Source System Integration using the Framework for ORCHESTRA Services

Peter Kutschera, Severin Ecker, Gerald Schimak

Abstract
In this paper, an easy to implement, therefore time and cost efficient approach to integrate source systems into an ORCHESTRA service network will be presented. The framework recently developed at the Austrian Research Centers is described in detail and by examples in the following chapters. The basis for this framework was taken from the results of the ORCHESTRA project, which in short is concerned with interoperability of systems among different countries and organizations.

1. Introduction
Those who have ever tried to integrate legacy systems (also called source systems, like information systems or environmental monitoring systems, etc.) into other information systems or networks know about all the challenges that the integrator faces. Sources of problems are manifold such as different invocation or communication protocols. In addition, the data layout might not meet the requirements of the target application. Summing up the integrator must overcome interoperability issues and data harmonisation, which usually is very resource intensive, both in time and financial aspects.

One of the main advantages of the ORCHESTRA architecture is the assurance of interoperability of heterogeneous systems as long as they comply with the proposed ORCHESTRA architecture. Therefore, source systems need to be integrated to adopt the ORCHESTRA architecture only once and can then be reused in every compliant ORCHESTRA service network, so-called OSNs. Though integrating a legacy system into ORCHESTRA is not much different than integrating it into any other system (e.g. proprietary environmental monitoring networks) considering the amount of work that has to be done. However, once the legacy system is compliant, the strengths of ORCHESTRA come into play.

The following, Chapter 2 will give a brief introduction into the ORCHESTRA project outlining the fundamental ideas and concepts of the Framework for ORCHESTRA Services (FOS), chapter 3 describes the FOS in detail while problems and future work will be discussed in chapter 4.

2. ORCHESTRA
The ORCHESTRA project (Open Architecture and Spatial Data Infrastructure for Risk management, http://www.eu-orchestra.org) is one of the strategic projects of the European Commission in the domain of risk management infrastructures in Europe. It started as a European integrated project under the 6th framework program in September 2004. Its goal is to develop a software-architecture for a European wide information infrastructure capable of managing environmental as well as risk management information in the context of cross-border business processes. ORCHESTRA understands cross-border information exchange as a cross-institutions, cross-organizations, and cross-languages task.

1 Austrian Research Centers GmbH – ARC, A-2444 Seibersdorf
e-mail: Peter.Kutschera@arcs.ac.at, Severin.Ecker@arcs.ac.at, Gerald.Schimak@arcs.ac.at internet: http://www.smart-systems.at/
The vision of ORCHESTRA is to enable a data, information, or system provider to integrate information for risk management from different sources as well as to enable users to access information in the easiest possible way. ORCHESTRA will provide a framework for the integration of single information entities (like documents or maps) up to whole information systems. The ORCHESTRA framework defines numerous services, so-called ORCHESTRA Architecture Services that facilitate the above-mentioned tasks. These services should be easy to create to favour the adoption of ORCHESTRA and the FOS tries to help in this regard.

3. Framework for ORCHESTRA Services (FOS)

Picking any area in the field of information technology one will find numerous systems already in place that accomplish some or a big portion of the work that needs to be performed. Usually such systems have evolved and grown over years yielding highly complex applications. In addition, these architectures are usually not completely separated from underlying technology aspects. Having built such systems ourselves we were facing the problem of integrating them into OSNs without requiring huge amounts of effort. Our proposal for the interoperable integration of source systems, as legacy systems are called in ORCHESTRA, will be described in the following chapters.

3.1 FOS Architecture

In order to come up with an easy to use integration framework we have been looking into the structure and architecture of service oriented applications. In addition the results and ideas of the ORCHESTRA project were considered and taken into account. During this process we identified three layers that play an important role in the source system integration. Figure 3.1.1. shows these three layers.

- Layer 3 is called the Source System Layer: This is where the legacy systems are situated. As sample source system in this figure, we use an Oracle database (of course, you could replace it by any other legacy system as well as by an IEEE 1451 smart transducer). Interfaces and architectures are fixed.
here and are considered as not volatile. Whether they are really fixed (this would usually be the case with hardware sensors that have strictly defined protocols), or if these are mere software interfaces, which can be changed easily, does not matter. These systems are simply left untouched.

• Layer 2 is the Integration Plugin Layer: This is the layer where the real integration takes place. All plugins developed in this layer are designed according to the requirements and specifications of ORCHESTRA. Therefore, a plugin plays an important role in ORCHESTRA’s concept of self-describing ORCHESTRA services. The interfaces for the plugins are derived from the ORCHESTRA abstract service specifications and therefore are independent of any communication (infrastructure) technology. The ‘backend’ of those plugins that are concerned with the integration of a source system is exactly this source system. E.g., Figure 3.3.1 shows an ORCHESTRA FeatureAccessService plugin that integrates an Oracle database.

• Layer 1 is the Dispatcher Layer: This layer is concerned with the communication infrastructure used in the service network. Ideally, the code for this dispatcher is generated automatically so the integrator faces no tasks other than some configuration and choosing the proper technology.

3.2 Integration Process

Integrating a new source system and the creation of a new ORCHESTRA service becomes a fairly simple and straightforward process when using the FOS. In an ideal situation selecting the desired plugins their configuration and the automatic generation of the desired dispatcher is all that the integrator will have to do. Of course usually there is more work to do since it’s unlikely that someone else has already implemented an integration plugin that exactly satisfies your needs and fits to your legacy system. Therefore, a new plugin must be created first unless the integrator only uses some common source system back ends with already existing integration plugins. In addition any specific capabilities that can’t be automatically derived must be provided as well. The source system integration process follows a five step approach:

• Investigate the interface of the source system regarding data types for parameters, return values and the implemented communication protocols.

• Browse the ORCHESTRA service and interface specifications and find a proper one (or a selection of multiple ones) that reflect the desired functionality.

• Check the existing plugin implementations and see whether there is already a plugin which satisfies both requirements that have been found in the previous two steps. (E.g., there might be a FeatureAccessService\(^2\) plugin using JDBC and SQL). If there is no such plugin available, it has to be implemented.

• Check whether there is a dispatcher generator that outputs a dispatcher for the desired communication platform. If there is no generator there still might be an existing dispatcher implementation. The dispatcher must be implemented if no appropriate one exists.

• Take the individual parts from layer 1, 2 and 3 and assemble them. Usually there are some configuration settings that must be filled but this is as simple as editing a very short XML file. This results in a new ORCHESTRA service that integrates the legacy system into an ORCHESTRA service network.

As described above the integration process is rather straightforward. A valuable effect is that the programming that needs to be done is usually reduced to an acceptable amount. For every ‘new’ legacy sys-

\(^2\) A FeatureAccessService (FAS) can sloppy be seen as ORCHESTRA’s data repository. Therefore, data can be retrieved from and stored in the FAS.
One of the most important and convenient facts of the FOS is the automatic derivation of invocation meta-information. Java reflection allows the ‘Capabilities plugin’ to extract all operations and the parameter information of all integration plugins used for the new service instance. This information is exposed through the service capabilities which are an integral part of ORCHESTRA’s idea of self describing components. This information can then be exploited by generic clients in order to invoke the available service operation. Of course this is possible through all communication infrastructures, even if there is no WSDL file available. The generic client developed by ARCS, mostly for testing purposes, will be introduced in chapter 3.6 with the integration example.

### 3.3 Developing Integration Plugins

Implementing a new integration plugin becomes a rather trivial task for any intermediate java programmer, even more for a professional software developer. First the ORCHESTRA interface of the required service must be specified, which simply consists of transferring the operations that are described in the service specification into a java source file. The listing shown in fig 3.3.1 is one sample operation of the FeatureAccessService plugin developed by ARCS.

```java
@OrchestraInterface (OAInterface="FeatureAccess", OAServiceType="FAS")
public interface FASInterface {
    @OrchestraOperation (OAInterface="FeatureAccess", OAServiceType="FAS")
    public OAFeatureCollectionDocument getFeatures(OAQueryDocument qDoc)
    throws OANoApplicableCodeException,
            OAInvalidParameterValueException,
            OAInternalErrorException,
            OALegQueryException,
            OAInvalidQueryException;
    // …
}
```

Figure 3.3.1: Listing of a FAS Interface

Clearly, as the service interfaces are derived from the abstract service specification these do not change when switching to a new communication platform or a plugin of the same service type but a different source system backend. Such an interface must only be written once for any service interface. Hence chances are high that the source code for a specific interface is already available.

Then the class implementing this interface must be developed which can communicate with the legacy system that must be integrated. Sometimes this can be a time-consuming and difficult task but usually this should simply be a translation of one interface to another and probably include a change of communication technology.

Once these tasks are completed a new integration plugin is available and ready to use to integrate the source system in mind.
3.4 Generating a Dispatcher

The dispatcher is responsible for handling any details (e.g., working with SOAP messages) of the communication infrastructure that is used in the service network. It has to map the requests from the platform to the plugin method invocations. Such a dispatcher can often be generated automatically as it is for example the case for the SOAP platform or HTTP POST. If such a dispatcher generator is not available and no implementation for the focused target platform exists, a dispatcher, or more desirable a generator, must be created. The dispatcher is only concerned with the platform of the service network and not the technology and platform that the source system makes use of.

A dispatcher generator creates a dispatcher for a specific communication infrastructure and the given list of plugins. There is no requirement on the implementation of a dispatcher generator; nevertheless there are two dispatchers already available. One creates a W3C Web Service using SOAP binding and the second one creates a web service using HTTP POST binding. Using an already existing dispatcher generator reduces the work for writing a new web service to the task of writing a simple configuration file.

3.5 Available Components

While dealing with source system integration into OSNs and the development of the FOS quite a lot of components have already been produced. Both dispatchers/dispatcher generators and integration plugins have been developed as proof of concept and for the use in operational software. In the list of dispatcher generators there are:

- **axis2**: This dispatcher generator will output a W3C Web Service using SOAP binding. The configuration file simply lists the plugins that must be integrated into the web service as well as configuration parameters that are used to configure the resulting web service itself.
- **httpPost**: This dispatcher generator basically is the same as the one mentioned above except that it will produce a web service accepting HTTP POST requests from a client.

After checking various forums and open source projects it is our feeling that the java community will move on towards maven2 as the default build platform. We have therefore based the FOS on maven2. In this sense the dispatcher generators have been implemented as maven2 plugins thus they can easily be integrated into any maven2 build process.

For the integration plugin layer the list of currently available components is as follows:
- **Capabilities-Plugin**: This is probably the most important plugin. It is responsible for collecting all the meta-information from every plugin available. Furthermore, it derives the method meta-information automatically from these plugins so that these operations are exposed through the web service capabilities and can be used by generic clients. Every ORCHESTRA service instance must have a getCapabilities operation and this capabilities plugin is responsible for delivering useful data to the client.
- **WFSFas-Plugin**: This plugin provides the FeatureAccessService interface and “front ends” an OGC Web Feature Server.
- **Monitorable-Plugin**: This plugin exposes previously specified data through the ORCHESTRA Monitorable interface. It is used to monitor the service itself and reports a large amount of properties concerning both the server and the environment it is running in depending on the configuration. Service monitoring is an important issue in a SOA and this plugin is supporting this task.
There is also a sample implementation for a generic client available that reads the capabilities of an ORCHESTRA service. It then can be used to invoke service operations simply by interpreting the information retrieved through these capabilities.

Additionally there are a few sample services available e.g. a FeatureAccessService, a minimal service example and a monitoring service example.

### 3.6 Creating a Service - Integration Example

This chapter will show a sample integration of an OGC Web Feature Service and finally accessing it with the generic client. The integration has been worked out and implemented by ARC and this example is intended to both, proof that our approach is indeed working and expressing its simplicity at the same time.

First we have to write the java interface according to the abstract ORCHESTRA service specification of the FeatureAccessService. This has already been shown in fig 3.3.1. The next step is to implement the plugin class along with the specific capabilities needed. These capabilities include among other things the list of feature types that are supported by the service. Further on the implementation of the plugin contains the mapping from the FeatureAccessService parameter types to the request parameters expected by the Web Feature Server. The WFS implementation backend can be configured along with the request method (HTTP GET, HTTP POST).

This almost finishes the implementation of a FeatureAccessService. The last step is writing the configuration file for the dispatcher generator. Fig 3.6.1 shows the stripped-down configuration file that builds a fully functional FeatureAccessService with SOAP binding. (XML Namespaces and headers have been omitted for readability reasons)

```xml
<OrchestraService>
  <plugin>
    <implementingClass>
      at.ac.arcs.orchestra.fos.plugins.capabilities.DefaultCapabilitiesPlugin
    </implementingClass>
  </plugin>
  <plugin>
    <implementingClass>
      at.ac.arcs.orchestra.fos.plugins.fas.WFSFasPlugin
    </implementingClass>
    <option name="ConfigurationResource">fas.xml</option>
  </plugin>
</OrchestraService>
```

Figure 3.6.2: Listing of SOAP Dispatcher generator configuration file

After installing the service we can connect to it with the generic client and access all its operations without prior knowledge of the service and its interface.
Figure 3.6.1 shows the generic client before it a connection to the specified service has been established. After the client has been requested to connect to the specified service (using the menu) the service’s getCapabilities operation will be invoked and the result will be parsed. This can be seen in Figure 3.6.2.

Figure 3.6.4: Generic client, capabilities

The menu of the generic client is dynamically enhanced as more information about the service is retrieved. One can see that there are capabilities that contain basic information needed for operation invocation as well as specific to the FeatureAccessService. Browsing these capabilities will result in another new menu item which is dedicated to operation invocation as well as the listing of the FAS specific capabilities in the text area. The screenshot has been produced using the standard installation of the geoserver. Of course the implementation works with any backend that implements the Web Feature Server specification.

Footnote: The geoserver is an open source and free OGC WFS implementation that we have used at ARC while developing the FOS.
The list of operations corresponds exactly to the ones specified in the abstract specification of the FeatureAccessService (Friis-Christensen, An. 2007). In the WFSFas-Plugin only five methods were implemented, getCapabilities will automatically be added through the Capabilities plugin.

Creating a W3C Web Service that integrates the OGC WFS into the ‘SOAP world’ took less than a day using the FOS. From there on switching to a different communication infrastructure would only require writing a dispatcher; in case of HTTP POST it is just a simple configuration file meaning less than five minutes of work.

4. Problems & Future Work

Most problems we were facing during the implementation work of the FOS were related to tool incapacities, specifically when using maven2. Other than that most effort went into creating a framework which enables the users to come up with running services as fast as possible while keeping implementation work at a minimum.

Currently we are working on fine tuning the monitorable plugin which is used to provide information about certain properties of a server. Along with this we are implementing a service monitoring plugin which allows the implementing service to monitor the exposed properties of other services. Additionally, we are investigating and starting to implement a FeatureAccessService plugin which allows generic access to databases. That way many backend source systems which often are simply databases would be integrated at once.

5. Conclusions

Heterogeneity of systems especially in the field of environmental computation unfortunately is not a rarity. This often makes data exchange a tedious and complex task. Especially in case of environmental disasters such as floods or forest fire across borders, this situation not only is tedious but life threatening. The ORCHESTRA architecture tries to overcome these problems. However, the ORCHESTRA architecture will succeed only then if there is support for the system developers and integrator for adopting it. Otherwise, the effort imposed on the implementers most likely is too high to be feasible. Using the Framework for ORCHESTRA Services allows quick adoption of the beneficial ORCHESTRA architecture. That way data and service exchange can happen among different system while leaving these applications in place rather than requiring a full reimplementation of these systems.
This work has been co-funded by the European Commission’s IST programme, under Unit G5 “ICT for the Environment,” through the Integrated Project FP6-IST 511678 ORCHESTRA (see http://www.eu-orchestra.org). Special thanks to the members of sub-project 3 and 4.

Bibliography

Friis-Christensen, An. 2007, Specification of the Feature Access Service
http://www.opengeospatial.org/standards/wfs

