* Binary

Many developers found SOAP cumbersome and hard to use. For example, working with SOAP in JavaScript means writing a ton of code to perform extremely simple tasks because you must create the required XML structure absolutely every time. one perceived disadvantage is the use of XML because of the verboseness of it and the time it takes to parse.

REST doesn’t have to use XML to provide the response. You can find REST-based Web services that output the data in Command Separated Value (CSV), JavaScript Object Notation (JSON) and Really Simple Syndication (RSS). The point is that you can obtain the output you need in a form that’s easy to parse within the language you need for your application. While this may seem like it adds complexity to REST because you need to handle multiple formats, JSON usually is a better fit for data and parses much faster. REST allows better support for browser clients due to its support for JSON.

REST is more efficient because SOAP uses XML for all messages, REST can use smaller message formats so it is better to use when information about objects doesn’t need to be communicated to the client. It is beeter to use

using a binary message format resulting from compilation of the source program and that uses self-description information only when needed.

* Textual representation leads to parsing overheads (binary compression benefits communication but not processing overheads) and poor support for binary data
* 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.
* A compiler transforms Listing 19.1 into binary code, like a Java compiler transforms Java source into byte codes. 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).

native support for binary data (eliminating the need for encoding or compression).

* A dual representation format: text, a distributed programming language, and binary, obtained by compiling the text source and using TLV (Tag, Length and Value) binary markup (Dubuisson, 2000). Messages can include only source, binary with names and only binary
* Platform
* Web Sockets, now part of the HTML5 world (Lubbers, Albers & Salim, 2010), removes this restriction, adds binary support and increases performance.   XML is verbose and complex, has limited support for binary formats, is inefficient in computer terms due to parsing and exhibits symmetric interoperability, based on both sender and receiver using the same schema, which constitutes a relevant coupling problem.
* Other factors, such as a connectionless protocol (HTTP) and the lack of native support for binary data and contextual information, are limiting for many applications, although not impeditive.
* Interoperability

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.

The basis of data interoperability with XML and JSON is schema sharing(at runtime or with previously standardized or agreed upon internet media types). 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.

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.

In our solution, we propose to use *partial interoperability*, based on the concepts of *compliance* and *conformance*. It introduces a different perspective, stronger than similarity but weaker than commonality (sharing). The trick is to allow partial interoperability, by considering only the intersection between what the consumer needs and what the provider offers. It allow us for increased interoperability,  adaptability and changeability, without the need to have resource types necessarily shared or previously agreed.

This architectural style for integration, Structural Services, which combines the behavior flexibility of SOA with the structural hypermedia capabilities of REST and structural interoperability based on compliance and conformance.

In summary, SOA is a good option for large-grained, slowly evolving complex resources. REST is a good option for small-grained, structured resources, which are relatively simple. In search for simplicity and maintainability, many providers (including cloud computing providers, such as Rackspace) have stopped using SOA APIs in favor of REST, by modeling operations as resources, using naturally occurring lists of resources as structured resources with links and dynamically built URIs instead of operation parameters. Decoupling in REST, however, is not as good as its structure dynamicity seems to indicate. All the resource types (media types, in REST and Internet parlance) must be known in advance and the client may break if a new media type unknown to the client is used by the server. In addition, even when the type is known, the client cannot invent what to do and which link to choose when it receives a resource representation of a given type. The pragmatic layer (Table 1) requires behavior and its effects to be implemented at both the server and the client, and these cannot be agnostic of each other.