*. This means that, if we use *D* as a provider of *A*, *B* is how *A* views provider *D* and *C* is how *D* views consumer *A*.

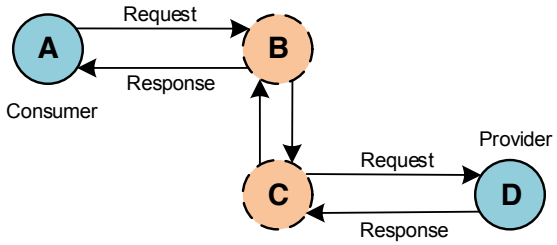

Fig. 19.3. Partial interoperability based on compliance and conformance

There are two necessary conditions to ensure that *A* is interoperable with *D*:

- *Compliance*. *B* must comply with *C*. Since *A* complies with *B* and *C* complies with *D*, this means that *A* complies with *D* and, therefore, *A* can use *D* as if it were *B*, as it was designed for;
- *Conformance*. *C* must conform to *B*. Since *D* conforms to *C* and *B* conforms to *A*, this means that *D* conforms to *A* and, therefore, *D* can replace (take the form of) *B* without *A* noticing it.

Partial interoperability has been achieved by *subsumption*, with the set of capabilities that *A* uses as a subset of the set of capabilities offered by *D*. This inclusion relationship, without changing characteristics, is similar in nature to the *polymorphism* used in many programming languages, but here in a distributed context. It constitutes the basis for transitivity in compliance and conformance.

Listing 19.2 illustrates these concepts by providing two additional temperature sensors (`weatherSensor` and `airSensor`) in relation to Listing 19.1. Only the additional and relevant parts are included here, with the rest as in Listing 19.1.

```
tempSensor: spid {
    getTemp: operation (->[-50.0 .. +60.0]);
};
```

```
weatherSensor: spid {
    getTemp: operation (->[-40.0 .. +50.0]);
    getPrecipitation: operation (-> integer);
};

airSensor: spid {
    getTemp: operation (->[-40.0 .. +45.0]);
    getHumidity: operation (-> [10 .. 90]);
};

// tc contains a reference to tempController
tc@addSensor <-- ts1; // reference to a tempSensor resource
tc@addSensor <-- ts2; // reference to a tempSensor resource
tc@addSensor <-- as1; // reference to an airSensor resource
tc@addSensor <-- ws1; // reference to an weatherSensor resource
tc@addSensor <-- ws2; // reference to an weatherSensor resource
tc@addSensor <-- as2; // reference to an airSensor resource
x: tc@sensors[0]@getAverageTemp <-- 10;   // average last 10 hours
s: {max: float; average: float};
s = tc@getStats <--; // only max and average are assigned to s
temp: [-50.0 .. +50.0];
temp = ws1@getTemp <--; // complies. Variability ok
temp = ts1@getTemp <--; // does not comply. Variability mismatch
```

Listing 19.2. Example of partial interoperability, with structural compliance and conformance

Note that:

- `weatherSensor` and `airSensor` conform to `tempSensor`, because they offer the same operation and the result is within the expected variability. This means that they can be used wherever a `tempSensor` is expected, which is illustrated by adding all these types of sensors to `tempController` (through `tc`, a reference to it) as if they were all of type `tempSensor`. Non relevant operations do not matter;
- The result of invoking the operation `getStats` on `tempController` is a resource with three components (see Listing 19.1), whereas `s` has only two. But the assignment is still possible, thanks to structural interoperability (the extra component is ignored);
- The last statement in Listing 19.2 also triggers a compliance check by the compiler, which issues an error. The variability of the result value of operation `getTemp` in `tempSensor` (referenced by `ts1`) is outside the variability range declared for component `temp`.

In addition, it is important to note that compliance and conformance could also be implemented with XML and JSON (in what data are concerned, since they do not support operations). It would be just a matter of comparing descriptions. However, technology has not evolved that way. One of the reasons is that performing a structural compliance and conformance check in every message would be too slow. Therefore, they chose to stick to full interoperability, with schemas shared and media types previously agreed, but with more coupling that actually needed and lower changeability.

Practical implementation of partial interoperability requires an optimization mechanism, such as the token mentioned in section 19.5.4, and a protocol to support it, as described in section 19.5.5.

19.7 Improving interoperability with context awareness

Non-functional information, or *context*, is very important in many applications, in particular in IoT environments with mobile applications that extract value from context awareness (such as location and local settings). If there is no support for it, functional parameters can become cluttered with contextual information.

Most context-oriented programming languages support context adaptation by activating and deactivating layers [46]. A context layer is a set of operations that deal with a specific context (or scenario) and that, when that layer is activated, automatically extend or replace operations that are already defined in various classes. The layer's operations can be scattered throughout the classes which they pertain to (*layer-in-class*) or declared together, in a specific construction separate from classes (*class-in-layer*), but achieving the same effect. When a layer is active, the behaviour of some operations is modified. Typically, this is a control-only feature, with no support for storing contextual state.

SIL includes support for context-oriented programming in a smaller granularity and with provision for context state, by allowing operations to have two parameters and two results (functional and non-functional). Exceptions have been integrated in the context-passing mechanism, which can be described in the following way:

- *Forward context*. A resource can be passed to an operation as non-functional parameter, together with a functional parameter, by using a *with* clause. This can be used to adapt the behaviour of the message recipient. If several types or values of context are expected, yielding different behaviour, there can be multiple implementations of the corresponding operation, each specifying which context type it accepts (with structural compliance). When a message with a specified context is received, the SIL environment invokes the first of the implementations of the target operation that declares a context with which the message's context complies;

- *Backward context*. SIL returns values by executing `reply`, `reply...with` and `raise...with` statements. The first returns a functional value and the others add a context value, weak and strong, respectively. The resource with receives the returned value can use a `do...catch` statement to handle context values. The difference between `reply` and `raise` is that the former invokes the `catch` handler (or ignores if the context does not comply with any `catch` handler) and resumes execution, as if there was no context, and the latter breaks execution and backtracks as much as needed until a handler is found (this includes exceptions).

Listing 19.3 provides a simple illustration of some of these features, in which sensors have been modified to exploit context information. In particular, it is possible in some cases to specify whether we want the temperature in degrees Celsius or Fahrenheit and in others to detect which temperature scale applies to returned values, simply be reading the backward context. Only the relevant parts are shown. The rest is as in Listings 19.1 and 19.2.

```
tempSensor: spid {
   getTemp: operation (-> [-50.0 .. +60.0] with "celsius");
};

weatherSensor: spid {
   getTemp: operation (-> [-40.0 .. +50.0] with "celsius");
   getTemp: operation (with "celsius" -> [-40.0 .. +50.0]);
   getTemp: operation (with "fahrenheit" -> [-58.0 .. +140.0]);
   getPrecipitation: operation (-> integer);
};

airSensor: spid {
   getTemp: operation (-> [-58.0 .. +140.0] with "fahrenheit");
   getHumidity: operation (-> [10 .. 90]);
};

// ts has a reference to a tempSensor
// ws has a reference to a weatherSensor
// as has a reference to a airSensor
//
do {
   tempC: float;     // to hold a celsius temperature
   tempC = ts@getTemp <--;
   tempC = ts@getTemp <-- with "celsius";
   tempC = ts@getTemp <-- with "fahrenheit"; // invokes catch
   tempC = ws@getTemp <--; // weatherSensor: returns celsius
   tempC = as@getTemp <--;    // airSensor: invokes catch
}
```

```
catch (tf: float with "fahrenheit") {
   tc: [-50.0 .. +60.0];
   tc = tf * 1.8 + 32;
   proceed tc;    // replaces value and type of message response
};
```

Listing 19.3. Example of forward and backward context awareness

`tempSensor` can only provide temperature in Celsius degrees (it does not accept indication of another scale), but at least informs the consumer (which invoked `getTemp`) of that fact, by returning the string "celsius" as a backward context.

`weatherSensor` is more elaborate and has three implementations of `getTemp`, allowing to specify the scale or to omit that indication (in which case it assumes Celsius degrees, by default). To obtain a temperature there is no need to specify a functional parameter, but the context can be specified. The adequate operation will be chosen at runtime, according to the forward context.

`airSensor` can only provide temperature values in degrees Fahrenheit.

The `do...catch` statement is similar to the `try...catch` statement in Java, but the `try` has been replaced by `do` because it is more encompassing than an exception handling mechanism. Inside the `do` clause, forward context can be specified by using a `with` clause, passing a value that must comply with the `with` clause of the invoked operation.

When the backward context complies with the `catch` clause (in this case, when the context string is "fahrenheit"), the value part of the reply is assigned to the specified component and the corresponding code executed. In this case, it computes a temperature value in degrees Celsius from the original value in degrees Fahrenheit and executes the `proceed` statement with the new value, which now acts as the reply and is finally assigned to `tempC`.

This shows how a backward context can be used to modify a reply value on the fly and increase context awareness and range of interoperability, since now more sensors can be used with `tempController` due to automatic context-oriented adaption. In particular, note the last statement in the `do` clause, in which no temperature scale was specified but the returned value could still be used and correctly converted to degrees Celsius because it was accompanied by contextual information.

19.8 Discussion and rationale

There are several signs that show that global interoperability is shifting away from classical client-server HTTP-based interactions, initially conceived for human consumption in what has become known as the Web. Namely:

- The IoT is changing the emphasis to machine-to-machine interactions, using the peer to peer model. Although the most immediate solution is to use existing Web Technologies (in most cases REST, due to its simplicity), these are not the most adequate model for IoT. The mere development of CoAP [28] is an acknowledgment of this. The IoT is also bringing heterogeneity of resources and networks, and this is why we contend that the Web (rather homogeneous at the platform level) will evolve into a Mesh, as discussed in section 19.4;
- The need for more dynamic applications than simple browsing have spurred the development of technologies such as AJAX [47] and Comet [48]. More recently, the WebSockets protocol [14] paved the way for full-duplex communication at the Web level. Even more recently, there is a movement towards implementing real-time communication directly between browsers, without the need for a server, with WebRTC [13];
- Cloud computing applications are changing the overall computing scenario, with an increasing emphasis on distributed programming and support for asynchrony, concurrency and handling of unreliability. Languages such as Orc [49] for distributed task orchestration, Jolie [50] for distributed service orchestration and, more recently, Orleans [51] for distributed programming in cloud environments, show that there is the need and the capability of implementing distributed applications in a more consistent way than merely resorting to current Web technologies, such as those used by SOA and REST.

Backward compatibility with existing technologies is an asset that usually is half way to success, but it can also be a straightjacket, with constraints and compromises that increase complexity and hamper performance. Therefore, we need to revisit the global interoperability problem, in the light of the needs of new applications (typically, based on services rather than on documents) and of the capabilities that new technologies can offer. Then, we can try to find a way to migrate from existing technologies to those that are found to be more suitable to tackle the new classes of problems.

This is our justification to conceive the Structural Services architectural style, trying to reap the benefits from both SOA and REST, and to design SIL not as a passive data description language but as an active service implementation language, with the corresponding execution platform [52].

This can be illustrated by considering how service discovery, or the equivalent of UDDI, is done in SIL. There is a directory service, which resources available to provide services can be registered with. This is a regular service, not a special mechanism, and it can register resources within a local computing node or available remotely. The directory maintains a list of references to registered resources. When we want to search for a service, we supply the directory with the resource SPID that describes the service that we want and the directory returns a list of references to the all registered resources that conform to that service. Conformance includes context information, so that we can specify both functionality and non-functional

aspects and constraints, such as security mechanisms, SLA, cost, and so on. This is how the references used in Listings 1 through 3 can be obtained.

In terms of programming, the most used solutions are external programming languages (e.g., a Web Service or a REST resource implemented in Java) and WS-BPEL, an XML-based language. Services are described separately (e.g., in a WSDL file). Orc and Jolie are more concerned with scheduling and control of processing activities than with interoperability (for which they resort to conventional solutions, such as Web Services and XML). Several languages and specifications are needed to implement the full solution.

SIL, and its execution platform, is complete in the sense that one single language is enough to describe, implement and execute services, with their state structure and behaviour. A complete distributed application can be implemented in SIL. On the other hand, interoperability is supported, so that native operations can be implemented not in SIL but in a native language, such as Java or C#, and different native resources, implemented in different languages, are able to interoperate through a SIL interface (SPID).

Therefore, the integration benefits of Web Services and RESTful applications are not lost. Nevertheless, SIL is a new solution, not following an evolutionary approach. The migration path from current technologies can be established in two main ways:

- *Co-existence*. The SIL server receives binary messages and does not depend on any particular transport protocol, relying solely on message delivery. Therefore, any existing server can be used, based on HTTP, WebSockets or any other protocol. In fact, several servers can be used simultaneously, receiving messages that are handed over to message handlers. The SIL handler first checks if this is an identifiable SIL message (in TLV format). If not, the full message can then be inspected by other handlers, which implement current technologies;
- *Conversion*. Although space limitations have prevented us from discussing SIL in full detail, it has been conceived to support the features provided by XML, JSON and the corresponding schema languages, which means that any XML, WSDL or JSON file can be converted to SIL, allowing current clients to use a service in a SIL server.

19.9 Future research directions

Currently, there is a partial implementation of the SIL language and environment, with basic functionality [52]. Future work will be carried out along the main following lines:

- Semantics. SIL supports semantics through rule resources (a set of condition-then-action statements), which become active upon creation in the scope of their container resource and get evaluated each time a component affecting one of the

conditions is changed. It is up to the compiler to establish dependencies between conditions and components. This mechanism needs to be optimized;
- The compliance and conformance algorithms are implemented in the compiler but these need to be optimized in the binary version, to increase the performance of dynamic type checking;
- The interface to native languages needs to be diversified and improved. At the moment, only Java is supported;
- A quantitative comparison assessment of the SIL platform versus Web Services and RESTful solutions needs to be carried out in IoT applications (particularly in non-IP networks).

19.10 Conclusions

The IoT is changing the scenario of global interoperability. From a rather homogeneous, reliable and static network of servers that we identify as the Web, built on top of the Internet, we are moving to what we call a *Mesh*, a heterogeneous, unreliable and dynamically reconfigurable network of computing nodes, ranging from stationary, generic and fully-fledged servers to mobile, application-specific and low-level devices, such as sensors. Usually, the simpler a node is, the larger the quantities in which it is used, which leads to an enormous increase of the number of messages exchanged and raises a big data problem.

This is shifting the emphasis of the rather human level of the Web to the rather machine level of the Mesh, built on top of the IoT. The current approach to implement the IoT vision [4] is to extend current Web technologies to small devices. We contend that, although evolutionary and faster to implement, this is not the most adequate to the nature of IoT.

In fact, even the "I" in IoT needs clarification. Our understanding of the IoT concept is not a network of devices connected by IP-based protocols, such as CoAP (on UDP) and HTTP (on TCP), but a network of networks, usually heterogeneous in terms of protocols. In this respect, a sensor should be reachable globally (from any other device connected to the IoT) and directly (which implies transparent gateways, not application specific, between networks).

The main tenets of our approach are:

- To reduce the lowest level of interoperability to binary messages, with a very basic protocol and no source information, to reduce the computing power, memory and energy requirements of simpler devices, which repeatedly use the same message types and patterns;
- To maintain the capability of source-level interoperability, for flexibility in more dynamic systems;

- To increase interoperability by basing it on the compliance and conformance concepts, which support partial interoperability, and context awareness, by automatic adaptation of behaviour;
- To reduce complexity (the main drive behind the success of JSON and REST), by designing a language that can describe, implement and execute services, without the need of a plethora of languages and specifications to implement distributed applications.

We have presented several examples that illustrate how this can be done, by supporting distributed resources and services natively, instead of emulating them on top of data description languages.

Our main scientific contributions are the following:

- A systematization of the resource interoperability problem, by providing a resource interaction model (Fig. 19.1) and an interoperability abstraction scale (Table 19.1);
- A new architectural style, Structural Services (Table 19.2), which combines the structural capability of REST with the functional variability of SOA, with a platform-independent execution model and a language well matched to this style;
- New ways of supporting Big Data in the IoT, namely in terms of efficiency (for volume and velocity) and decoupling (for variety, or heterogeneity), with:
  - Native support for binary data;
  - Variable source information (which can be omitted in frequent cases);
  - Coupling reduced to the bare minimum required by the application, due to the use of structural interoperability (compliance and conformance) instead of schema sharing;
  - Resource-based contextual information, which provides a better decoupling than the traditional approach of activating context layers since the context sent as part of a message is data also subjected to structural interoperability.

We hope that our approach is a small contribution to the goal of implementing the IoT vision in a simpler but more encompassing and efficient way.

References

1. Zikopoulos P et al (2012) Understanding big data. McGraw-Hill, New York
2. Demchenko Y, Zhao Z, Grosso P, Wibisono A, de Laat C (2012) Addressing Big Data challenges for Scientific Data Infrastructure. Proc. IEEE 4th International Conference on Cloud Computing Technology and Science, pp.614-617. Taipei, Taiwan
3. Karen T (2013) How Many Internet Connections are in the World? Right. Now. http://blogs.cisco.com/news/cisco-connections-counter/. Accessed 28 August 2013

4. Sundmaeker H, Guillemin P, Friess P, Woelffle S (2010) Vision and challenges for realising the Internet of Things. European Commission Information Society and Media. http://bookshop.europa.eu/en/vision-and-challenges-for-realising-the-internet-of-things-pbKK3110323/. Accessed 28 August 2013
5. United Nations (2013) World Population Prospects: The 2012 Revision, Key Findings and Advance Tables. Working Paper No. ESA/P/WP.227. United Nations, Department of Economic and Social Affairs, Population Division. http://esa.un.org/unpd/wpp/Documentation/pdf/WPP2012_%20KEY%20FINDINGS.pdf. Accessed 28 August 2013
6. Schaffers H et al. (2011) Smart cities and the future internet: towards cooperation frameworks for open innovation. In: Domingue J et al. (eds.) The future internet, pp. 431-446. Springer Berlin Heidelberg
7. Bolton F (2001) Pure Corba. SAMS Publishing, Indianapolis
8. Venkitachalam G, Chiueh T (1999) High performance common gateway interface invocation. Proc. IEEE Workshop on Internet Applications, pp. 4-11. San Jose, California
9. Earl T (2005) Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ
10. Webber J, Parastatidis S, Robinson I (2010) REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media, Sebastopol
11. 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
12. Loreto S, Romano S (2012) Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. IEEE Internet Comput 16(5):68-73
13. Johnston A, Yoakum J, Singh K (2013) Taking on WebRTC in an Enterprise. IEEE Commun Mag 51(4):48-54
14. Lubbers P, Albers B, Salim F (2010) Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development. Apress, New York
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. 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
18. 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
19. 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
20. 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, pp. 97-129. Springer, Berlin
21. Kokash N, Arbab F (2009) Formal Behavioral Modeling and Compliance Analysis for Service-Oriented Systems. In: Boer F, Bonsangue M, Madelaine E (eds.) Formal Methods for Components and Objects, Lecture Notes in Computer Science, 5751, pp. 21-41, Springer-Verlag, Berlin, Heidelberg
22. Adriansyah A, van Dongen B and van der Aalst W (2010) Towards robust conformance checking. Proc. Business Process Management Workshops, pp. 122-133, Springer Berlin Heidelberg, 2010
23. Perera C, Zaslavsky A, Christen P, Georgakopoulos D (2012) Ca4iot: Context awareness for internet of things. Proc. IEEE International Conference on Green Computing and Communications, pp. 775-782. Besançon, France

24. Kapitsaki G, Prezerakos G, Tselikas N, Venieris I (2009) Context-aware service engineering: A survey. J Syst Softw 82(8):1285-1297
25. Gislason D (2008) Zigbee Wireless Networking. Elsevier, UK
26. Hui J, Culler D (2010) IPv6 in Low-Power Wireless Networks. Proc IEEE 98(11):1865-1878
27. 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, pp. 191-213. IGI Global, Hershey
28. 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
29. Balasubramaniam S, Kangasharju J (2013) Realizing the Internet of Nano Things: Challenges, Solutions, and Applications. IEEE Comp 46(2):62-68.
30. Akyildiz I, Brunetti F, Blázquez C (2008) Nanonetworks: A new communication paradigm. Comp Netw 52(12):2260-2279
31. Moustafa H, Zhang Y (2009) Vehicular networks: techniques, standards, and applications. Auerbach publications, Boston
32. ISO/IEC/IEEE (2010) Systems and software engineering – Vocabulary. International Standard ISO/IEC/IEEE 24765:2010(E), First Edition, p. 186. Geneva, Switzerland
33. Sheth A, Henson C, Sahoo S (2008) Semantic sensor web. IEEE Internet Comput 12(4):78-83
34. Palm J, Anderson K, Lieberherr K (2003) Investigating the relationship between violations of the law of demeter and software maintainability. Proc. Workshop on Software-Engineering Properties of Languages for Aspect Technologies. http://www.daimi.au.dk/~eernst/splat03/papers/Jeffrey_Palm.pdf. Accessed 28 August 2013
35. Bruno R, Conti M, Gregori E (2005) Mesh networks: commodity multihop ad hoc networks. IEEE Commun Mag 43(3):123-131
36. Dillon T, Wu C, Chang E (2007) Reference architectural styles for service-oriented computing. In Li, K. et al. (eds.) IFIP international conference on Network and parallel computing. Lecture Notes in Computer Science 4672, pp. 543–555. Springer-Verlag Berlin, Heidelberg.
37. Fielding R, Taylor R (2002) Principled Design of the Modern Web Architecture. ACM Trans on Internet Technol 2(2):115–150
38. Juric M, Pant K (2008) Business Process Driven SOA using BPMN and BPEL: From Business Process Modeling to Orchestration and Service Oriented Architecture. Packt Publishing, Birmingham, UK
39. Severance C (2012) Discovering JavaScript Object Notation. IEEE Comp 45(4):6-8
40. Gil J, Maman I (2008) Whiteoak: Introducing structural typing into Java. ACM Sigplan Notices 43(10):73-90
41. Dubuisson O (2000) ASN.1 Communication Between Heterogeneous Systems. Academic Press, San Diego, CA
42. 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
43. Jeong B, Lee D, Cho H, Lee J (2008) A novel method for measuring semantic similarity for XML schema matching, Expert Syst with Appl 34:1651–1658
44. Euzenat J, Shvaiko P (2007) Ontology matching. Berlin. Springer
45. Kim D, Shen W (2007) An Approach to Evaluating Structural Pattern Conformance of UML Models. Proc. ACM Symposium on Applied Computing, pp. 1404-1408, ACM Press.
46. Salvaneschi G, Ghezzi C, Pradella M (2012) Context-oriented programming: A software engineering perspective. J Syst Softw 85(8):1801-1817
47. Paulson L (2005) Building rich web applications with Ajax. IEEE Comp 38(10):14-17
48. McCarthy P, Crane D (2008).Comet and Reverse Ajax: The Next-Generation Ajax 2.0. Apress, New York
49. Kitchin D, Cook W, Misra J (2006) A language for task orchestration and its semantic properties. Proc. 17th International Conference on Concurrency Theory, pp. 477-491, Springer Berlin Heidelberg.

50. Montesi F, Guidi C, Lucchi R, Zavattaro G (2007) Jolie: a java orchestration language interpreter engine. Electron Notes Theor Comp Science 181:19-33
51. Bykov S et al (2011) Orleans: cloud computing for everyone. Proc. 2nd ACM Symposium on Cloud Computing, p. 16, ACM Press
52. Delgado J (2013) Service Interoperability in the Internet of Things. In Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence, pp. 51-87, Springer Berlin Heidelberg

INDEX