

Service Versioning across Distributed Systems

Upgrading Processes across Loosely Coupled Systems

In loosely coupled systems, the development lifecycle of services and the service consumers is decoupled requiring compatibility assessments to be made when introducing major service upgrades. Using manual procedures to assess compatibility can be time consuming and cumbersome particularly when multiple services need to be upgraded simultaneously.



Background

The initial implementation of a loosely coupled system is designed to deliver a spectrum of functionality. Over time, this required functionality changes. These changes may be of two types: major or structural changes, where the functionality between the consumer(s) and the service(s) is fundamentally altered, and minor adaptations, able to be introduced without impacting the functional relationships between the co-operating services.

The conventional approach to introducing major changes is to ensure the new service version is able to support the interactions generated by the existing consumers, but ensuring this “backward compatibility” is complex. Manual backward compatibility assessment, particularly when large numbers of services are engaged, requires careful attention whilst the automated methodologies rely on Registry based solutions¹ that carry compatibility assumptions.

When dealing with stateful services, further complications arise as there is a need to ensure the upgraded service has an understanding of the transaction state, particularly in circumstances where access to third party functionality is required.

The HELIXsystem Service Versioner

The HELIXsystem Service Versioner enables multiple service versions to be concurrently deployed across a single infrastructure and enables multiple services to be upgraded simultaneously, without introducing compatibility risk or adding to system complexity.

The HELIXsystem Service Versioner delivers this by using service end point behaviour as a first class concept at the Registry. The HELIXsystem Service Versioner removes the embedded assumption of the current automated approach that assumes the latest major service version iteration will display compatible functionality, by formally verifying the behavioural compatibility between the consumer and the service.

In order to deliver this functionality, the Registry needs to be furnished with the end point behavioural descriptions of the services. These may either be extracted from the formal process description² or, should this not be available, derived from the BPEL or other appropriate description language. Once the end point descriptions have been derived, it is necessary to ensure the “library” of descriptions is complete. This is achieved by associating each behavioural description with the services recorded at the Registry.

Once the library is complete, the consumer interrogates the Registry for the latest service version able to present compatible functionality. The returned service does not need to display functionality that precisely mirrors the behaviour requested by the consumer. All that is required is the returned service must be able to process the interactions generated by the requesting consumer(s). In circumstances where multiple service versions are able to support the requested behaviour, the Registry returns the URL to the latest compatible service version and the consumer is automatically upgraded to this service.

This process now repeats across the system, enabling the entire service network to be upgraded without the introduction of incompatibility risk to existing processes.

¹ Most usually based around name space and version number conventions

² See the associated white paper “*Changing Gears on Process Discovery*” for details
Hat Trick Software Limited
25 - 31 Tavistock Place
LONDON WC1H 9SF



An Example

Below is a (pseudo) description of the behaviour required by a consumer acting as the Buyer in a Purchasing conversation.

The consumer initiates the sequence by sending a QuoteRequest to the receiving service (Seller). This service may either return Quote or OrderFailed. If the Seller returns Quote, the Buyer may respond with Order and the transaction concludes with a Confirmation interaction from the Seller.

```
conversation Purchasing@Buyer {
  role Seller;

  QuoteRequest to Seller;

  if @ Seller {
    Quote from Seller;
    if @ Buyer {
      Order to Seller;
      Confirmation from Seller;
    }
  } else {
    OrderFailed from Seller;
  }
}
```

Representation 1: Pseudo code detailing consumer to service transaction logic for a *Purchasing* example

From this we can determine the service behaviour required by the consumer. In this example the consumer (Buyer) requires the service (Seller) to:

- ▶ receive QuoteRequest
- ▶ send Quote, or send OrderFailed
- ▶ receive Order (if Quote)
- ▶ send Confirmation (if Order)

The consumer supplies a reference to this behaviour when communicating with the Registry. The Registry searches the library of service behavioural descriptions and returns the URL to the latest compatible version of the Seller. This is provided below:



```
conversation Purchasing@Seller {
    role Buyer, CreditAgency;

    QuoteRequest from Buyer;
    CreditCheck to CreditAgency;

    if @ CreditAgency {
        CreditOk from CreditAgency;
        Quote to Buyer;
        if @ Buyer {
            Order from Buyer;
            Confirmation to Buyer;
        } else if {
            Cancel from Buyer;
        }
    } else {
        NoCredit from CreditAgency;
        OrderFailed to Buyer;
    }
}
```

Representation 2: Pseudo code detailing the returned service version to consumer request

We can see that the behaviour provided by the returned service (Seller) in this example is as follows:

- ▶ receive QuoteRequest
- ▶ send Quote, or send OrderFailed
- ▶ receive Order or receive Cancel (if Quote)
- ▶ send Confirmation (if Order)

Although the Seller behaviour includes a set of interactions with a CreditAgency, this does not impact the Sellers behaviour with the consumer.

As can be seen, the interactions between the consumer and the service are not precisely matched, however the provided behaviour is compatible to that requested by the consumer, as the Cancel response is an optional interaction.

This illustrates that the returned functionality does not need to precisely mirror the behaviour requested by the consumer. All that is required is the returned service must be able to support the behaviour requested by the consumer. This behaviour may be a subset of the spectrum of behaviour provided by the service.

Compatibility and Stateful Services

When dealing with stateful services, it may be necessary to additionally identify compatibility in the context of the overall transaction logic. This circumstance typically arises when suspending and restarting long running processes. In these circumstances, the transaction re-commencement point needs to be consistent with the processing logic at the time the transaction was suspended. This requires the internal state of the upgraded service version to be compatible with the service version in use at the time the process was suspended.



This is illustrated below. The following behavioural description is an extension of *Seller* representation used in the earlier example. In this example the *Quote* returned to the consumer (*Buyer*) is contingent upon a satisfactory response from the *Distribution*³ role, with the request to *Distribution* being contingent upon a satisfactory response being received from the *CreditAgency*.

```
conversation Purchasing@Seller {
  role Buyer, CreditAgency, Distribution;

  QuoteRequest from Buyer;
  CreditCheck to CreditAgency;

  if @ CreditAgency {
    CreditOk from CreditAgency;

    RequestEstimate to Distribution;
    DeliveryEstimate from Distribution;

    Quote to Buyer;

    if @ Buyer {
      Order from Buyer;

      parallel {
        Delivery to Distribution;
      } and {
        Confirmation to Buyer;
      }
    } else if {
      Cancel from Buyer;
    }
  } else {
    NoCredit from CreditAgency;
    OrderFailed to Buyer;
  }
}
```

Representation 3: Stateful service behaviour

In this example, although the internal state of the service is invisible to the consumer, in order for the Registry to return compatible service behaviour, it needs to return a service version able to re-commence at the same point in the logic flow at the time of the service suspension. That is, compatibility needs to be matched both in terms of the behaviour requested by the consumer, and in terms of the context of the overall transaction logic.

The HELIXsystem Service Versioner addresses this by the consumer providing the business transaction identity information to the Registry. Using this identifier, the Registry retrieves the execution history for the service from the HELIXsystem Transaction Information Service. This provides a record of all the interactions relevant to the transaction instance including the execution event at the point the transaction was suspended. The Registry is now able to return the latest compatible service version with this compatibility being assessed both in terms of the end point behaviour required by the consumer and in the context of the overall transaction state.

³ This example assumes Distribution will always respond with a *DeliveryEstimate* response. The circumstance of a *NoEstimate* response is not handled in this example



Comparison with BPEL Engines

With regard to BPEL engines, once a transaction instance commences using a specific BPEL process version, any subsequent service interactions associated with the transaction instance are routed via the originating process version. This provides some of the functionality delivered by the HELIXsystem Service Versioner but can only be used in circumstances where all the services are BPEL implemented.

Further complications arise where co-operating stateful services are implemented in BPEL. In these circumstances, in order to reflect process change, the behavioural interfaces of the impacted services need to be synchronised and updated simultaneously.

This can be accommodated for transaction flows of short duration but difficulties arise when dealing with long running processes. In these situations, it is possible to encounter compatibility inconsistencies where a transaction instance suspends, and the BPEL encoded consumers and services are upgraded, as upon re-commencement, the interactions will be routed via the originating process version. When the transaction engages BPEL services that had not been engaged at the time of suspension, compatibility inconsistencies may be encountered. Whilst it is possible to address this situation using customised administration procedures, this is manually intense and fails to facilitate seamless deployment of new service versions.

Summary

In loosely coupled systems, the development lifecycle of services and the service consumers is de-coupled requiring compatibility assessments to be made when introducing major service upgrades.

Using manual procedures to assess compatibility can be time consuming and cumbersome particularly when multiple services need to be upgraded simultaneously. The current automated approaches either contain embedded compatibility assumptions or are only applicable where all the services are BPEL implemented. Further complexity is encountered when dealing with long running, stateful processes. In these circumstances identifying compatibility requires a further understanding of transaction state.

The HELIXsystem Service Versioner removes the embedded assumption associated with the current automated procedure by establishing a direct link between the behaviour offered by the service and the behaviour required by the consumer. This compatibility link is formally verified both in the context of the behaviour required and in the context of the overall transaction logic. This is achieved by enabling the Registry to utilise service behaviour as a first class concept.

The HELIXsystem Service Versioner is able to accommodate systems supported by a heterogeneous mix of implementation technologies and scales to accommodate the deployment of multiple new services simultaneously, as well as enabling multiple service versions, only some of which may offer compatible behaviour, to be concurrently deployed across a single infrastructure.