Why is the Web Loosely Coupled?  
A Multi-Faceted Metric for Service Design

Cesare Pautasso  
Faculty of Informatics University of Lugano  
6900 Lugano, Switzerland  
cesare.pautasso@unisi.ch

Erik Wilde  
School of Information  
UC Berkeley  
dret@berkeley.edu

ABSTRACT

Loose coupling is often quoted as a desirable property of systems architectures. One of the main goals of building systems using Web technologies is to achieve loose coupling. However, given the lack of a widely accepted definition of this term, it becomes hard to use coupling as a criterion to evaluate alternative Web technology choices, as all options may exhibit, and claim to provide, some kind of “loose” coupling effects. This paper presents a systematic study of the degree of coupling found in service-oriented systems based on a multi-faceted approach. Thanks to the metric introduced in this paper, coupling is no longer a one-dimensional concept with loose coupling found somewhere in between tight coupling and no coupling. The paper shows how the metric can be applied to real-world examples in order to support and improve the design process of service-oriented systems.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—Decision Tables; D.2.8 [Software Engineering]: Metrics—Software Science; H.3.5 [Information Storage and Retrieval]: Online Information Services—Web-Based Services

General Terms

Design, Measurement

Keywords

Loose Coupling, Tight Coupling, Web Services, SOA, REST, WS-*., RPC, HTTP

1. INTRODUCTION

One of the defining properties of Web services and service-oriented architectures is their “loose coupling” [20]. Loose coupling has a positive connotation as it implies that services share only a small set of assumptions and therefore the impact of change is limited, and services can evolve independently [18]. A loosely coupled service-oriented system is thus easy and cheap to evolve and has the potential to grow as rapidly and scale as easily as the Web. Beyond this point, however, there is little agreement upon what the term “loose coupling” actually means in the context of specific Web technologies (e.g., [3, 9, 27, 19]).

Taking the ongoing SOAP vs. REST debate [31] as an example, the argument of being “more” or “less” loosely coupled has been brought forward by each side. For example, giving an explicit description of a service interface in a WSDL document can be regarded as loose coupling, because it enables the interoperability of clients with services implemented in any programming language. However, it can also be regarded as tight coupling, because changes made to the WSDL may break clients that are built using code automatically generated from an earlier version of the service description. Given the lack of a clear definition of the concept of coupling, as we are going to discuss, different interpretations are possible. Each side is therefore, in its own view, correct in labeling the other as the one fostering a “tightly coupled” approach to the design of service-oriented systems.

This paper explores how to apply the concept of coupling to the design of service-oriented systems in greater detail, defines what it means to do so, and uses this definition in a way which leads to a better understanding of the different semantics people refer to when they mention loose and tight coupling. Our goal is to make explicit the various semantics that are often implicitly associated with the term, so that alternative interpretations become clear, and Web service technologies can be evaluated and compared in terms of the multi-faceted coupling metric we introduce in this paper.

Through a systematic study, we have collected 12 facets which can be associated with coupling related to specific aspects of Web technologies. This paper gives a definition of each facet, and discusses their relationships and interdependencies based on concrete examples. We also apply our multi-faceted definition to evaluate the coupling of existing Web technologies and Web services frameworks (remote procedure calls over HTTP [2, 13] vs. the RESTful usage of the HTTP protocol [12, 14] vs. a WS-* compliant enterprise-service bus [6]). We present our findings indicating that whereas current technologies can be considered loosely coupled as far as they provide platform independent solutions, they achieve a different degree of coupling according to all of the other facets. In particular, no technology per-se provides loose coupling in terms of all the facets. Also, we show that the degree of coupling of a service-oriented system does not fully depend on the choice of the underlying Web services platform. As far as certain facets are concerned (i.e., granularity and evolution), loose coupling is not influenced by the choice, e.g., between REST or WS-* but instead depends on the outcome of more specific design decisions.
This paper is organized as follows. We first introduce some related work (Section 2) and discuss the origins of the term (Section 3) to motivate the need for a better definition of loose coupling. Section 4 enumerates twelve facets of coupling giving concrete examples on how they affect Web technologies. Section 5 evaluates our multi-faceted definition to measure the coupling of concrete technology examples. Section 6 concludes the paper.

2. RELATED WORK

Most authors recognize that the “notion of designing services to be loosely coupled is the most important, the most far reaching, and the least understood service characteristic” [27, p. 75]. Loose Coupling has been named the “secret sauce of service-orientation” [3, Chapter 5]. Also, the “need for loose coupling” is somewhat self-evident and the “notion of loose coupling is a fundamental underpinning of SOA” [41, p. 10]. However when it comes to defining precisely what characteristics of coupling are the most significant for the design of service-oriented systems, only a limited consensus emerges from the literature.

Zimmermann et al. [47, p. 157] use loose coupling implicitly as a synonym of flexibility by means of information hiding: “a client application is not tightly coupled with a server, but coded against an abstract service description”. This definition resonates with the following:

“The phrase loosely coupled describes enterprise services’ characteristic of interacting in well-defined ways without needing to know each other’s inner workings. This means that the service’s functionality can change without affecting the services that use it, as long as the behavior described in its interface remains the same—that is, as long as it continues providing the functionality it provides.” [44, p. 111]

Delivering loose coupling by means of contracted interfaces is also recommended by Bloomberg and Schmelzer [3, p. 90]. This view is related to the “forgiving nature of the Web”, which enables browsers to work together with Web servers developed by different vendors. A similar definition of interface coupling is provided by Newcomer and Lomow [27]. The same authors also mention the importance of using service interfaces (so they are not coupled to a single process) as well as the danger of coupling clients and services to a specific platform (i.e., to avoid vendor lock-in). Platform coupling is also mentioned by Weerawarana et al. [41]. However, the authors stress that “a message-based approach fosters loose coupling” and “Web service technology is built on the concept of [asynchronous] messaging”. Hence, Web service technology is loosely coupled, as opposed to distributed object middleware platforms based on synchronous interactions. The tightly coupled nature of RPC-style Web services is discussed by Holpe [18], where message-oriented middleware is presented as the loosely coupled solution to enterprise application integration problems.

Kaye [20, Chapter 10] discusses the implications of tight vs. loose coupling on the following aspects: interaction, messaging style, message paths, technology mix, data types, syntactic definition, bindings, semantic adaptation, software objective, and consequences. Krafig et al. [23, Chapter 3] also take a multi-level approach to describe coupling in distributed systems. These are: physical level (whether services are directly or indirectly connected), platform (i.e., OS and programming language) dependency or independency, communications style (synchronous vs. asynchronous), type system (strongly typed interfaces vs. weakly typed payloads), interaction patterns (chatty distributed objects vs. message bus), process control (centralized vs. distributed), and discovery (static binding vs. dynamic binding).

Whereas the classifications developed by Kaye and Krafig are related to the multi-faceted definition we are going to present in this paper, we have revisited their classifications in the context of the SOAP vs. REST debate, entirely focusing on service design issues (i.e., the technical questions regarding loose vs. tight coupling), while the other classifications also include considerations that are more related to the implementation, deployment, and management aspects of services.

3. ORIGINS

In computer science, the notion of coupling predates the emergence of service-oriented computing. Some early usage examples can be found in distributed systems and software engineering design principles. In distributed systems, loosely coupled architectures are distinguished from “closely coupled” [36] ones based on whether two processes may communicate through some form of shared memory (close) or may only rely on message passing (loose). In practice, loose coupling in terms of space, time, and synchronization has been associated the properties of distributed systems designed according to the publish/subscribe paradigm [10]. In software engineering, the basic design principle of modularity implies the decomposition of a software architecture into modules characterized by high cohesion and low coupling [39, 45, 26]. Thus, coupling “measures the interdependencies of two modules” [15, 5] and it can be understood that a low degree of coupling is beneficial to aid the understanding and support the evolution of a system [25].

Even though the term “loose coupling” is often associated with software architectures (and was introduced into computer science as early as 1974 [35]), its origins can be traced back even earlier, leading to research on organizational structures as early as 1967 [37]:

“The concept [of loose coupling] has a rare combination of face validity, metaphorical salience, and cutting-edge mysticism, all of which encourage researchers to adopt the concept but do not help them to examine its underlying structure, themes, and implications. [. . . ] Because the concept has been underspecified, its use has generated controversy. Researchers who oppose the concept on the basis of its imprecision have watched as more and more researchers adopt it. Researchers who advocate the concept on the basis of its face validity have watched it become unrecognized. Researchers who are in the middle have often used the concept hesitantly, convinced that it fits the phenomena they study, but uncertain about its meaning.” [29]

By applying this quote from organization research to the context of service-oriented systems, it is easy to realize how marketing buzzwords sweep through the IT industry and gain acceptance thanks to the positive impression they deliver and thanks to their imprecise definition, and thus their
ability to be quoted in many different contexts. The problems which may cause tight coupling or which could be avoided by implementing loose coupling are the same in organizations and IT system architectures: the tension between the efficiency and safety of an internally determinate and completely rational structure, vs. the necessity to fit into an open world of uncertainty and conflicting concepts. Introducing tight coupling in complex systems may also affect their reliability, making accidents more prone to happen, because local failures are more likely to propagate [34]. In case of loose coupling, the following quote is relevant to illustrate how it can fit well with the design of service-oriented systems:

“A final advantage of coupling imagery is that it suggests the idea of building blocks that can be grafted onto an organization or severed with relatively little disturbance to either the blocks or the organization. [...] Thus, the coupling imagery gives researchers access to one of the more powerful ways of talking about complexity now available.” [42]

“Loose coupling” often is being perceived as referring to any intermediary state on a linear scale going from “no coupling” to “tight coupling”, with “loose coupling” being anywhere in between these two extremes. This conceptualization looks at coupling as a one-dimensional concept. Tenen and Weick [29] argue that the linear view of coupling is too constrained, and suggest to follow a two-dimensional approach.

<table>
<thead>
<tr>
<th>responsive</th>
<th>tightly coupled</th>
<th>loosely coupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-responsive</td>
<td>non-coupled</td>
<td>decoupled</td>
</tr>
<tr>
<td>non-distinctive</td>
<td>distinctive</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Two-Dimensional View of Coupling in Organizational Studies

The two-dimensional definition shown in Figure 1 has been developed within the context of organizational studies. But even for the goal of achieving loose coupling in the context of service-oriented systems design, this model can be useful to understand that a multi-dimensional approach to define the concept can lead to improved understanding. If the IT landscape for which services are being designed has distinctive components, then it is not realistic to look at tight coupling as the ultimate goal. If, however, for organizational, contractual, political, or legal reasons, it is possible to remove distinctiveness, it is possible to design and deploy a tightly coupled system. In the two-dimensional view of coupling, the question is how to achieve responsiveness (i.e., the ability for all business-level goals to be achieved by having services interact) given the design goals and/or the constraints of the environment.

Even with this two-dimensional view of coupling, however, it is still not completely clear what these two orthogonal axes (responsiveness, distinctiveness) refer to in the context of service-oriented systems design. Nowadays it is, for example, well-accepted that distinctiveness in terms of the physical network infrastructure is mostly irrelevant — the Internet provides a protocol stack which makes it unnecessary to deal with network protocol issues that were serious issues when computer networks were still dominated by vendor-specific protocol suites. This begs the question what kind of properties one should look for when evaluating a given technology according to the model shown in Figure 1. The next Section thus introduces a number of facets which we have identified as being important properties for deciding how a given service-oriented design should be evaluated to measure its coupling properties.

4. COUPLING FACETS

This section introduces a framework which can serve as a tool for analyzing the kind of coupling implied by a given technology choice during the design of a service-oriented system. From the previous discussion, it is clear that coupling is a complex concept that requires to explore a multi-dimensional space. To do so, we look at various facets to which the term can be applied. We prefer the term facets over dimensions to emphasize that not all facets are completely independent. While enumerating the following facets (summarized in Table 1), we have attempted to achieve a high degree of independence between them, but we do not claim that they are fully orthogonal.

The facets cover all relevant design aspects that help to understand which kind of coupling can be found in a system. Our method for identifying relevant facets is based on the principle that a facet should be included if there is a concrete scenario where the facet can help to better understand how a system design or a technology might be perceived as being tightly or loosely coupled.

4.1 Discovery

Discovery is one of the facets which can be approached in different ways. In closed environments, it is possible to define and implement policies for compulsory registration of service descriptions and interfaces, making it possible for clients to discover them (e.g., using UDDI registries [7]). The Web, however, has no central registry beyond the DNS (which is not Web-specific, but a core part of the Internet protocol suite). Some attempts were made to have “Web site registries” (similar to Yahoo! or the Open Directory Project) but such centralized registry-based approaches were overwhelmed by the Web’s rate of growth, and the lack of a universally accepted scheme for classifying Web resources. Nowadays, clients perform discovery on the Web through search engines. Search engines do not necessarily require the registration of new Web pages, as they can rely on crawlers following hyperlinks to discover them.

RESTful Web services are usually not described or registered in any standardized or centralized way. The idea is that a RESTful Web service can be discovered by decentralized referral (i.e., by exchanging a hyperlink pointing to it) and does not need a description in terms of its operations, and that the representations that the service accepts and produces are exposed through HTTP, and are documented as media types. Thus, discovery for a RESTful Web service means interacting with it, and the only possible “registry” might be sets of URI templates [16], describing how its resources are addressed through URIs.

In the case of SOAP and WS-**, no equivalent “loosely coupled” mechanism is available to describe relationships
Identification is done using a Uniform Resource Identifier (URI) [1]. URIs can use various identification schemes, with the most frequently used one on the Web being the http scheme. Services are free to use whatever identification scheme they like, the important aspect being that agreeing on URIs as the common way of identification makes it easier to use globally unique identifiers without relying on a centralized authority. If applications insist on using opaque local identifiers, these can be folded into URIs using the tag scheme [22]. However, this would re-create the problem of these URIs needing a proper context for interpretation that is typical for tightly coupled identification.

### 4.3 Binding

Closely related to discovery and identification, binding refers to the process of resolving symbolic names into identifiers used at a lower abstraction level. For example, a DNS name can be bound to one of the corresponding IP addresses using a DNS lookup (a form of discovery). In service-oriented systems, binding can be applied in different ways [30]. Abstract service interfaces referred to by clients need to be bound to a concrete network endpoint and transport protocol used by the service provider. Likewise, the partner links of a BPEL process need to be bound to compatible WSDL port types.

A tightly coupled binding is one that is hard to change, e.g., when it is resolved early, a long time before the result of the lookup is actually needed. Thus, compile-time or deployment-time binding are considered to establish a tight coupling between the bound entities, as it will not be possible to change the binding during the rest of the lifecycle of the system.

Dynamic binding instead happens at run-time, and particularly at the latest possible time (e.g., right before a service invocation). This is a form of loose coupling, because the binding is established only when it becomes necessary. Performing a lookup before every service invocation, however, can impose a significant overhead and establishes a tight coupling between the clients and the resolver. Not only the resolver becomes a scalability bottleneck, but if the resolver would become unavailable, all client lookups will fail and clients will be unable to perform any service invocation.

### 4.4 Platform

Platform coupling concerns the requirements for all services to be based on a homogeneous middleware infrastructure. If two services need to communicate, they must be...
built using a compatible platform and programming language. Interaction between services built using different, heterogeneous platforms (e.g., CORBA, DCOM, .NET, J2EE) is not easy to achieve, as complex and expensive bridging solutions are required [28]. A service is thus tightly coupled to a specific technology platform if this limits the possibility of outsourcing the management of the service to an external provider as well as the ability of clients built on a different platform to use the service.

Standardization helps to decouple services thanks to the interoperability it provides between different platforms. Thus, the benefit of the Web services stack of standards is to enable such interoperability and make the underlying technology platform irrelevant when it comes to connecting services. In this sense, loose coupling means that services that still need to be deployed on a specific platform (by a specific vendor) are nevertheless able to exchange messages with each other no matter which middleware platform and programming language is chosen.

4.5 Interaction

The interaction facet is about whether two services need to be available at the same time in order to successfully interact. Tight coupling is implied by synchronous interactions, which require both parties to be available at the same time in order to communicate. Loose coupling is typically associated with asynchronous interactions, where a successful interaction can happen even if one of the involved parties is not available at the same time.

In service-oriented systems, the interaction facet of loose coupling plays a major role in order to enable the communications of subsystems that are provided as a service by a different organization than the one consuming them. Thanks to the properties of asynchronous message-based communications, it becomes possible to remove the time dependencies between both ends of the communication.

For example, when it comes to performing maintenance tasks on the services, as these do not always need to be available to immediately handle client requests, the service providers do not need to schedule outages taking into account the needs of their clients. On the other hand, a service that is published using a synchronous communications protocol requires a bigger investment to avoid outages in the provider infrastructure, as client request messages will be lost if the service supposed to process them becomes (even temporarily) unavailable.

HTTP is typically perceived as a synchronous communications protocol: when a Web server is not available, Web browsers stop working. However, in more complex setups, thanks to HTTP caches, proxies and reverse proxies, it becomes possible to service HTTP requests even if the main back-end system is down. Also, the HTTP protocol can be used to interact with non-blocking RESTful Web services, which do not immediately respond to requests. Instead, together with a 202 (Accepted) status code, they provide a URI from which to download the response at a later point in time. However, as it can be seen from this example, this is only a non-blocking synchronous interaction. The client does not have to remain connected to the server while it waits for a response, but when the request is sent, the server must be available to receive it.

4.6 Interface Orientation

Interfaces to services can be designed with two alternative orientations: horizontal or vertical. Horizontal interfaces (or APIs) are local interfaces from a higher-level component to a lower-level component. APIs for remote operations still allow the higher-level component to only interact with a local component (the API), since the component providing the horizontal interface acts as a stub, hiding the distributed nature of the interaction.

Vertical interfaces (or protocols) instead define the rules that are necessary to communicate with a remote component found on the same abstraction level. A protocol still needs a communications mechanism which provides the foundation on top of which the protocol is supposed to be executed, but the protocol itself provides a vertical separation between two services.

Horizontal interfaces assume that there is a lower-level component which provides the functionality, and this component typically either sits on top of another horizontal interface (as in a layered architectural style), or directly implements a vertical interface. A typical example for this is the well-known socket API, which provides a local library for establishing inter-process communications.

Designing services based on horizontal interfaces mostly results in tight coupling, because the component providing the interface must be a more homogenous component, adhering to both the horizontal interface it implements, and the way in which the functionality is implemented, such as using a protocol to communicate with a remote service provider (Figure 2.1).2 Designing services based on vertical interfaces typically results in loose coupling, because this approach is based on the minimal set of definitions that are necessary to interact — the protocol definition.

Vertical interfaces are better for heterogeneity, because they do not make any assumption about how the local ab-

---

2 As a historical footnote it is interesting to mention that the Internet protocol stack only specifies vertical interfaces (protocols), whereas the ISO/OSI protocol architecture [46], the major Internet contender in the 80s, specified protocols as well as standardized horizontal interfaces (APIs) for all its layers.
4.7 Model

This facet is about whether the design assumes that there is a common application-level data model that is shared among services within a problem domain. If this is the case, then messages are simply treated as a serialization of that model, which is only used to transfer a model instance from one service to the other through some communications medium. This approach often uses generated code for marshaling and unmarshaling, because the model instance in essence is the shared information, and all services use the same method for mapping models to the wire format.

This shared model design introduces tight coupling, because there is a strong conceptual connection between all services sharing the same data model. This coupling often is not only tight in principle, it also is tight in practice, because services are built with a particular method or tool for marshaling and unmarshaling. Services wishing to use a different method or tool have a hard time doing so, because the representation on the wire has not been designed for interoperability and reuse in different contexts, but instead is just a serialization of the shared model.

Loose coupling does not assume that there is a shared model, instead messages being exchanged are self-contained and are designed to be processed as documents in a standardized representation format. Thus messages can be processed with any toolset for that format [43]. Loosely coupled services can use a different internal model (adapted to their local requirements) as long as they solve the problem of mapping the standard document format to their internal models. While this additional step of mapping document structures to an internal model might be regarded as overhead that is not required in the tightly coupled scenario, it is essential for loose coupling, because it allows cooperation between services as long as there is a sufficient shared understanding to define a mapping between the external canonical message format and the internal models of each service.

4.8 Granularity

Granularity of services interfaces is about the design trade-off between the number of interactions that are required to provide certain functionality to service a large client community, and the complexity of the data parameters (or operation signatures) to be exchanged within each interaction. In API design, a typical goal is to minimize the number of interactions [17]. In service design, this is even more so, due to the high latency involved in a service invocation.

By using more coarse-grained interfaces (i.e., fewer interactions required), services can exploit the extensibility of well-designed message formats, and service evolution (Section 4.10) can be based on message extensibility, rather than an extension of the set of possible service interactions.

4.9 State

Depending on the scale of a service landscape, state management can become one of the central problems in efficient service design. Stateful services are based on the assumption that a service keeps state of an ongoing interaction, leading to problems for services with many clients, high throughput, and long-running transactions. The alternative is to choose a stateless service design, which keeps state in the messages that are passed back and forth between cooperating services.

Loose coupling for state management means stateless services. As an example, the acronym REST itself stands for Representational State Transfer and highlights that stateless interactions are one of the key properties of the Web. Mechanisms for session management (e.g., cookies [24] or URI rewriting) are nevertheless also available. However, shared state (i.e., state that is kept by two or more interacting services) always implies tight coupling, because there must be associated mechanisms of establishing and recovering stateful sessions, comparing states, resolving inconsistencies, implementing time-out mechanisms, and performing distributed garbage collection. The management overhead of shared state is substantial, so it should only be employed when required, i.e., to reduce the size of message payloads.

4.10 Evolution

This facet concerns how services can evolve over time, and how that affects their clients. From the point of view of the service provider, compatibility among versions can be seen in two directions: Backward compatibility allows older clients to keep functioning when using a service that has been upgraded to a new version. Forward compatibility allows newer clients to use an old version of a service, even though they have been developed against a newer version of the service. Evolution becomes particularly important in connection with the discovery facet, because the compatibility of clients and services should be taken into account during the lookup. Also, in case of late binding, run-time errors may be produced if clients are bound to evolved (and potentially incompatible) services.

Tight coupling for this facet is implied by an exact match of versions, so that neither forward or backward compatibility are supported. A loosely coupled design instead attempts to provide as much forward and backward compatibility as possible, even if there are limits to how many changes an interface can withstand without breaking clients. Loose coupling therefore needs to combine rules about how to handle differing version numbers, and what rules to apply in the event of differing version numbers.3

3Interestingly, one of the core specifications of the Web, XML itself, at the time of writing is undergoing a controversial update. The proposed 5th edition of XML 1.0 [4] would not change the version number of XML, but introduce new features. This way, the loose coupling of XML-based services would be compromised, because some documents conforming to the 5th edition could not be processed using older XML processors. It is still unclear whether the proposed
Common design patterns for this facet are mustUnderstand and mustIgnore, so that the service has a mechanism to communicate to its clients how to deal with new protocol constructs. However, mustUnderstand also introduces the risk of fragmentation, because older clients cannot use services whose new features are flagged with mustUnderstand — this mechanism therefore has to be used with care. mustIgnore, on the other hand, helps to provide a forward compatible evolution path, as it refers to the convention of ignoring unknown protocol elements (e.g., Web browsers do not break if they encounter unknown HTML tags, which are simply ignored).

Another important design aspect related to service evolution are well-defined rules about extensibility. Extension points of a service interface must be well thought out and should be clearly marked, so that clients know where to expect changes and extensions. The patterns mentioned above can then be used to learn what to do when such extensions are encountered.

4.11 Generated Code

For sufficiently precise interface descriptions of models (Section 4.7), it is possible to automatically generate code for handling key communications functions. Code generation only works if the communication requirements are completely specified in machine-readable form, and if evolution (see Section 4.10) of the system happens in well-defined, predictable ways.

Code generation takes a service description and turns it into code representing the service at the time the code was generated. This produces tight coupling between the generated code and the description. If the description changes, the code of the automatically generated stubs may no longer work. Moreover, code generation introduces a dependency on the corresponding run-time environment. Also if this changes, the code may have to be regenerated.

A loosely coupled design does not use static code generation. Instead, it uses declarative mechanisms to model core concepts of the system design (e.g., content types on the Web), methods to communicate using these concepts (i.e., content negotiation), and then leaves it to the participating systems to either hardcode assumptions about the system environment, or to provide dynamic extension mechanisms.

For example, most Web browsers feature an extension mechanism for MIME content types, because media types appear at a faster rate than Web browsers are updated. On the other hand, most Web browsers do not have a well-designed extension mechanism for URI schemes, because new schemes do not appear very often. So with respect to the code generation facet, browsers provide a good foundation for loose coupling on the media type level, but much less so on the URI scheme level.

4.12 Conversation

Most service invocations span multiple basic interactions, forming conversations. Services may provide functionality that requires clients interacting with them to follow a certain path, which prescribes to follow a set of partially ordered message exchanges.

A tightly coupled design aims at augmenting the service description with metadata constraining all possible interaction sequences to the correct ones. Whereas this enables to statically check that a client correctly interacts with a service, it also restricts its future evolution, because the client makes many assumptions on how to interact with the service. Specification languages for orchestration such as the Business Process Execution Language (BPEL) or choreography such as the Web Services Choreography Description Language (WS-CDL) can be used to augment service interface contracts with a static description of possible correct conversational interactions.

Enabling clients to discover at runtime how to correctly interact with a service is a loosely coupled design practice for enforcing guarantees about the conversation. This can be achieved with a reflective inspection mechanism, enabling clients during the invocation of a service, to inquire with the service itself about what are the possible future steps of the interaction, and to dynamically pick among them the next interaction.

This approach is followed by RESTful Web services, which use hyperlinks as the mechanism to steer clients participating in loosely coupled conversations. In a very simple scenario, a RESTful multistage process for filling out forms on the Web controls the conversation by replying to each form submission with a new form and a URI where to submit that new form. Clients can correctly follow this long interaction without any prior knowledge of the corresponding process. Also, no description of how the process is designed must be provided in advance by the service. By simply inspecting the representations of the service’s resources, clients are able to follow the required conversation pattern of the service. Hyperlinks thus create the ties which hold together the various resource URIs which are accessed while interacting with the service. Since these ties are established at run-time, no tight coupling is established between services.

5. Evaluation

In this section we aim at moving from a qualitative description of these facets to a quantitative evaluation, addressing the problem of how to use the twelve facets to analyze the properties of three representative Web services technologies: RESTful HTTP, RPC over HTTP, and WS-* based messaging on a so-called enterprise service bus (ESB [6]).

A similar analysis can be conducted over the design of concrete service-oriented systems.

Table 2 and Figure 3 summarize our findings, listing for each technology the corresponding coupling characteristics and visualizing the implied degree of coupling. For all technologies we immediately see that all are operating system and programming language independent, making them loosely coupled according to the Platform facet. However this was not necessarily true for traditional vendor-specific middleware platforms, thanks to the WS-* standardization efforts and the pervasive support for the HTTP protocol, nowadays loose coupling in terms of platform independence can be safely assumed for all Web services technologies.

In the first two columns we compare a RESTful usage of the HTTP protocol against its use for implementing Remote Procedure Calls (RPC) across the Web. This is an important comparison, as it helps to highlight the differences between true RESTful Web service APIs, and RPC-based Web services provided using, for example, SOAP over HTTP, or XML-RPC. Concerning the discovery (Decentralized), identification (Global), and binding (Dynamic) facets, we do not observe any difference due to the properties of the HTTP
However, when it comes to the interaction facet, some differences become apparent. RPC interactions are by definition synchronous. RESTful interactions are instead asynchronous, since services interact indirectly by updating (with POST, PUT, or DELETE) the state of resources, which can be later accessed by other services (with GET) [33]. Interface orientation is vertical in REST, which only relies on the protocol and resource representations, whereas RPC is often based on the stub mechanism where the client of a remote service accessed via RPC calls a local interface which then handles the fact that the service is implemented remotely. The two technologies also differ in terms of the model facet, where RPC follows a shared model approach, where all services must agree beforehand on the syntax and the semantics of the exchanged messages, while REST promotes a “Self-Describing Representations” solution. RESTful interactions are also stateless, while RPC offers both options, as interacting services may establish a session by sharing state among them, but also may be designed to avoid such tight coupling. RPC is also based on generated code stubs, while REST does not require them as it follows a more dynamic approach based on plugin extensibility. Also regarding conversations, REST promotes a dynamic, reflective approach, while RPC-based Web services make the interaction constraints explicit at design-time.

It is worth noting that not all the facets are bound by the properties of a given technology. For example, the granularity, state, and evolution facets depend on the concrete design choice of a given service-oriented system architecture, and are not constrained by the choice of the REST vs. RPC styles. In other words, it is possible to design tightly coupled RESTful Web services, which can be “chatty” in their interactions, if their interfaces expose a large number of fine-grained resources. The same can be said about RPC-based services, which can either publish many fine-grained operations, or a few coarse-grained ones, depending on the chosen design strategy. Also in terms of the evolution facet, a service design needs to be evaluated at a more specific level. For example, XML technology can support a loosely coupled evolution facet only if it is used with additional guidelines for versioning and the enforcement of mustUnderstand rules.

We have chosen to include in our evaluation the WS-*/ESB technology (which is not Web/HTTP centric), because the enterprise service bus family of middleware products is widely perceived to be the foundation for loosely coupled SOA implementations. Thus, it is interesting to apply our multi-

### Table 2: Web Services Technology Evaluation Summary

<table>
<thead>
<tr>
<th>Facet</th>
<th>RESTful HTTP</th>
<th>RPC over HTTP</th>
<th>WS-* / ESB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Discovery</strong></td>
<td>Referral</td>
<td>Referral</td>
<td>Registration</td>
</tr>
<tr>
<td><strong>Identification</strong></td>
<td>Global</td>
<td>Global</td>
<td>Context-based</td>
</tr>
<tr>
<td><strong>Binding</strong></td>
<td>Late</td>
<td>Early/Late</td>
<td>Late</td>
</tr>
<tr>
<td><strong>Platform</strong></td>
<td>Independent</td>
<td>Independent</td>
<td>Independent</td>
</tr>
<tr>
<td><strong>Interaction</strong></td>
<td>Asynchronous</td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td><strong>Interface Orientation</strong></td>
<td>Vertical</td>
<td>Horizontal</td>
<td>Horizontal</td>
</tr>
<tr>
<td><strong>Model</strong></td>
<td>Self-Describing Messages</td>
<td>Shared Model</td>
<td>Self-D. Messages/Shared Model</td>
</tr>
<tr>
<td><strong>Granularity</strong></td>
<td>Fine/Coarse</td>
<td>Fine/Coarse</td>
<td>Fine/Coarse</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>Stateless</td>
<td>Stateless/Shared, Stateful</td>
<td>Stateless</td>
</tr>
<tr>
<td><strong>Evolution</strong></td>
<td>Compatible/Breaking</td>
<td>Compatible/Breaking</td>
<td>Compatible/Breaking</td>
</tr>
<tr>
<td><strong>Generated Code</strong></td>
<td>None/Dynamic</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td><strong>Conversation</strong></td>
<td>Reflective</td>
<td>Explicit</td>
<td>Explicit</td>
</tr>
</tbody>
</table>
faceted definition also to measure the coupling implied by this technology. We observe that ESB provides loose coupling according to four facets: binding (Dynamic), interaction (Asynchronous), state (Stateless) and platform (Independent). However, the technology uses context-based identification (a tightly coupled solution) and requires centralized service registration to support service discovery. Also, interactions based on multiple message exchanges are explicitly modeled using workflow and conversation models. The development process of services connected by an ESB relies on code generation techniques (thus, an ESB presents an horizontal interface orientation). Therefore, according to all of these other five facets, this kind of middleware technology does not help to achieve loose coupling. Similar to the other two alternatives, the choice of using an ESB does not constrain the evolution and granularity facets. Additionally, it is also possible to choose between a shared model design, or to leverage the ESB mediation capabilities to foster a more loosely coupled design based on self-describing messages.

In more quantitative terms, the table counts the number of facets for which a technology results in loose or tight coupling. We also distinguish facets for which the degree of coupling does not depend on the chosen technology but may vary depending on more specific design decisions.

<table>
<thead>
<tr>
<th>Coupling</th>
<th>REST</th>
<th>RPC</th>
<th>WS-*/ESB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loose</td>
<td>10</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Tight</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Design-Specific</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Only in the best case (assuming that all design-specific facets follow loosely coupled options) we can conclude that using RESTful HTTP would provide a system architecture featuring loose coupling according to all facets. Choosing RPC over HTTP, instead, would not result in a completely loosely coupled system, due to the 4 facets (interaction, model, generated code, and conversation) which only present a tightly coupled approach. This alternative also requires more effort to achieve a loosely coupled system, as up to 5 facets are design-specific and are unconstrained by the technology choice. This number is smaller in case of the WS-*/ESB alternative, where only 3 facets are design-specific. A more detailed, facet-by-facet comparison is visualized with the radar chart of Figure 4. It is interesting to notice that the curve indicating the degree of coupling implied by RESTful HTTP is strictly bounded by both of the other curves. If we do not consider the discovery and identification facets, the same holds between the RPC and the ESB curves. Thus, our multi-faceted metric can be used to establish a partial ordering between different Web services technologies in terms of their degree of coupling.

6. CONCLUSIONS

SOAP-based Web services and the REST architectural style have been and still are the topic of many debates. Many of these debates are heated, often missing the point that the more prescriptive style of the SOAP approach and the more descriptive style of the REST approach have their roots in different scenarios, the former assuming closed worlds and contractual relationships, whereas the latter caters to an open world with ad-hoc interactions [40]. So far, only few attempts have been made to compare both approaches as objectively as possible [31]. “Loose coupling” and “tight coupling” are frequently used terms in such debates, given the positive connotation of the former and the negative implications of the latter. Reduced coupling is beneficial because interdependencies typically make complex IT application systems brittle and slow to adapt to changes [32].

In terms of the goals which should be accomplished when designing service systems, WS-* and REST can be described by integration vs. cooperation (Fiedler et al. [11] make a similar distinction for database systems). Both goals (as well as “loose” and “tight” coupling) are not good or bad per-se. They are the result of a strategic decision on how to design and implement IT architectures, and there can be valid business objectives for both of these goals. These business objectives should be the input for a decision how to design a system, for example putting a higher emphasis of performance optimization (usually easier with tight coupling) or agility (usually easier with loose coupling).

The twelve facets described in this paper make it easier to understand which approach is more appropriate for a given problem, and for which facet of the system design a loose or tight coupling approach should be preferred. In the end, as we have shown in our evaluation, very few systems are loosely or tightly coupled according to all facets. Instead, they use a mix of both depending on the business objectives and the constraints of the chosen Web technologies. Our multi-faceted metric thus also defines a set of choices that need to be made, giving system designers a more structured approach for making better design decisions and comparing alternative Web services technology options.

Acknowledgements

The authors would like to thank Domenico Bianculli for his constructive feedback. This work is partially supported by the EU-IST-FP7-215605 (RESEvoir) project.

7. REFERENCES
