Automated Service Evolution

Dynamic Version Coordination Between Client and Server

Virginia Smith
Business Technology Optimization
HP Software
Roseville, CA, USA
virginia.smith@hp.com

Bryan Murray
Business Technology Optimization
HP Software
Bellevue, WA, USA
bryan.murray@hp.com

***Abstract*— While client/server integrations may be loosely coupled so that the evolution of the service endpoints occurs with minimal impact on backward compatibility, installing and configuring application upgrades to take advantage of new application functionality is still painful for customers and involves manual work by administrators. Coordinating changes in version between client and server has traditionally been done using either a central registry or through manual configuration, both of which can be error prone. The authors propose that clients and servers be aware of the versions they consume and provide and that they coordinate between themselves to adapt dynamically to new versions.**

Keywords - automation, versioning, web service, REST client/server, evolution

I. INTRODUCTION

Upgrading deployed software has been an ongoing problem in the software industry. Much of the research has focused on this problem in several main areas. One area of focus is adaptive software where a software system can adapt itself in response to specific internal or external conditions as detailed in [10]. This research focuses on the software system itself (the service) and does not address the problems that occur in the client/server communication when a service is upgraded. Another area of research focuses on maintaining backwards compatibility to eliminate client problems after the service upgrade, usually through the addition of a new component. Some examples are [5] which uses adapters and [4] which uses an interface monitoring component.

This version evolution problem is even more acute today as more and more functionality is deployed as web services where the client and server are independently controlled. In addition, in many enterprise deployments, multiple versions of client and services must coexist due to business requirements or software supplier constraints. Services must evolve to handle new customer requirements and clients want to know when a service is upgraded so they can take advantage of new functionality immediately without waiting for a manual configuration. To handle this dynamic environment, clients must be able to deal with multiple service versions and services must be able to deal with multiple client versions. The authors propose a method of dynamic negotiation between client and server that enables them to adapt to this kind of deployment environment.

We showcase our proposed solution using the Representation State Transfer (REST) [2] client/server architectural style as defined by Roy Fielding in his doctoral dissertation. One of the key benefits of the REST architectural style is that the client and server become much more loosely coupled than was possible using the operation-oriented approach. A RESTful architecture is being adopted by many applications to enable easy and consistent integration development. While RESTful application integrations may be loosely coupled and, therefore, the evolution of the service endpoints occurs with minimal impact on backward compatibility, installing and configuring application upgrades to take advantage of enhanced application functionality is still painful for customers and involves manual work by administrators. The authors demonstrate dynamic version coordination between client and server using a method that enables RESTful integration participants to seamlessly configure themselves to use a new endpoint version as the client is updated and/or a new service version becomes available.

All services, even those written using the REST architectural style, will need to modify their data models at some point. With care, a client and service can continue to work even with many data model changes, as long as those changes are backwards compatible. The W3C TAG draft document [8] on versioning languages addresses the issue of maintaining compatibility between versions of a language and provides insight into a number of design patterns for constructing extensible languages and defining a language versioning strategy. These strategies help to ensure compatibility between versions of a language and thus between a service and its clients.

However, even when there is language compatibility between a service and its clients, there are reasons that may prompt a server to move to a new version. Bug fixes are one scenario. Another scenario is when there is a functionality change in the content of a client request to the server. For example, a server might support new query parameters. The client can then add new logic to communicate with the server using the new functionality. A third scenario occurs when there is an expectation of some new action related to the resources controlled by the service. For example, a resource has a state attribute of 'on' or 'off' but a new state is introduced such as 'standby'. The new language version may be compatible with the old version but there is new functionality represented by this change. The service supports the new state attribute with some specific actions. In fact, the service might require that clients make use of the new state in a new version of the service. In these scenarios, the client is not satisfied with simply maintaining language compatibility with the server. The client is interested in using the latest version of a service to take advantage of new functionality or new language elements. Therefore, even with a well-defined versioning strategy, there is a need to address the ease of migration of a service and its clients to newer versions when that migration is desired.

The remainder of this paper is organized as follows. Section II defines the terms used throughout the paper and presents an example that is used to demonstrate the concepts. Section III describes the problem that occurs when individual applications are combined to deliver an enhanced solution to the customer. Sections IV and V present an approach to solving this problem using common REST technologies. Section VI offers suggestions for implementing version evolution in non-RESTful environments. Finally, the paper concludes with thoughts on the general applicability of the approach.

II. TERMS

An **integration** is a point of communication between two applications for the purpose of sharing resources. For example, the operations management application can open a ticket in the help desk application when an alarm is raised. The business impact analysis application can add additional relevant information to the help desk ticket to help the operator triage the problem.

There are two parties to every integration point, a **client** and a **server**. In REST architecture, these are defined as the two main connector types. “The essential difference between the two is that a client initiates communication by making a request, whereas a server listens for connections and responds to requests in order to supply access to its services. A component may include both client and server connectors.” [2]

An **endpoint** is the implementation of a service interface. In a RESTful web service, it is defined by a set of related URLs and the HTTP methods that are valid for those URLs. The endpoint implementation acts as the server in an integration. The term **service** is also used here to mean the service endpoint.

III. ASYNCHRONOUS MANUAL CONFIGURATION

While loosely coupled integrations allow for the client and server to evolve independently, upgrading to a new version can cause integration configuration problems. With multiple versions of the client and multiple versions of the server available in the field, it is necessary to configure which version of the server a client connects to. This is often a manual process that is performed by administrators and is error prone. Making the matter worse is that the applications participating in integrations rarely follow the same upgrade timeline. When and how should an administrator configure a new version of an integration (e.g., reconfigure the endpoint URLs) and what happens if there are multiple application versions that exist in a customer’s environment?

Consider the example of an IT management solution. This solution is enabled through integrations between four related applications as shown in Figure 1. This product suite solution is now being upgraded to enable additional new collaboration between the applications. Each application must implement its part of this new collaboration functionality. The products have the following schedules for the release of the version that will support the enhanced solution:

- Product A: version 5 is already released.
- Product B: version 6 will release in 2 months.
- Product C: version 7 will release in 3 months.
- Product D: version 2 will release in 6 months.

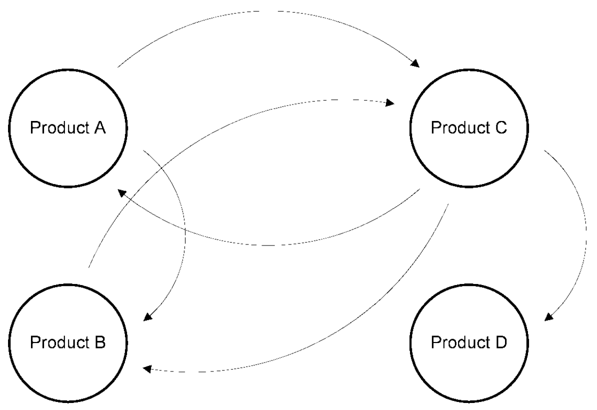

Figure 1. A 4-product solution showing the integration points.

While it is difficult to synchronize the release timelines of any two products developed independently, it is even more difficult to synchronize the upgrade of different applications in a customers’ environment where there may be sets of constraints by users of those applications on availability, risk of change or introducing incompatibilities, etc. The administrator must not only install the new version of the application, but also reconfigure new versions of all of the integrations between that application and other applications. Some applications may or may not be ready for a new version of an integration, making the upgrade process error prone. This results in customer frustration and increased support calls. As a result, customers are sometimes very slow to upgrade their applications. This can have a detrimental impact on the ability to bring end-to-end solution improvements to customers.

The authors propose an approach that enables dynamic version coordination between client and server. The approach defines how a client can automatically discover when a server is upgraded and how the client can reconfigure itself to use the new version of the server without requiring either a central registry or manual intervention by an administrator.

IV. AUTOMATED EVOLUTION

Through careful orchestration of the messages exchanged and the incorporation of version information in the messages, integrated applications can maintain their relationship automatically, always using the latest version shared by the client and server. This significantly improves the decoupling of individual product releases for integrated applications and makes the deployment of enhanced integrations and solutions a simpler, more automated process.

The authors’ proposal is composed of two behaviors: Discovery and Notification. Discovery is used to assure that when a client starts, it is using the latest version of the service that it supports. Notification is used to inform the client of an available new service version when the client has been running and the service was asynchronously updated.

The Discovery behavior defines how a service advertises its capabilities, and how a client approaches using the service. The key elements of Discovery are:

- Each application that provides services to integrating partners makes available to the client information about what versions it currently supports and how to access each version.
- Clients are expected to access this information when they begin using the service and select the appropriate version of the service to access the resources of interest.

The Notification behavior defines a process for independently updating a client or service without updating, manually reconfiguring, or restarting other applications. For example, it allows for installing a new version of a service (in parallel with an existing version of the service) without requiring a manual reconfiguration of an application that is a client of that service. The Notification behavior defines how a service informs a client that a newer version of the service is available, and how a client behaves upon receiving such a notification. The key elements of Notification are:

- When a server receives a message from a client that is not the latest version, the server includes a notification that a new version is available as part of the response to the client (along with the location of the new version). The location of the version information document is also included in the response.
- When a client receives a notification indicating a newer version, it may follow the link to the latest version information document and discover the server versions there or it can follow the link to the requested resource using the latest version of the service. If the client does not support a newer version, it ignores the new version notification.

There are significant benefits to this approach. No endpoint registry needs to be maintained, no periodic checking for new application versions needs to occur, and no manual configuration of the upgraded client or server application is necessary. The version information document is always up to date and the binding of the server and client occurs at the last possible time. Clients can seamlessly configure themselves to use a new server version as the client is updated and the server version becomes available.

V. IMPLEMENTING DISCOVERY AND NOTIFICATION BEHAVIORS

This section will map the proposed dynamic version coordination approach to an implementation suitable for use in RESTful services.

A. Discovery

In order to address the Discovery behavior described in Section IV, the authors propose using the Atom Publishing Protocol (APP) Service Document to advertise the versions that a service currently supports. The APP specification [3] defines a Service Document to be a set of Workspaces, each containing references to a set of Collections. The APP specification does not attach any particular meaning to a Workspace.

The authors define a new element, `version`, to be a child of the APP `workspace` element. The `workspace` element already groups collection references into a cohesive set. The addition of the `version` element adds the concept of version to a workspace. Multiple workspaces may have the same value for the `title` element, as long as the value of `version` is different for the two workspaces. An example of a Service Document using the new `version` element is shown below. The example shows how the workspace grouping is used to advertise two versions of a service. Note the difference in the URLs for the collections.

```
<?xml version="1.0" encoding="utf-8"?>
<service xmlns="http://www.w3.org/2007/app"
    xmlns:atom="http://www.w3.org/2005/Atom"
    xmlns:v="urn:x-auto-version:version">
  <workspace>
    <atom:title>Help Desk Svc</atom:title>
    <v:version>1</v:version>
    <collection
        href="http://example.org/incidents">
      <atom:title>Incidents</atom:title>
      ...
    </collection>
  </workspace>
  <workspace>
    <atom:title>Help Desk Svc</atom:title>
    <v:version>2</v:version>
    <collection
     href="http://example.org/v2/incidents">
      <atom:title>Incidents</atom:title>
      ...
    </collection>
    <collection
     href="http://example.org/v2/operators">
      <atom:title>Operators</atom:title>
      ...
    </collection>
    ...
  </workspace>
  ...
</service>
```

The `version` element allows a service to advertise multiple versions of its endpoint(s) with links to resource collections as a cohesive set for a given version. A service supporting multiple workspaces before adding a second version can still support multiple versioned workspaces. A client can easily determine whether the service supports multiple versions by searching for the `version` elements and selecting the workspace(s) to use based on the available versions and the versions supported by the client.

B. Notification

In order to address the Notification behavior described in Section IV, the HTTP Link header [7] is used in response messages. Use of the HTTP Link header allows the Notification behavior to work with any media type. The Link header includes a URI reference and an indication of how the resource indicated by the URI reference is related to the resource in the response body. Two relation types are used in the Notification behavior. First, the `service` relation defined in the Web Linking specification [7] is used to identify the location of the Service Document. Second, a new relation is defined to indicate the location of a resource using the latest version of the service: `urn:x-auto-version:new-service-version`.

A link with the `service` relation can be included in any response message from a service. The link must be included when a newer version of the service is available. The URI reference in a service link identifies the location of the APP Service Document used for the Discovery behavior.

A link with the `new-service-version` relation indicates that the service provides a newer version than the client was accessing in the request message. The resource referenced by the URI is the resource the client requested, but in a newer version of the service. The `version` link parameter is also defined for the `new-service-version` relation. This parameter indicates the new version of the service and must contain the latest version supported by the service. The client may also access the Service Document for information on how to access other versions of the service if appropriate. The response containing the `new-service-version` link will use the same version that the client used in the request. A `new-service-version` link must not be included in a response message unless the service supports a newer version.

An example of how the links are used in a response sent from a service is shown below. The example shows how the service link is used to indicate the location of the service document, and how the new-service-version link indicates the location of the requested resource using a newer version of the service.

```
Link: <http://example.org>; rel="service"
Link: <http://example.org/v2/incidents>;
  rel=
   "urn:x-auto-version:new-service-version";
  version="2"
```

The `new-service-version` and `service` link relation types allow a service to notify a client that a newer version of the service is available. A service indicates the location of both a newer version of the referenced resource and the service's Service Document. A client can use the referenced Service Document to find the available versions and determine which version is appropriate. The client also has access to the newest version of the resource it was accessing. This document does not define any meaning for the `new-service-version` and `service` link relations in requests sent from the client to the service.

C. Service Actions

When a service deployment is updated to support a new version, it is important for the service to continue supporting one or more older versions to allow for clients that cannot be upgraded at the same time and preserve loose coupling between a service and its clients. The service provides an updated Service Document advertising the new version of the service and one or more supported older versions. In the case where a service receives a request sent to an older version, it notifies the client of the availability of the newer version.

It is not difficult for the service to support a newer version of the Service Document. All requests, regardless of version, will return the same Service Document listing all of the available versions. The Service Document for the service should always be at the same location. In any case, the `service` link relation will always indicate the location of the Service Document.

Support for the notification to clients when a newer version is available requires that older versions of a service are aware that a newer version is available. This awareness only needs to extend to the ability to add the HTTP Link header to the response where the request used an older version.

There are four use cases that occur in a multi-version environment. The following discussion will use version 1 to mean an older version of the application (client or service), and will use version 2 to mean a newer version of the application. With respect to the client, the version is intended to indicate which version(s) of the service the client understands. That is, a version 1 client understands only version 1 of the service and a version 2 client understands version 2 of the service and also supports version 1 of the service.

a) Version 1 service receives version 1 client request
b) Version 1 service receives version 2 client request
c) Version 2 service receives version 1 client request
d) Version 2 service receives version 2 client request

The cases where a service receives a message from a client matching its version (use cases a and d above) are not interesting and will not be discussed here. In addition, a well-behaved client will only send messages to a service that the service will understand because the client has performed a discovery of supported versions of the service. Thus, use case b will not occur in a well-behaved environment.

The main concern is with an 'evolving state' where the client and server are out of sync with respect to their versions (use case c). When a version 2 service receives a message from a version 1 client, it means the service has been upgraded to a newer version while the client remains at an older version. In this situation, the service will generate a response using the same version that the client used, but will add the HTTP Link header indicating that a newer version is available and where to find it.

D. *Client Actions*

A well-behaved client will initially start from the Service Document for a service in order to find the resources in which it is interested. A client is given the address of the Service Document through configuration performed when the client is first deployed. The client will choose the appropriate version of the service from the Service Document.

Version selection is done by reviewing all of the `workspace` elements within the Service Document, noting their respective versions based on the value of the `version` element. The client will choose to use the endpoints in the workspace(s) with the highest version that is less than or equal to the highest version the client understands. Once the workspace(s) have been selected, the client can proceed with the discovery of resources.

If the installed client is version 1, the client will choose the version 1 workspace(s). The client may receive newer version notifications from version 2 of the service but will continue to use version 1 since that is the latest version it understands. In the case where the installed client is version 2 but the service is version 1, the only workspace available to the client will be version 1. (Normally, a client will continue to support several versions of the service for some period of time in order to handle this use case.) Later, after the service is upgraded to version 2, the next time the client accesses the service, it will receive a notification in the response that indicates a newer version of the service is available.

A client that is using an older version than the highest it can understand should check every response from the service to see if it includes the HTTP Link header indicating a newer version. Just because a new version of the service was deployed, does not mean that the clients of that server must be restarted. When a client receives the notification of a newer version it should either start the discovery process over, or update its cache for the resource location and continue on using the newer version for the resource.

As described for services, there are four use cases that will be examined from the client's point of view. As mentioned previously, the following discussion will use version 1 to mean an older version of the application, and will use version 2 to mean a newer version of the application. With respect to the client, the version is intended to indicate which version(s) of the service the client understands. That is, a version 1 client understands only version 1 of the service and a version 2 client understands version 2 of the service and also supports version 1 of the service.

e) Version 1 client receives version 1 server response
f) Version 1 client receives version 2 server response
g) Version 2 client receives version 1 server response
h) Version 2 client receives version 2 server response

As with services, the case where a client receives a response from a service with the matching version (use cases a and d above) are not interesting and will not be discussed here.

When a version 2 service sends a response to a version 1 client, it must add the HTTP Link header (defined in Section V.B) as the notification to the client that a newer version is available. In this case the client is not capable of understanding the newer version and will ignore the notification. It is possible that the client will not even be checking for newer versions if it is already using the highest version it understands.

If a client receives a response message containing the HTTP Link header indicating a newer version is available and it supports a later version of the server than it is currently using, it should use the links to begin using the newer version.

E. *Example Scenario*

The following scenario demonstrates the Discovery and Notification behaviors. This scenario occurs when the client is upgraded to a newer version before the service. The sequence of steps involved is shown in Figure 2. The opposite scenario where the service is upgraded first is very similar and involves the same actions although in a different order.

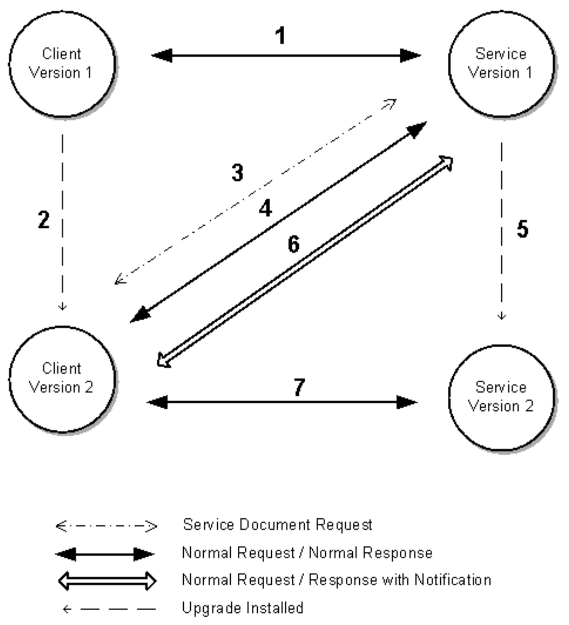

Figure 2. Steps to evolve client and service to new version

The actions that occur at each step are the following:

Both the service and its client are at version 1 and continuously execute the normal request/response cycle.

The client is upgraded to version 2 although it still supports version 1 for ease of migration. (The service is unaware of this upgrade.)

As part of its normal startup, the client requests the service's Service Document. The client selects version 1 of the service. In this scenario, version 1 is the only version currently supported by the service and therefore is the only version available in the Service Document.

The client and the service continue to execute the normal request/response cycle as if both were at version 1.

The service is upgraded to version 2 although it still supports version 1. (The client is unaware of this upgrade.)

The next time the client sends a request to this service (still using version 1), the service sends back the normal response but this time it includes a notification to the client that there is a later version of the service available.

Upon receiving a response that includes a newer version notification, the client automatically begins to use version 2 of the service from this point forward.

F. Performance Impact

A demonstration of the described research has been coded as APP-based client and service, and a pilot project within HP has been initiated. The impact of the Discovery behavior on message size and processing time is minimal since Discovery is used only when a client connects to a service for the first time or after a service notifies a client of a new service version. These are infrequent events.

The impact of the Notification behavior on message size and processing time is more important since it can affect most messages between the client and service. The demonstration service uses different URLs for different versions, always sends the link to the service document, and conditionally sends the link to the new resource version. The message size is increased by the size of these two HTTP headers. The processing time is increased by the time to write the headers.

The processing time impact for every message is the check for the presence of the notification. If present the Discovery behavior is initiated. The demonstration client checks for notifications only when it is not operating with its most recent version, otherwise the client can ignore them thus incurring no extra processing time.

The minimal change in processing time and message size is deemed a good trade-off for the significantly reduced manual configuration normally done for version changes of services and their clients.

VI. EXTENDING THIS APPROACH TO OTHER TYPES OF SERVICES

The previous section describes an implementation of the proposed approach to automated service evolution that can be easily applied to RESTful services and clients. There are other alternatives that could be used in this same context. For example, the HTTP Link header is explicitly defined as semantically equivalent to an HTML `LINK` element [8] or `atom:link` elements in an Atom feed [4]. The advantage of choosing the HTTP Link header is that it can be used to provide version notifications independent of the media type used for the data in the response body.

There are some types of services, for example SOAP-based services, where it is less obvious how to apply the proposed approach to enable independent version evolution of applications. It is still necessary to provide both Discovery and Notification behaviors.

Using a non-RESTful architecture (such as SOAP), applications can still perform the Discovery behavior by using the APP Service Document as described above. However, other approaches may be more natural for the environment. For instance, versions of services could be advertised in a registry. The basic actions that the client goes through for discovery are similar to the approach described above, just using a different source for the version information.

The HTTP Link header will continue to work for the Notification behavior in a non-RESTful architecture as long as HTTP is used as a transport. When HTTP is not used as a transport, it will be necessary to find a way to convey the availability of a newer version from the service to the client either as part of the transport or in the body of the message itself. For instance, XML-based messages could include an optional element or attribute in the message body to provide the notification.

VII. CONCLUSION

The original motivation for this solution was the integration of HP enterprise management products in order to bring more comprehensive, end-to-end, and synergistic IT management solutions to our customers. However, the authors feel that this approach provides value in general situations where the client and server are under the control of different organizations as is the case for many web services. This approach enables a seamless, automated evolution of web services and their clients.

REFERENCES

[1] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, "Hypertext transfer protocol—HTTP/1.0", IETF RFC 2616, May 1996.

[2] R. T. Fielding, "Architectural styles and the design of network-based software architectures", PhD Dissertation. Dept. of Information and Computer Science, University of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (last access October 27, 2010).

[3] J. Gregorio and B. de hOra, "Atom Publishing Protocol", IETF RFC 5023, October 2007.

[4] B. Kalali , P. Alencar , D. Cowan, "A service-oriented monitoring registry", Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, October, 2003.

[5] P. Kaminski , H. Müller , M. Litoiu, "A design for adaptive web service evolution", Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, May, 2006.

[6] M. Nottingham and R. Sayre, "Atom Syndication Format", IETF RFC 4287, December 2005.

[7] M. Nottingham, " Web Linking", IETF Draft, January 2010.

[8] D. Orchard, ed., Extending and Versioning Languages: Compatibility Strategies, World Wide Web Consortium, September 2008.

[9] D. Raggett, A. Le Hors, I. Jacobs, eds., HTML 4.01 Specification, W3C Recommendation 24, December 1999.

[10] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges", ACM Transactions on Autonomous and Adaptive Systems, May 2009.