Practical Challenges with Web Service Evolution: A Review of Literature and Current Industry Practices

S M Sohan and Frank Maurer

University of Calgary
{smsohan,frank.maurer}@ucalgary.ca
http://ucalgary.ca

Abstract. In this paper we identify the key challenges with web service evolution such as backward and forward compatibility, serving and testing multiple concurrent versions, version specific documentation, communicating future changes, etc. Following an appropriate strategy to overcome these challenges will enable web service publishers to iterate their services as required while minimizing the impact on existing consumers of the services. In order to find an appropriate strategy, the web service publisher needs to balance a tradeoff between the business drivers and technical challenges. To understand and deal with these challenges, we performed a systematic review of the existing literature on web service evolution and investigated some real world web services. Based on our findings, we discuss the challenges and suggested pragmatic techniques to address the challenges.

Keywords: Web Service, Evolution, Versioning, Adapt, SOA, SOAP, REST

1 Introduction

Web services offer a web-based communication mechanism between two software systems, commonly known as web service publishers and consumers. Wikipedia describes web service as follows:

> A web service is a method of communication between two electronic devices over the World Wide Web. A web service is a software function provided at a network address over the web or the cloud, it is a service that is "always on" as in the concept of utility computing.

$$http\,en.wikipedia.org/wiki/Web_service$$

For example, geographical mapping web services are consumed by many different kinds of consumers in real life such as: real-estate listing web sites, local business search portals, transit information web sites, etc. Similarly, payment processing web services are used by a multitude of consumers such as: e-commerce applications, retail stations etc.

Photo (producer, consumer)

Web services provide web-addressable endpoints for the communication between the two parties, where the consumer initiates the requests and the publisher responds to the requests. To carry out the actual communication, the web service publishers and consumers follow specific protocols. Of the many protocols that are in use today

$$http://en.wikipedia.org/wiki/List_of_web_service_protocols$$

, two commonly used protocols are as follows:

- Simple Object Access Protocol or SOAP web services
- Representational State Transfer or RESTful web services

1.1 SOAP Web Services

SOAP was designed in 1998 by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein for Microsoft and is currently maintained by the World Wide Web Consortium. [http://en.wikipedia.org/wiki/SOAP] In brief, SOAP is an XML based protocol to communicate structured data and necessary headers between the consumers and publisher of a web service.

A SOAP web service is composed of one or more operations or methods. SOAP XML messages are used for communication between a consumer of the web service and its methods. The structure of these methods, their corresponding messages and other meta information of a SOAP service can be described using an accompanying XML based Web Services Description Language or WSDL file. In addition to providing a structured definition of the service, these WSDL files can be automatically parsed by code-generators to provide the consumers with an easy integrate interface by abstracting out the details of the protocol level implementation.

1.2 RESTful Web Services

Being an XML based protocol, SOAP messages are often verbose. Also, the additional header information introduces communication overheads. On the other hand, RESTful web services provide a lightweight and flexible approach, where the HTTP protocol stack is leveraged without imposing constraints on the actual format of the messaging.

The term Representational State Transfer (REST) was introduced by Roy Fielding

$$https://en.wikipedia.org/wiki/Representational_state_transfer\#cite_note-Fielding-Ch5-1$$

and is used interchangeably as RESTful web service or Web API. In brief, REST is an implementation for web services where user defined “resources” are created, accessed, modified and deleted utilizing the various features of the HTTP stack

such as caching, security, layering and different request methods (GET, POST, PUT, DELETE etc.), content types, etc.

Due to their historical usage, the term "web service" is often interpreted as SOAP. But throughout this paper, unless otherwise specified, we have used this term to represent a web based communication method between two software systems irrespective of their actual implementation details. However, even though there are different kinds of web services

$$http://en.wikipedia.org/wiki/Web_service$$

, the findings mentioned in this paper are based on SOAP and/or REST based services only. This is due to their relative popularity over the other types.

XXXXX WHAT FOLLOWS XXXXX

2 Motivation

Web services, like other software systems, need to evolve to stay on top of the bugs and ever changing requirements. For example, between May 15th 2013 and May 22nd 2013, the Facebook API team saw 238 new bugs, resolved 211 bugs and fixed 24 bugs.

$https://developers.facebook.com/blog/post/2013/05/22/platform-updates--operation-developer-lo$

However, often times web services are consumed by an unknown number of consumers, and the web service publishers have little control over them. Updating a web service can be tricky in such a situation since a breaking change may upset the existing consumers. To be able to evolve in such a decoupled scenario, multiple versions of a service need to co-exist which poses both technical and business challenges in multiple fronts. At a conceptual level the following diagram shows the different aspects of a web service that need to deal with evolution challenges:

Web service publisher (QoS, SLA) web service consumer Documentation Source code Testing Web service endpoint/address Web service protocol Service interface and source code Data store

However, there is a little to no standard approach to tackle these aspects of an evolving web service. The motivation for this research stems from the need to improve this situation.

3 Research Questions

To better understand the challenges related to web service evolution we have focused on the following list of research questions:

- Why do web services need to evolve?
- How to evolve the web services?
- What are the key challenges for evolving web services?

Focusing on these questions will help us identify some of the gaps of the currently available techniques for evolving web services. Using this as a background, innovative future work can be carried out to fill these gaps and simplify web service evolution.

4 Research Method

To address the aforementioned research questions, we have limited our search to two high level sources as follows:

- Published articles from the academia and the industry
- Real world web services

We have performed a Systematic Literature Review to extract information from the published articles following

$$Guidelines for performing Systematic Literature Reviews in Software Engineering$$

. We have also investigated the published real-word web services to find out the answers to the research questions based on todays solutions.

4.1 Systematic Literature Review

Kitchenham et al defines systematic review as follows [Guidelines for performing Systematic Literature Reviews in Software Engineering]:

> A form of secondary study that uses a well-defined methodology to identify, analyse and interpret all available evidence related to a specific research question in a way that is unbiased and (to a degree) repeatable.

We have carried out a systematic review because it helped us examining and summarizing the relevant work from the existing literature. To this regard, we have identified relevant papers from the following sources: IEEExplore, ACM Digital library:, Google scholar, Citeseer library, ScienceDirect, SpringerLink, and Google.

To identify the relevant papers from these sources we have performed keyword searches. As mentioned in

$$Systematic Mapping Studies in Software Engineering$$

, keywording is a two step process, 1) extracting keywords from reading the abstracts and, 2) combining keywords from different papers. At step 1, we have initially searched for the papers containing the terms “web service” and extracted some commonly used synonyms from their abstracts. We have seen repeated mentions of the terms Service Oriented Architecture or SOA, SOAP, Web API and REST to denote similar concepts. In addition to this, we have expanded

the word "change" with its commonly used English synonyms. At step 2, we combined the keywords and formed the following case insensitive search query:

(web service OR SOA OR SOC OR SOAP OR REST OR Web API) AND (change OR evolve OR evolution OR version OR update OR Adapt OR versioning)

We performed searches using this query on the full contents of the articles, ignoring their publication date since the topic of web service is a relatively new one. From the search results, we performed a primary screening after reading the abstracts. Then, for the remaining articles, we continued reading until we could determine if it was relevant or not. If it was found relevant, we read the full contents and extracted information to answer our research questions. For the scope of this research, we defined an article as relevant only if it contained topics about the evolution of web services. After the screening process, we had a shortlist of articles to extract information from. In the next paragraphs, we discuss the findings from these sources in light of the research questions.

5 Research Findings

5.1 Literature Review

Why Do Web Services Evolve? In this section, we have discussed the drivers behind web service evolution from the related literature. To begin with, in

$$On Analyzing Evolutionary Changes of Web Services$$

Treiber et al provided a list of changes related to web services with their interdependencies. To identify the source and impact of a change, they categorized the stakeholders into four roles:

- web service provider, who is responsible for planning and conception of the service
- developer, who implements the planned service
- service integrator, who integrates with external web services and
- user, who actually uses the integrated application.

One or more of these roles may be played by one individual. The stakeholders take part in driving changes to a web service. The authors identified the following types of changes in a web services: Interface, Implementation, QoS, Usage, Requirement, SLA, Pre and post conditions, and Feedback.

A change in any of the above impacts one or more of the stakeholder roles based on its type. In addition to stakeholder impacts, the sources are also interdependent on each other. For example, when a web service provider decides to change the interface, the developer needs to write the changed interface and its implementation. If the changed interface if not backward compatible, and the service integrator needs to modify the integration. The new interface and implementation may instigate a change on its QoS, Usage and Pre-post conditions, which may impact the end user.

To summarize, in this paper we see the drivers behind web service evolution, a list of parameters that can be involved in a change, and their interdepency that must be considered for evolving web services.

Borovskiy et al identified some drivers behind web service evolution from the perspective of an enterprise, where both the web service consumer and publisher may be part of the same organization

$$Evolution Management of Enterprise Web Services$$

. In this regard, they categorized two classes of drivers behind the evolution of web services as follows:

- Intrinsic change drivers: Poor design, Poor implementation quality
- Extrinsic change drivers: Market drivers, Differentiation drivers, Business requirement drivers, Operational process drivers, Legislative regulatory drivers, User related drives

To summarize, although the authors discussed these change drivers for enterprise web service context, we found these drivers to be relevant in a broader context and applies to non-enterprise web services.

How to evolve the web services? Now that we have discussed the drivers behind web service evolution in the previous section, in this section we have presented the different approaches to tackle web service evolution based on the existing literature.

Tradeoffs: Lublinsky compared between multiple approaches of implementing evolving web services based on different dimensions as follows

$$Versioning in SOA$$

.

- Units of versioning: One approach to version a service is to apply a single version to all its methods that can be deployed as a group. Another approach is to version individual methods or operations of the service as they change, this confines the change to only specific methods and provides a finer grained versioning. However, deployment of such versioned methods are complicated and it forces the consumer to specify the service, its method name and an additional version to use for each operation.
- Service changes, constituting a new version: In line with [On Analyzing Evolutionary Changes of Web Services] Lublinsky also pointed out that a new version may be required primarily because of changes in its interface, schema of the message payload or in its implementation.
- Service version life-cycle considerations: Based on the actual need, Lublinsky suggests adjusting the life-cycle of versions, as too many versions increase the maintenance overhead while too few versions leaves the consumers a smaller time window to upgrade.

– Version deployment/access approaches: When deploying versioned services, Lublinsky compares between multiple versions deployed on the same endpoint vs. each version deployed on its own endpoint. Lublinsky points out that the former approach may introduce conflicts in names of the objects, databases and other related resources, and requires intermediate routers, which typically lowers performance and may introduce SLA complications. On the other hand, as mentioned in the paper, the latter approach provides better scalability and loose coupling between versions, but requires the consumers to use a different endpoint to target a different version.

To summarize, this paper paper points out some key trade-offs that need to be addressed while evolving web services based on the specific use case.

When multiple versions of a service are deployed at the same time, a versioning scheme is required to identify and use a specific version. Leskey discusses about the versioning of web services in

$$Considerations for Versioning SOA Resources$$

with respect to the OASIS SOA Reference Model (SOA-RM). In this paper, the author discussed versions in terms of identifiable resources, that can have multiple versions and are modified by applying new versions to the original one.

Versioning: To identify a specific version of a resource, Leskey suggests using a unique identifier with an accompanying explanation of how to interpret it. The versioning identifiers should be used consistently and be readily accessible and understandable to its consumers. In addition to this, the author also recommends specifying a policy describing compatibility between the versions so the consumers clearly understand the impacts of version changes. Because a web service is a contract between the publisher and its consumers, the policies from both sides need to be considered for its evolution. As an example, the author mentioned a generic policy as follows:

> A versioning scheme for a service may include a generic policy, such as any succeeding version identified as 1.j will be backward compatible with any 1.i previous version in the sense that results are identically generated in version 1.j for functionality that existed in previous 1.i versions.

The author then identified the key reasons for changes between versions such as: a) service description changes, b) changes in functions, c) changes in the interaction mechanics, and d) change of known pre and post conditions. These changes would require new versions of the service as well as accompanying any explanation of the specific changes.

To locate each version, the author provided the following example URL based scheme: http:///a.b.c/services1/20080601/ , where service1 is the name of the resource and 20080601 is the date based version identifier. Given this URL scheme, the author suggests the service descriptions to dereference the URL and provide necessary details about the version so that the versioning information is transparent to the consumers.

Implementation: Once an appropriate evolution and versioning strategy is selected, the required changes need to be implemented. In this section we have discussed the implementation approaches as suggested in the literature from the perspective of two levels, 1) protocol level and 2) application level.

Protocol Level Implementation: Since web services are typically based on protocols such as SOAP, REST, etc., protocol level support for evolving web services would allow for uniform standards. In

$$Semantically Extensible Schemas for Web Service Evolution$$

Wilde proposed a conceptual framework for SOAP web services where consumers can be forward compatible in a managed way.

At its core, the framework includes open and extensible XML schemas so the consumers of old version can work with the schema in a newer version. In addition to this, the framework recommends using a set of custom extensions to the XML schema. These extensions can be as simple as XML tags such as mustUnderstand, mayIgnore or some complex ones to suit the specific use case in hand. But irrespective of their actual usage, an accompanying guideline on their processing rules need to be provided. Both the consumer and publisher of the website need to follow the processing rules for these custom tags. Using the approach of this framework, the consumers of the web service can attain a predictable level of "graceful degradation" as the service evolves.

Using a similar approach but instead of arbitrary custom tags to extend the schema, Fang et al provides a concrete list of six XML nodes to create a version-aware service model for SOAP web services [A Version-aware Approach for Web Service Directory]. They introduced extensions to WSDL and UDDI to tackle evolving web services. These extensions are composed of six XML nodes: 1) version name, 2) version description, 3) startTime of the version, 4) endTime of the version, 5) alias such as (new, current etc) of the version name and, 6) original version that preceded it.

With the help of this versioning description of the service, a consumer can find and register for a target version through a service registry. The server can publish events as new a version of a service is deployed and the registered consumers can subscribe to these events. When a new version is released, the consumer gets notified about the change and can self update if the changes are backward compatible. The authors discussed an example implementation of this approach and demonstrated that using this approach SOAP services can be versioned and future changes can be communicated to the consumers.

In

$$End - to - End Versioning Support for Web Services$$

Leitner et al provided an alternative approach to transparently handle multiple versions of SOAP web services using their framework called VRESCo. At its core, VRESCo builds a graph of the versions showing predecessor relations as follows:

XXXXX VRESCO IMAGE XXXXX

For each version, VRESCo uses a free-form tag to identify it. The version tag and associated meta information from the WSDL file of each version is stored in a database. So, when new versions are released, it can perform an analysis to see if the changes are backward compatible. To route a consumer request to the right version, the versions of a service are deployed behind a proxy. Based on a "selection strategy" specified by the consumer (e.g. always use latest, always use stable, fix version, etc.) the proxy can then automatically select the desired version without needing the consumer to change its implementation. For compatible updates, this approach off-loads the consumer at the expense of an additional proxy layer.

In addition to the protocol level evolution support for SOAP web services, Mangler et al showed a solution for RESTful web services in

$$On the Origin of Services - Using RIDDL for Description, Evolution and Composition of RESTful Service$$

. They introduced a description language called RIDDL for RESTful web services to support service composition and evolution.

RIDDL adds XML based descriptions to RESTful web services. Using RIDDL, service evolution is expressed through a chain of adapters, where the output of a service matches with the input of its preceding version. From this XML declaration of the chain, it is possible to merge the inputs and outputs from each adapter to create a full description of a specific version.

Application Level Implementation: Following from the protocol level implementation approaches discussed in the previous section, we have presented the application code level solution approaches in this section.

Kaminski et al presented a design technique for evolving web services from the perspective of the source code implementation

$$A Design Technique for Evolving Web Services$$

. They identified some key requirements for their design such as: a) backward compatibility, b) common data store, c) no code duplication, d) untangled versions, e) unconstrained evolution, and f) a visible mechanism. To summarize, the design aimed to allow for web service evolution while staying compatible with the existing clients without introducing duplication in the code or in the data store in a transparent manner.

With these requirements laid out, they proposed a design technique called "Chain of Adapters" for the implementation of evolving web services. The following figure shows a high level view of the design:

XXXX Chain of Adapters XXXXX

$$Source : A Design Technique for Evolving Web Services$$

In essence, the design technique can be described as follows: To start with, a web service can be developed as usual. Then, the interface of the web service is

duplicated under a namespace V1. An implementation of this interface is created in the same V1 namespace that performs necessary translations on the data and delegates the methods to the original web service. This produces a snapshot of the V1 service which can be published for its consumers.

Now, to evolve to V2 of the service, the changes are made to the original web service. For any breaking change, it is compensated in a V1¡–¿V2 adapter. This V1 ¡–¿ V2 adapter is the first of the "Chain of Adapters" to be used for evolution. Once the changes are complete, the interface of the original web service is duplicated into a the V2 namespace. In the same namespace, an implementation of this V2 interface delegates the calls to the original web service, while the V1¡–¿V2 adapter targets this new implementation. This way, a new snapshot for the V2 of the service is created.

Since the adapter compensates for all breaking changes from V1 to V2, existing V1 clients still continue to work. As the service evolves, introducing new adapters for each version would produce a chain of such adapters to ensure backward compatibility and code reuse while being able to serve multiple concurrent versions.

The "Chain of Adapters" technique addresses their requirements, but they mentioned some caveats. For example, the adapters need to ensure all breaking changes are compensated, which my need for a separate data store when fields are removed from the datastore on subsequent versions. Also, this chain effectively connects all the versions, so a bug or its fix on one version may have impact on all the versions.

As a proof of concept, the authors also presented a prototype implementation following their technique and discussed its performance characteristics. Overall, we have found the "Chain of Adapters" technique to be a simple solution to the source code level implementation of evolving web services.

Communication: A web service involves multiple stakeholders that may be impacted by a change. So, it is important to communicate the changes and their impacts to the stakeholders as the service evolves. In this section, we have discussed the suggested ways to effectively communicate about the changes from the existing literature.

In

$$On Synchronizing with Web Service Evolution$$

Zou et al presents a technique to generate consumer specific changelogs for evolving SOAP web services. When a new version of a web service is published, it typically accompanies a custom formatted, often HTML based, documentation of its methods and parameters. For existing consumers, this may require a significant effort to identify the impacts of the new version specific to their usage by looking at the potentially large documentation. To address this problem, the authors suggested creating customized release notes for each consumer based on their usage pattern in the following steps:

- Service invocation monitoring: a monitoring system is needed to track the usage pattern of each consumer so when a service is changed, it can identify specific areas of interest for a consumer
- Service difference analysis: Since SOAP services are described using XML based WSDL files, the differences between the two versions such as addition, removal or modification can be analyzed by comparing the WSDL files
- Release note customization: The release note can then be tailored for each consumer by using the data collected at step 1 to only include relevant changes per customer
- Code release note linkage: Using the customized release note from step 3, it is possible automatically inspect the consumer code and discover fragments of the code that need to change for the new version.

Aversano et al provided an alternate approach to communicate changes in SOAP web services using a graphical way in

$$VisualizingtheEvolutionofWebServicesusingFormalConceptAnalysis$$

. To construct the graphical representation, the authors extracted meta information about methods and parameters for SOAP web services from the WSDL files. This meta information is then presented in a graph to visualize the service with its methods and parameters. The graph can be used to visually communicate the similarities and differences among the subsequent versions of the same service.

5.2 Review Of Industry Examples

To answer the research questions, in addition to looking at the published articles, we have also reviewed the following three popular web services from the industry:

- Google Maps
- Twitter API
- Facebook Graph API

A comprehensive review of all the real world web services is beyond the scope of this research. However, these web services are evolving with a large number of existing consumers. The different evolution approaches as followed by these popular web services are discussed in the remainder of this section.

Google Maps: Google Maps provides a JavaScript based web service for its consumers and updates the service to provide "new features, bug fixes and performance improvements"

$$https://developers.google.com/maps/documentation/javascript/basics\#Versioning$$

. Their versioning policy offers the following versions:

- Experimental version: This version has all the latest features, bug fixes and improvements. But this version is not guaranteed to be feature stable. Consumers can remain on the edge of the service by using this version.
- Release version: From the the experimental version, every quarter, a feature stable version of the service is cut and tagged as a "release" version. These release versions may receive bug fixes but the features remain stable.
- Previously released or Frozen version: Every time a new "release version" is tagged, the oldest existing release version is "frozen", so no new features or bug fixes are applied to ensure it remains completely feature stable.

The consumers can specify the desired version using a version parameter in the URL. For example, to use the version 3.12, the following URL can be used https://maps.googleapis.com/maps/api/js?v=3.12 In absence of a version parameter, by default the experimental version is used. Every time a new release version is tagged, the oldest version is retired. Google Maps guarantees the versions are backward compatible and automatically migrates the consumers using the oldest version to the following version without needing the consumers to make any change on their ends.

The Google Maps API has HTML based documentations and changelogs for each version. In addition to this, the consumers can subscribe to an email group to keep up on the new version announcements.

Twitter API: Twitter provides a RESTful web service to interact with its data from third-party applications

$$https://dev.twitter.com/docs/api$$

. This web service is also evolving to fix bugs and bring new features, but using a different approach from that of Google Maps.

At the time of writing this paper, they have versions 1 and 1.1 deployed for the consumers. Version 1 is marked as deprecated and announced to be retired in six months after version 1.1 was released

$$https://dev.twitter.com/docs/api/1.1/overview$$

. The consumers can specify the target version in the URL. For example, the web service calls to https://api.twitter.com/1.1/ will use version 1.1.

Once a version is released, future changes are made to the released version to provide new features and bug fixes. However, unlike Google Maps, new features are not always released under a new version name. Instead, the changes may be deployed under the same version name and are communicated to the consumers using an updated Calendar of API Changes

$$https://dev.twitter.com/calendar$$

through a developer portal.

The developer portal is also used to record issues and collect feedback from the consumers. The portal contains version specific HTML based documentations of the web service as well as an interactive web browser based console to explore the service on the browser

$$https://dev.twitter.com/console$$

.

Facebook Graph API: Unlike the previous approaches, Facebook Graph API on the other hand uses a different approach for its evolution. Facebook Graph API is the primary web service to retrieve and post data to Facebook

$$https://developers.facebook.com/docs/reference/api/$$

. They have a 90-day breaking change policy stated as follows:

> Platform changes that would require a code change from developers (security and privacy changes excluded) will be announced at least 90 days before the change goes into effect.

They also identified some example of the change drivers such as, change/removal of major functionality, backwards compatibility, Facebook product changes, and privacy and security related changes. For each new version, instead of a URL based addressing scheme, Facebook allows the consumers to enable the changes by pushing a button on the Facebook website. Using the button, the service consumers, aka platform developers, can opt-in to migrate to a new release to get an early look on the impact of the breaking changes specific to their use case and take necessary actions within the 90-day period.

To communicate the web service changes with its consumers and collect feedback, Facebook uses a developer portal as well as a StackOverflow channel

$$http://facebook.stackoverflow.com/$$

. The developer portal shares information about planned upcoming changes as well as the release notes of each version.

The Facebook Graph API accompanies both a static HTML based and an interactive documentation of the API called "Graph API Explorer". As the name suggests, the interactive API explorer lets a consumer to explore the features of the service using live data on the browser without needing to write any code.

6 Discussion

Based on our findings as discussed in the preceding sections, in this section we have summarized the key challenges and identified avenues for future work related to evolving web services. The challenges are as follows:

6.1 Deployment Challenges

Multiple versions of an evolving web service need to be deployed at the same time to support the consumers of each version. This poses the following challenges:

- Should multiple versions be deployed under a separate or a single endpoint?
- Should a version apply to a whole service or to the individual methods?
- How many versions need to be deployed at any given time?
- How and when to migrate the consumers to a newer version?
- How and when to migrate the consumers that are using a retiring version?
- How to identify and interpret a version?

6.2 Implementation Challenges

The implementation of a web service needs to be suitable for its evolution. In this regard, we have identified the following challenges:

- How to design and organize the code for evolving web services?
- How to provide protocol level support for versioning?
- How to fix defects that affect multiple versions?
- How to design the data storage for evolving web services?
- How to automatically test multiple versions of a web service?
- How to implement sandbox environment for testing different versions?

6.3 Documentation Challenges

A documentation of the web service is used by the consumer to understand and explore the features of the web service. Evolving web services introduce several challenges related to their documentation as follows:

- How to produce version specific documentation and usage examples for a web service?
- How to provide an interactive explorer for different versions a web service?
- How to generate changelogs between versions?

6.4 Communication Challenges

Communication plays a big role behind an evolving web service because the communication channels are used to share information about future changes as well as to collect feedback about defects and feature requests. The following is a list of communication related challenges with an evolving web service:

- What communication channels should be used?
- How to automatically notify the consumers about the changes?
- If multiple channels are used, how to aggregate the data from multiple channels?

From our review of the existing literature and industry practices, we have already discussed their suggested approaches to solve some of the aforementioned challenges. Even though these challenges are generic in nature, we recognize that the answers to these challenges may rely on the specific use-case of a web service. However, due to the lack of standard approaches and tool support to address these challenges, once a decision is made, it typically requires custom implementation to solve these general problems. We identify this lack of standards and tool support as an opportunity for innovative future work to simplify the evolution of web services.

7 Conclusion

Web services evolve for real world reasons, such as to produce business value, fix defects, adapt new technology and so on. However, there is little to no built-in support for evolving web services in the commonly used technologies of today. In this paper, we identified the reasons that cause a web service to change, techniques to implement the change and the key challenges associated with the evolving web services. Understanding these aspects of an evolving web service helped us to identify the opportunity for innovative future work.

From our systematic literature reviews, we have identified the change drivers behind evolving web services and several approaches to deal with the various aspects of an evolving web service, such as deployment, versioning, protocol and source code level implementation, documentation and communication. In addition to the literature review, we have found different approaches to deal with these challenges from three real world popular web services.

Based on the review, we have summarized our findings of the key challenges with evolutionary web services in four high level categories a) Deployment, b) Implementation, c) Documentation and, d) Communication. For each of these categories, we have compiled a shortlist of challenges in question forms. This list of questions can be treated as a checklist for working on evolving web services. We have also discussed different approaches to solve some of the challenges from our review of the existing literature and industry practices. In our future work, we aim to reconcile and augment these fragmented solutions to provide a standard approach and tool support for evolving web services.

References

1. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
2. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
3. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)

4. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
5. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
6. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`