Abstract

Virtualisation and abstraction are the keys to an open, flexible and dependable service deployment on the Internet. The Venice Service Grid is a framework that provides a set of services and an abstraction layer supporting the development and deployment of services. As part of its abstraction layer, the Venice Service Grid contains a service compiler that creates the necessary files and code constructs needed during the life-cycle of a Venice service. This paper describes the Venice Service Compiler that shifts the technology-dependant WSDL file creation from the design phase to the implementation phase and discusses the compiler’s benefits to developing services for the Venice Service Grid.

1. Introduction

The Venice Service Grid is an open framework for distributed service-oriented applications that allows the easy creation, deployment, integration, and usage of services [3]. The means to achieve this are virtualisation of the resources used and abstraction from the underlying technology by using Web service technology. With the Venice Service Grid it is possible for a service provider to run a single security domain with services for the users of his domain. Additionally, it is also possible to connect to other trusted domains in order to gain access to the services provided by these domains. Transparent and secure access to services across such a service federation is managed by the Venice Service Grid.

The Venice Service Grid contains a set of infrastructural services that are responsible for managing services, distributing necessary information and providing useful functionality (Service infrastructure). Additionally, the Venice Service Grid provides an abstraction layer that uses these services in order to transparently deliver a high-level abstraction to the functionality the services offer (like single sign-on, service replication or service relocation at runtime). This abstraction layer also enables the transparent usage of the Web service technology lying underneath the Venice services (currently it abstracts from a Tomcat 5.x and Axis 1.x Web services tools combination).
The Venice Service Grid offers an API that hides as much of the technology-related specifics as possible while still being as generic as possible. The only information visible to a programmer are the URIs of the services and the data types used by those services. As part of this abstraction layer, the Venice Service Grid contains a special Web service compiler that automatically creates the necessary files and code constructs needed during the life-cycle of a Venice service. This paper focuses on the service development using this service compiler.

The remainder of this document is organised as follows. Section 2 discusses some related work on developing Web services using different frameworks. Section 3 introduces the Venice Service Grid and its architecture while section 4 discusses the Venice Service Compiler in greater detail. A final conclusion will be drawn in section 5.

2. Related Work

Several Web service frameworks are available. They have changed and matured over the years and address the areas of the Web service development cycle differently.

With Metro [14] (formerly known as the Java Web Services Development Pack, JWSDP), it is possible to pursue a code-first or contract-first Web service development approach. The contract-first approach starts with an already existing WSDL file and uses the “wsimport” tool to generate all server and client-side stubs and skeletons. Logging, security and other common tasks can be manually implemented in handlers (interceptors). They must then be configured for each and every service. Some commercial tools like BEA’s WebLogic server incorporate Metro technology.

In Apache CXF [12] it is also possible to follow the code-first or contract-first approach. For the contract-first approach, an existing WSDL file can either be used to generate the stubs and skeletons with “wsdl2java” or a WSDL file can be partly generated by using “xsd2wsdl”, “wsdl2soap” and “wsdl2service”. Logging and security must be integrated and configured by hand.

Spring-WS [15] supports only the contract-first approach. The messages that will be exchanged by the services must be defined in XML schema files. In contrast to the other solutions, the WSDL file can then be automatically generated by mapping the Java implementation’s data types to those XML data types according to a configuration at runtime. So this is not a pure contract-first approach, but a mixture between the code-first and the contract-first approach.

XINS [17] follows a specification-first approach. Here, all data types and functionalities must be defined in an XML format. Then, the XINS tools will generate the WSDL files, the client-side JAR,
the server-side WAR, unit tests and documentation in different formats. Security, logging and statistics are provided by XINS runtime code.

While these frameworks also allow a developer to use the contract-first approach, they rely on an existing WSDL file to start with. From a SOA life-cycle perspective, this WSDL file has to be provided at the end of the design phase [3]. In the Venice Service Grid, it is possible to design services using an abstract notation and then generate the WSDL files during the implementation phase. This makes the design phase independent of the Web service technology.

3. The Architecture of the Venice Service Grid

The Venice Service Grid is based on a pure service-oriented architecture (SOA) [3] and focuses on openness [11], dependability [5] and security [6]. Venice services can be deployed on the Internet and are fully autonomous. Secure and reliable access to the services is managed by the Venice Service Grid. In order to accomplish this task, the Venice Service Grid is constructed using four layers (cf. Figure 1). Although the abstraction layer is placed beneath the service infrastructure in Figure 1, it uses the functionality offered by the services of the service infrastructure layer. And each service of the service infrastructure layer naturally uses the abstraction layer in order to seamlessly integrate into the Venice Service Grid.

On the bottom layer, the concrete Web service technology can be found. Currently, the Tomcat servlet container [16] and the Axis SOAP engine [13] are used by the Venice Service Grid. Additional tools and libraries complement this software selection as needed. On top of this layer, an abstraction layer is positioned that abstracts from this underlying technology. It hides all implementation details that are concerned with the Web service technology in use and provides an application programming interface (for both the client and the service developers) that solely uses URIs for identifying and accessing services. In order to be more flexible, the abstraction layer uses dynamic invocation to access the services and does not use automatically generated stubs. The most important functionality of the abstraction layer addresses:

![Figure 1: The Venice Architecture](image)
- **Security**: As Venice services can be secured and their providers can define access restrictions, the abstraction layer contacts the correct SSO service (see below) and requests the necessary tokens transparently to the user. Additionally, the abstraction layer automatically communicates with the services using SOAP over HTTPS (if enabled by the services and the client) and checks the certificates and DNS entries.

- **WSDL handling**: As only the URIs to the Web services are visible to the programmer, all necessary WSDL handling is performed by the abstraction layer. The current endpoints are extracted, the data types prepared and converted as needed.

- **Data collection**: Prior to accessing a service, the correct service must be found according to security constraints, meta-data and other potential data relevant to the decision process. Therefore the abstraction layer API offers methods like `getServices(...)` that search for suitable services. Additionally, the abstraction layer provides the means to register for events sent out by Venice services (see Notification Services below).

- **Service access**: By providing API methods like `makeCall(...)`, the abstraction layer communicates with the services. It automatically inserts security tokens into the communication and generates all necessary technology-specific runtime objects (dynamic invocation) for the Web services communication. Data type conversion and registration is also transparent to the programmer.

The third layer of the Venice Service Grid consists of a set of services that range from service management at runtime, resource and service information and access, collaboration and communication, and utility services for building applications upon. Among these services, the most important are:

- **Single Sign-on (SSO)**: The Venice Service Grid uses a Kerberos-like token-based single sign-on strategy for enabling secure service access. The user authenticates once and subsequently requests service tokens for accessing services. These service tickets contain the encrypted user’s authorisation data and can only be decoded by the service they are issued for. [6]

- **Domain Information Service (DIS)**: This “bootstrap” service is used by any entity in a service domain to gather the required information for working in the domain. It provides the URIs of important services and the means for contacting administrators or obtaining general information about the service domain [8].

- **Information Broker (IB)**: When communication across service domains is needed, this service provides the necessary means to do this. In order to support scalability, it uses peer-to-peer technology for cross-domain communication while providing a Web service interface for the local domain.[7]
- **Notification Services (NS):** By implementing a generic publish/subscribe mechanism, this set of services enables distributed applications and services to subscribe for topics of interest and receive notifications when an event of interest appears. This avoids polling and can even be used for deferred message delivery if a client or service is currently not running. [8]

- **Presence Services (PS):** This set of services implements a presence system according to [2] and allows users and services to subscribe to each other and be notified of changes in state. Additionally, it is possible to apply certain rules / filters of notification to groups of users. [9]

A more detailed discussion on these and the remaining services provided by the Venice Service Grid can be found in [8]. The topmost layer of the architecture is intended for the services of application domains using the Venice Service Grid. For example, a Voice over IP application has been implemented; details are published in [10].

As mentioned above, the services offered by the Venice Service Grid are implemented as Web services directly on top of the Tomcat [16] servlet container and the Axis [13] SOAP engine. The services are described using the Web Services Description Language (WSDL [1]) and communicate with SOAP [4] over HTTP or HTTPS. The service interface definitions are separated in an abstract service interface definition containing the port types and messages, a concrete interface definition containing the effective binding, and a concrete service implementation definition containing the ports and locations of the real implementation. All data exchanged between the service entities are specified in publically available XML schema files. This allows for re-use of the WSDL and XML schema files in the service descriptions. The Venice service compiler helps managing those files and thus enables a faster service development. This will be discussed in the next section.

### 4. Service Development with the Venice Service Compiler

The life-cycle of the services within a service-oriented architecture is comprised of several steps that have to be completed in order to construct a service-oriented solution [3]. First, a service-oriented analysis and a subsequent service-oriented design phase specify the services’ functionalities, data types and interfaces. In case of Web service technology, the design phase ends with the specification of the WDSL files that represent service interfaces. In the development and test phases, the services are implemented and tested according to their specifications. When the implementation is finished, the services are deployed and later administered during run-time. Of these six phases, the Venice Service Grid currently addresses development, deployment and administration. While service administration is handled via dedicated services and interfaces in the Venice service infrastructure, development and deployment are being addressed by the Venice Service Compiler. In contrast to normal Web service development, the interface of a Venice service
is not specified using WSDL at the end of the design phase. Instead, an abstract syntax has been
declared that makes the design phase independent of Web service technology. During service
implementation, this abstract syntax is converted to WSDL.

The Venice Service Compiler is a flexible tool that supports the contract-first approach for Web
service development. In the contract-first approach, a service’s technical life starts with the
definitions of the interface and data types, only after that the coding starts. These data types and
interface definitions can be collected in order to build a repository. This approach fosters loose
coupling and reusability. So the Venice Service Compiler takes as input a service interface
description, and this description is defined using an abstract syntax (a combination of XML and
Java syntax has been chosen to represent a service’s interface). An example can be seen in Figure 2.
In line 843 an interface called “EMail” is defined. In line 846 its XML namespace is defined, and in
line 847 the location within the interface repository on a Web server is denoted. Then, the XML
data types used by the interface are imported in lines 850-853. The data types can be externally
designed as XML schema files with any XML tool. In lines 856-868 three operations for the
interface are declared. An operation definition is comprised of the result type, the operation name,
the input parameters and the faults the operation may raise.

```java
// This service type can be used to send emails.
interface EMail {

    // Imports the data types
    namespace "http://www.icsy.de/wSDL/basic/";
    location "http://www.icsy.de/-venice/wSDL/basic/";

    // Imported data types
    types basic "http://www.icsy.de/-venice/types/basic.xsd";
    types email "http://www.icsy.de/-venice/types/basic/EMail.xsd";
    types domain "http://www.icsy.de/-venice/types/domain.xsd";
    types faults "http://www.icsy.de/-venice/types/faults.xsd";

    // Sets the email address
    operation void setEmailAddress(domain:SS0Information sso, email:EMailAddress email)
     throws faults:AuthorizationFault;

    // Returns the stored email address
    operation email:EMailAddress getEMailAddress(domain:SS0Information sso)
     throws faults:AuthorizationFault;

    // Sends an EMail
    operation void sendEMail(domain:SS0Information sso, email:EMail email)
     throws faults:AuthorizationFault;
}
```

Figure 2: A Venice Interface Definition

Out of this abstract interface definition, the Venice Service Compiler produces output from four
different areas relevant to service development and deployment (cf. Figure 3):
WSDL: With the data defined in the “operation” sections of the abstract service interface, the Venice Service compiler generates a WSDL file containing the corresponding wsdl:portType definitions. Namespaces, data types and documentation are extracted from the abstract definition and converted into such a WSDL fragment. Next, the Venice Service Compiler generates a WSDL file for the SOAP bindings for each operation, soap:style and soap:encoding can be chosen. Additionally, a WSDL file containing a wsdl:service element with the concrete endpoints can be generated from an abstract service definition (not shown in Figure 3). These WSDL files include each other and thus create a valid WSDL document. The advantage fragmenting the WSDL definitions into several files is that the service interface definition can be reused and the binding can be changed without having to touch the other elements.

Implementation: The Venice Service Compiler will also produce several Java source code files for implementing the service. The WSDL2Java tool is used to generate the Java classes for the XML data types and the stubs and skeletons for Axis. On top of that, a Java class for binding a Venice service to Axis will be produced. This class makes it possible to change the underlying Web service technology at a later point without having to change the service implementation. But this binding class does not only contain the binding to Axis. It is possible to optionally generate several dependability and debugging mechanisms so that not every service implementation has to deal with these issues on its own:

- **WSDL compliance**: It is possible to integrate source code that checks at runtime whether the Java implementation is still compliant with the WSDL file of the service. This is helpful during the development of a service if the interfaces, faults or data types change. Any problem identified will be published using Log4J.
- **Statistics**: Knowing the current amount of users of a service or the amount of faults that have been raised due to different errors is very beneficial for service dependability. Thus it is possible to generate code counting the successful and unsuccessful service invocations as well as the authentication and authorisation faults. These values can be retrieved from the service by a special Venice interface at runtime and help identifying bottlenecks, misconfigurations or even attacks.
- **State management**: A Venice service can have different states (like initialising, running, disabled and migrated). If a state forbids service operation, this will be detected and the user notified.
- **Error handling**: The Venice Service Grid has a well-defined error model with a base fault and faults inhering from it. Normally, any unhandled Java runtime exception would be converted by Axis into a simple SOAP fault. In order to prevent this, code is generated that makes sure that all faults are descendents of the Venice base fault. All occurring exceptions are of course logged using Log4J in order to make debugging easier.
Logging: It is also possible to automatically generate Log4J code that notifies about every service usage prior to calling the actual implementation of a service.

Performance measurement: The Venice Service Grid provides a mechanism and a service for making performance measurements. The code for sending performance records to this service can be automatically inserted into the binding. This helps identifying bottlenecks during the service development and test phase.

As a final step, a skeleton of the actual Java implementation will be generated. Here, all things relevant to the implementation will be generated. This includes importing the correct types, defining the correct method signatures and inserting the documentation from the abstract interface definitions as Javadoc.

Deployment: As the Venice Service Grid is currently using Axis 1.4, the deployment descriptor which is necessary for deploying a Web service into a container can be automatically generated by the Venice Service Compiler. If the underlying technology changes, the deployment mechanisms can be easily adjusted.

Documentation: In a service-oriented architecture, documentation is not only important to developers and stakeholders but also to the customers as they rely on well-documented WSDL files. So the Venice Service Compiler takes the documentation in the abstract interface definitions and inserts it into every generated source code related to it – to the WSDL files, to the Java implementation and to the Java-to-Axis binding. Additionally,
HTML files can be produced that link every service to its interfaces, bindings and data types alongside with the documentation. This helps to keep track of the developed services and to quickly find the right documentation.

By extracting the necessary information from the abstract interface and service definitions, the Venice Service Compiler abstracts from the underlying Web service technology, it only uses pre-defined XML data types and generates the rest to make the service developers focus on implementing the services’ functionalities. This way, the service definition and implementation remain the same even when the underlying technology changes, e.g. when switching from WSDL 1.1 to WSDL 2.0 or from Axis 1 to Axis 2. Then only the compiler has to be changed or extended. By separating the implementation from actual Web service technology, it is also possible to regenerate and recompile the Java classes without the additional debugging, logging and performance measurement code for production releases.

5. Conclusion and Outlook

The Venice Service Grid is an open, dependable and secure framework for deploying Web services on the Internet – secure and reliable access to the services is managed automatically. A combination of an abstraction layer and a service infrastructure make it easy for architects to define new services, for developers to implement new services, for administrators to deploy new services and for stakeholders to manage and observe their services. As part of the abstraction layer, the Venice Service Compiler is a powerful tool for developers. It generates WSDL files from an abstract interface definition, Java code for the actual implementation as well as Axis bindings and deployment descriptors. Additional code can be automatically inserted for testing WSDL compliance, statistics, managing state, error handling, logging and for performance measurements.

In the future, the Venice Service Compiler will be expanded to support Axis 2 and WSDL 2.0. It is also intended to equip the compiler with a graphical front-end – currently only a command line tool is available. This graphical front-end could also be used to manage the repository of services and WSDL files as well.
7. References


Links


