Bridging the Gap between Legacy Services and Web Services

Tegawendé F. Bissyandé, Laurent Réveillère, Yérom-David Bromberg, Julia L. Lawall, Gilles Muller

Bridging the Gap between Legacy Services and Web Services

Tegawendé F. Bissyandé[1], Laurent Réveillère[1], Yérom-David Bromberg[1]
Julia L. Lawall[2,3], and Gilles Muller[3]

[1] LaBRI, University of Bordeaux, France
[2] DIKU, University of Copenhagen, Denmark
[3] INRIA/Lip6, France

Abstract. Web Services is an increasingly used instantiation of Service-Oriented Architectures (SOA) that relies on standard Internet protocols to produce services that are highly interoperable. Other types of services, relying on legacy application layer protocols, however, cannot be composed directly. A promising solution is to implement wrappers to translate between the application layer protocols and the WS protocol. Doing so manually, however, requires a high level of expertise, in the relevant application layer protocols, in low-level network and system programming, and in the Web Service paradigm itself.
In this paper, we introduce a generative language based approach for constructing wrappers to facilitate the migration of legacy service functionalities to Web Services. To this end, we have designed the Janus domain-specific language, which provides developers with a high-level way to describe the operations that are required to encapsulate legacy service functionalities. We have successfully used Janus to develop a number of wrappers, including wrappers for IMAP and SMTP servers, for a RTSP-compliant media server and for UPnP service discovery. Preliminary experiments show that Janus-based WS wrappers have performance comparable to manually written wrappers.

1 Introduction

The Web Services (WS) instantiation of Service-Oriented Architectures has progressively been adopted as a practical means to implement distributed applications [18]. WS exploit the pervasive infrastructure of the World Wide Web to set up loosely coupled software systems composed of a collection of services. Services rely on a set of standards and specifications[4] to make their functionalities available according to platform-independent interfaces, facilitating the construction of heterogeneous compositions.

Many services, however, continue to rely on legacy application layer protocols (ALPs). Examples of such protocols include IMAP for retrieving mail, SMTP for sending mail, RTSP for controlling media streaming, and UPnP for discovering networked home appliances. These protocols are considered to be reliable and effective, but complicate service composition. While WS can easily and safely be combined using widely used standards, such as WS-BPEL [16] and WS-CDL [23], composing ALP-based services requires integrating a protocol stack for each ALP in the client application.

[4] A specification is a potential standard that has not yet been approved.

To provide ALP-based services with a uniform interface, to allow them to be more easily combined to provide rich functionalities, one solution is to use wrappers to convert them to Web Services. A wrapper is essentially a gateway that provides a WS interface to the existing capabilities of an ALP-based service. It makes accessible, through appropriate operations, the independent functionalities that the service provides, without the complexity of reimplementing the service as a WS. Nevertheless, this approach requires translating WS requests into ALP requests and ALP responses into WS responses. Implementing these translations safely and efficiently involves challenging programming at both the network and systems level.

At the network level, the wrapper programmer must take into account the variety of ALP definitions. For example, some ALPs are symmetric, relying on request-response communication, while others are asymmetric, relying on message-based communication. An ALP may also support sessions or reliability, which must then be accounted for at the WS level. WS are normally unicast; wrapping an ALP-based service relying on a multicast ALP requires using UDP rather than HTTP, and a specific set of WS standards. Finally, in practice, to ease the development of a WS client and improve efficiency, it may be desirable to create a single WS operation that corresponds to a series of ALP requests and responses.

At the systems level, expertise in thread, memory and socket management is necessary to efficiently handle simultaneous requests, to dispatch responses to appropriate endpoints and to keep track of established sessions. Furthermore, in order to avoid requiring a wrapper to actively wait for asynchronous responses, the execution of a request handler must be stopped until corresponding responses arrive, then restarted to process results. These processing tasks must not prevent the wrapper from handling other synchronous and asynchronous requests. The complexity of such programming tasks makes manual wrapper construction laborious and error prone. Naive implementations of such code can introduce severe performance bottlenecks.

This Paper. In this paper we propose a generative language-based approach for constructing wrappers to enable the migration of legacy service functionalities to Web Services. This approach involves two domain-specific languages: z2z, which was developed in our previous work [2] for describing ALP message structures and behaviors, and Janus, which is the main contribution of this work and targets the specific needs of WS. Our approach targets programmers who are familiar with an ALP and with basic Java programming. Its main benefit is to allow such developers to quickly and easily develop efficient and safe WS wrappers. Our contributions are:

- We define the Janus domain-specific language that allows describing the interface of a legacy service and its representation as a WS. A Janus description is expressed at a high level that hides the low-level details of the WS paradigm and of the ALPs.
- We describe the translation of a Janus description into a wrapper implementation compatible with a WS environment, and the Janus runtime system that supports the execution of this wrapper. The translation and the runtime system together address various network and systems programming issues, hiding this complexity from the programmer.
- We show that the expressiveness of Janus is sufficient to describe the interface of a number of ALP-based services and to generate the appropriate WS wrappers. Our

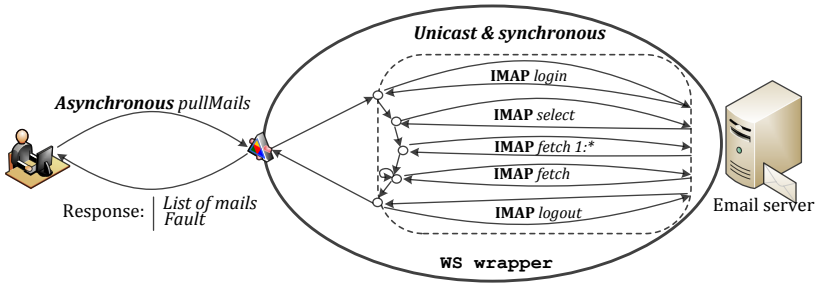

Fig. 1. IMAP service wrapper

case studies include well-known legacy services relying on ALPs such as IMAP, SMTP, RTSP and UPnP.
- The experiments that we have carried out show that our approach produces wrappers that have performance comparable to manually developed wrappers based on existing WS and ALP stacks.

The rest of this paper is organized as follows. Section 2 introduces the case studies that we use to present the details of our approach and the issues that these case studies entail. Section 3 presents the Janus language and Section 4 describes the generated code. Section 5 demonstrates the efficiency of our approach. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2 Case Studies

A developer creating a WS wrapper for an ALP-based service by hand must first select the functionalities that should be made available as WS operations and then describe for each operation the corresponding WS interface and the structure of the operation's parameters and results. We present some of the issues confronting the developer in creating such wrappers, and illustrate these issues using wrappers for IMAP and SMTP mail services, for an RTSP-compliant media service and for UPnP service discovery.

Message granularity. ALPs are generally implemented directly on top of TCP, resulting in lightweight messages, and thus are able to provide fine-grained functionalities. WS, on the other hand, are built on top of SOAP and either HTTP or UDP, resulting in messages that are complex and verbose. Thus, in a WS environment, to reduce the bandwidth consumption and to simplify the client implementation, it is often desirable to provide higher-level operations. As an example of this granularity mismatch, we consider a WS wrapper for an IMAP server, illustrated in Figure 1. The IMAP server shown inside the oval on the right side of the figure allows a client to retrieve mail using a sequence of synchronous exchanges of messages, for authentication, folder selection, message listing, message fetching, etc. The WS wrapper, however, encapsulates this sequence of low-level IMAP requests and responses as a single WS operation, *pullMails*. A WS wrapper for an SMTP server could be constructed similarly.

Message transmission synchrony. In the implementation of our IMAP wrapper, we have chosen to make the WS operation asynchronous, even though the protocol used by the service is synchronous. The management of this asynchrony must be implemented by the wrapper developer.

Message return values. The result of the WS operation is the list of mails retrieved from the server in case of success, or an error message otherwise. The wrapper programmer has to be aware of the specific data types to be used in constructing WS error messages.

Session management. Our second case study involves the construction of a session-based WS wrapper to remotely control a RTSP-compliant media server. Although RTSP requests flow within different TCP streams, some requests need to be associated to the same session. For instance, the *play* and *stop* requests include session information. The wrapper developer must thus translate RTSP session management into WS session management, through the use, for instance, of the WS-reliability specification.

Multiple ALPs. The media server case study also illustrates the case of a wrapper that needs to process messages from several different ALPs. For instance, the media service wrapper may need to process SDP messages describing multimedia session information that are encapsulated in the body of RTSP responses.

Multicast. In a networked environment, UPnP-enabled clients discover the services provided by available UPnP-compliant devices. To successfully discover existing services, clients have to send UPnP search requests to a multicast group address. However, supporting the multicast communication paradigm in the WS realm requires the use of several WS specifications that are not part of the basic WS standards. Our implementation of the WS wrapper for UPnP relies on the specifications SOAP-Over-UDP [17] and WS-Addressing [22]. The development of this wrapper is significantly different from other traditional wrappers because SOAP messages are not encapsulated inside HTTP messages but flow directly over UDP. Therefore, constructing such multicast wrappers significantly raises the level of expertise required by the wrapper developer.

3 Wrapper Development

A WS wrapper converts a WS invocation into a sequence of ALP interactions, and then converts the information collected by these ALP interactions into a WS result. Constructing such a wrapper requires information about the ALP behavior (*e.g.*, whether messages are transmitted by unicast or multicast, synchronously or asynchronously, etc.), the structure of the WS and ALP messages, and the logic for translating between them.

In previous work, we have developed the z2z language for constructing network protocol gateways [2]. A WS wrapper can be seen as a particular kind of gateway, dedicated to the specific needs of WS. Z2z provides facilities for describing network protocol behaviors, message structures, and translation logics, and an optimized run-time system. It is suitable for expressing the behaviors and message structures of protocols that are

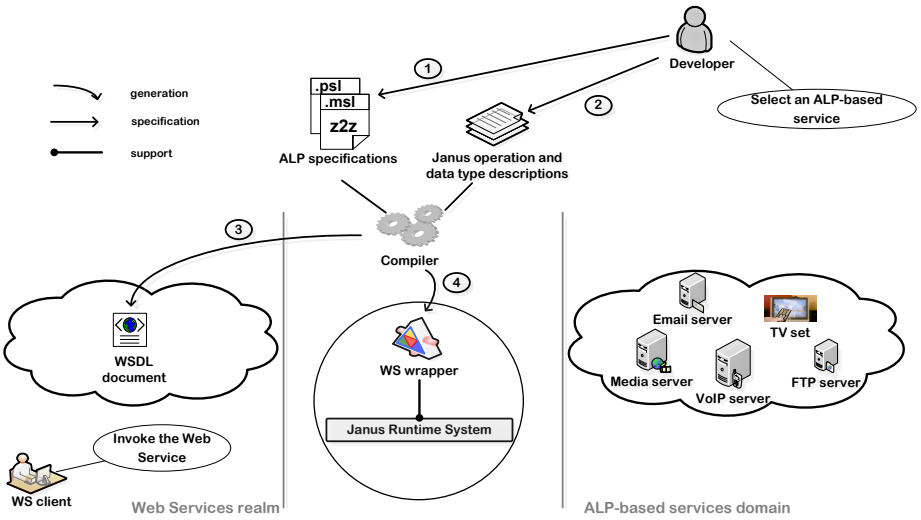

Fig. 2. Scenario for constructing wrappers with z2z and Janus

built directly on the transport protocols TCP and UDP, which is typically the case of ALPs. WS, however, are at a higher level, being built on SOAP [24], which in turn is built on HTTP or directly on top of UDP in the multicast case. Defining WS messages and the WS-ALP translation logic using z2z would require expressing these features in terms of SOAP/HTTP or SOAP/UDP messages, which would be extremely tedious and require a high degree of WS expertise. We have thus developed a new language, Janus, for describing WS messages and the translation between WS and ALP messages directly, and a supporting runtime system that translates WS interactions to the lower level SOAP and HTTP protocols. To fit with the expertise of the expected developer community, Janus uses a Java-like syntax.

Based on z2z and Janus, we propose a generative language-based approach to WS wrapper construction that relies on z2z for describing ALP behaviors and message structures, and Janus for describing WS message structures and the translation between WS messages and ALPs. Fig. 2 gives an overview of this approach. For each ALP relevant to the functionalities that the developer has chosen to expose as WS operations, the developer provides a z2z specification, consisting of a *protocol specification*, describing how the ALP interacts with the network, and a *message specification*, describing the structure of ALP requests and responses.[5] The developer then uses Janus to describe the desired WS interface to these functionalities, including the structure of the WS operation arguments and return values and the translation of each WS operation to the corresponding ALP messages. The Janus compiler translates the z2z and Janus specifications to an executable wrapper and a WSDL document that describes the

[5] As shown in Fig. 2, a protocol specification is provided in a `.psl` file and a message specification in a `.msl` file.

```
1  protocol rtsp {
2    int cseq_number;
3    attributes { transport = tcp/554; mode = sync; }
4    start { cseq_number = 1; }
5    request req {
6      response DESCRIBE when req.method == "DESCRIBE";
7      response PLAY     when req.method == "PLAY";
8      ...
9    }
10   sending request req { req.cseq = cseq_number++; }
11   flow = { cseq }
12   tcp { void tcp_connect(); }
13 }
```

a) RTSP protocol specification

```
1  read {
2    mandatory public fragment code;
3    mandatory public fragment line;
4  }
```

b) IMAP request message view

```
1  request template response getMail {
2      magic = "SEP";
3      newline ="\r\n";
4      public int id;
5      private int tag;
6  --SEP
7  <%tag%> fetch <%id%> body[text]
8  --SEP
9  }
```

c) IMAP request template

Fig. 3. Z2z protocol behavior and message structure descriptions

generated WS. The wrapper is then linked with a runtime system that provides various optimized systems functionalities.

In the rest of this section, we present the use of z2z for describing ALPs and the Janus language for describing WS messages and operations.

3.1 Z2z protocol behavior and message structure descriptions

The first step of our approach uses z2z to describe how the relevant ALPs interact with the network, as illustrated for the RTSP protocol in Fig. 3a. A z2z protocol specification first declares any needed local variables, such as `cseq_number` in line 2, and then contains a collection of blocks describing various properties of the interaction with the network. The `attributes` block specifies the transport protocol used, whether requests are sent in unicast or multicast, and whether ALP responses are received synchronously or asynchronously (line 3). The `start` block initializes the local variables (line 4). The `request` block specifies how to dispatch a received request to a specific handler for processing (lines 5-9). The `sending` block specifies some default information for each request or response, such as the `cseq_number` for an RTSP message (line 10). The `flow` block indicates the message information that a wrapper must use to match asynchronous requests to their subsequent responses (line 11). A similar `session_flow` block is used to recognize messages that are associated with a particular session, when the protocol supports sessions. Finally, the `tcp` block specifies a handler for opening connections on a socket (line 12).

A WS wrapper must also be aware of the structure of ALP messages. These messages are also described using z2z. A z2z *message view* defines the information to be extracted from incoming messages (Fig. 3b). Similarly, z2z *templates* (Fig. 3c) describe the structure of new ALP messages to be created by the wrapper. Both message views and templates may contain fields declared as `private` that are handled automatically by the runtime system and fields declared as `public` that must be managed by the Janus message translation logic. For example, the `tag` field is declared as `private`

in Figure 3c (line 5) because its value is automatically generated for each constructed IMAP request message. This is also the case of the `cseq` field for RTSP, which, as illustrated in Figure 3a (line 10), is filled in by the `sending` block of the RTSP protocol specification. Whenever a RTSP request is sent, the value of the `cseq` field is automatically incremented by the runtime system.

3.2 Janus service operation descriptions

The second step of our approach uses the Janus domain-specific language to describe how to invoke the chosen functionalities of the ALP-based service. Janus has been designed according to requirements that we have identified as critical to ease WS development and to enforce good practices in WS design. For instance, Janus follows a *contract-first strategy* in the implementation of a WS. That is, an abstract description of a WS (e.g., WSDL) is made available before the actual production of the WS. By design, Janus *supports stateful Web Services* by enabling side-effects. While presenting a Java-like syntax (see Fig. 4), with which programmers are familiar, Janus *limits the functionalities to what is needed for WS development*. The only objects available are data structures which only have a default constructor, for initializing their different fields, and a default method (`send`), for forwarding them in the network. Finally, Janus enables the creation of robust wrappers by *encapsulating subtle and error-prone code* such as code for network message processing. For example, it provides the `send` operator for sending messages on the network and the structure field notation for easily accessing message fields. These features are provided within a language, rather than in a library as done in Java, allowing the complete code to be checked for various coherence properties. For example, the Janus compiler checks that the code is *type safe*, that all variables are *initialized before they are used*, and that all message fields are *initialized before the message is sent*.

In Janus, the wrapper functionalities are described in a service definition. As shown in Fig. 5, a Janus service is defined by the keyword `service` (line 3), followed by the name of the service being defined. The service is parametrized by the hostname and port number of the machine that hosts the WS wrapper. These values are set when invoking the Janus compiler. A Janus service defines data types to describe the parameters and return value of a WS operation, and a set of methods to specify the series of ALP messages that need to be exchanged with the service to define each WS operation. The ALP attributes previously defined with z2z are imported through the `#import` directive at the beginning of the file. Other utilities' interfaces can also be referred to using this directive.

Data types A WS operation typically has some arguments and return values, of various data types. The wrapper must know the structure of these data types so that it can extract information from the arguments in order to construct the corresponding ALP requests, and so that it can construct the return values from the ALP responses. Janus data types are either primitive or complex. Primitive types are strings and integers. A complex type is defined by a Java-like class containing only fields. Such a class also defines an implicit constructor that takes as arguments the initial values of the fields in the order

```
program              ::= external_import* service_def
external_import      ::= #import interface_id ;
                       | #import protocol.protocol_id ;
service_def          ::= [qualifier] service (service_params) { datatype_def* operation+ }
service_params       ::= COMMA_LIST(primitive_type datatype_id)
datatype_def         ::= class complex_type_id [extends complex_type_id] { nested_data* }
nested_data          ::= primitive_type COMMA_LIST(var) ;
operation            ::= datatype operation_id ([operation_params]) { statement+ }
datatype             ::= primitive_type | complex_type
                       | List<datatype>
operation_params     ::= COMMA_LIST(datatype datatype_id)
primitive_type       ::= String | int
complex_type         ::= complex_type_id
qualifier            ::= multicast
statement            ::= decl_stmt | affect_stmt | if_stmt | for_stmt | except_stmt | return_stmt | {statement+}
decl_stmt            ::= datatype COMMA_LIST(var) ;
                       | request<protocol_id> COMMA_LIST(var) ;
                       | response<protocol_id> COMMA_LIST(var) ;
affect_stmt          ::= var = data ; | datatype var = data ;
data                 ::= new complex_type ([COMMA_LIST(primitive_type_id)])
                       | data.field | data.send() | function (data)
if_stmt              ::= if (boolean_expr) statement+
for_stmt             ::= for (datatype var : list_var) { statement* }
except_stmt          ::= throw data ;
return_stmt          ::= return data ;
boolean_expr         ::= data | boolean
COMMA_LIST(elem)     ::= elem (, elem)*
```

Fig. 4. Janus language grammar

```
1  import protocol.rtsp.*;
2
3  service mediaPlayer (String hostname, int port) {
4    /* Data type definitions */
5    class MediaRequest { String resource; }
6    class PlayRequest extends MediaRequest { ... }
7    class PauseRequest extends MediaRequest { ... }
8    class StopRequest extends MediaRequest { ... }
9
10   ...
11   /* Operation descriptions */
12   Media PLAY (PlayRequest req) { ... }
13   ...
14 }
```

Fig. 5. Janus service for RTSP Media service

in which they appear in the class definition. Janus provides an inheritance mechanism that enables one data type to be defined as an extension of another. This is useful when data types share a number of fields, as in the case of the invocation parameters of the `PLAY`, `PAUSE`, and `STOP` operations defined by the media service wrapper (Fig. 5). For each of these operations, the parameter includes a `Resource` field that defines the URI of the media being served. Therefore, the corresponding Janus classes extend the `MediaRequest` class that contains this `Resource` field.

Operation descriptions A WS operation is described in Janus as a method whose arguments and return values correspond to the input and output parameters of the WS

```
1  List<Mail> pullMails(String login, String passwd, String folder) {
2    /* operation pullMails retrieves unread mails from an IMAP server */
3    request<imap> req;
4    response<imap> resp;
5    List<Mail> mails = new List<Mail>();
6    List<int> ids = new List<int>();
7    Mail m;
8
9    req = new Login(login, passwd); resp = req.send();
10   if (resp.code == "error")
11     throw new ServiceFault("[login]", "server failed");
12
13   req = new selectFolder(folder); resp = req.send(); ...
14   req = new listMessage(); resp = req.send(); ...
15
16   ids = List.parse2int (resp.line, " ");
17   for (int id : ids) {
18     req = new getMail(id); resp = req.send(); ...
19     m = new Mail(id, resp.line);
20     List.add (m, mails);
21   }
22
23   req = new Logout(); resp = req.send(); ...
24   return mails;
25 }
```

Fig. 6. IMAP `pullMails` service operation

operation. The main function of such a method is to translate between WS and ALP messages. Nevertheless, Janus also provides abstractions to support sessions and multicast services. Using the example of the *pullMails* operation defined in Fig. 6 for our IMAP server case study, we illustrate how the interface to a functionality of an ALP-based service is expressed using Janus.

A Janus method exchanges a sequence of ALP messages with a service in order to provide the requested functionality to the WS client. To create an ALP message, the Janus code uses the constructor implicitly associated with the corresponding z2z template (line 9). This constructor takes as arguments the values for the template's `public` fields in the order in which they appear in the template definition. A template also provides a method `send` for sending a created message into the network (line 9). The Janus compiler translates a use of the `send` method into an invocation of the z2z `send` operator. This operator transparently handles the difference between sends with synchronous and asynchronous responses, freeing the developer from the need to manage this complexity. To extract information from an ALP response, the Janus code uses the standard field access notation (line 10), as in Java. Any field that is qualified as `public` in the corresponding z2z message view is accessible in this manner.

As in Java, the `return` keyword indicates the value returned by a method to its caller (line 24). In Janus, the returned value of a method is represented by a complex data type and must be created by the Janus code using the data type's associated constructor. To send the returned value back to the WS client, the Janus compiler generates code to serialize and encapsulate this value as a WS compliant message.

Janus also supports a mechanism for error management. For example, if the login to the IMAP server fails (line 10), a fault message has to be sent back to the WS client.

```
1  import protocol.rtsp.*;
2
3  service mediaPlayer (String hostname, int port) {
4    String SESSION_ID;
5    ...
6    Media PLAY (PlayRequest preq) {
7      ...
8      req = new Setup(...); resp = req.send(); ...
9      /* Save the session ID returned by setup */
10     SESSION_ID = resp.sessionId;
11     ...
12   }
13   Media STOP (StopRequest sreq) {
14     ...
15     /* Use the session ID previously saved */
16     req = new Teardown(hostname, sreq.resource,
17                                          SESSION_ID);
18     ...
19   }
20   ...
21 }
```

a) Excerpt of the RTSP PLAY service operation

```
1  import protocol.ssdp.*;
2
3  multicast service controlPoint () {
4   class UPnPService { ... }
5   ...
6   List<UPnPService> SEARCH(SearchRequest sreq){
7     ...
8   }
9   ...
10 }
```

b) Excerpt of the UPnP control point wrapper description

Fig. 7. Janus descriptions

As in Java, a Janus exception is raised using the keyword `throw` (line 11), aborting the method execution. Unlike in Java, Janus exceptions cannot be caught by the programmer and are only used to report unexpected situations to the WS client. A fault message can be created using the constructor of the default `ServiceFault` data type (line 11) or of a defined data type that extends this one.

When an ALP uses sessions and the requests within a session are associated with different WS operations, then the WS wrapper must manage sessions as well. For example, as described in Section 2, the media service wrapper needs to manage a session that has been set up by the media service. As shown in the Janus implementation in Fig. 7a, to process a WS `STOP` operation, an ALP *Teardown* request must be sent with the session information (line 16) that was previously returned by the ALP *Setup* request (line 10). However, these ALP requests are sent within different WS operations. Janus implements sessions using the WS-Reliability [15] specification. The Janus code can then declare global variables that are visible within a session. For example, line 4 of Fig. 7a declares a global variable `SESSION_ID` that maintains the ALP session identifier across multiple WS requests. Such a variable can be set (line 10) and read (line 16) like a local variable. The Janus compiler automatically generates the code to manage session information in the WS realm.

In the excerpt of Fig. 7b, the Janus description of the wrapper for UPnP service discovery is declared with the keyword *multicast* (line 3), indicating that the implementation must be multicast-compliant. Janus then produces a wrapper that can process SOAP messages carried by UDP instead of HTTP. All information that is carried by HTTP in the unicast case, including the client endpoint reference to which WS responses must be returned, is now encapsulated in the SOAP message using the WS-Addressing specification.

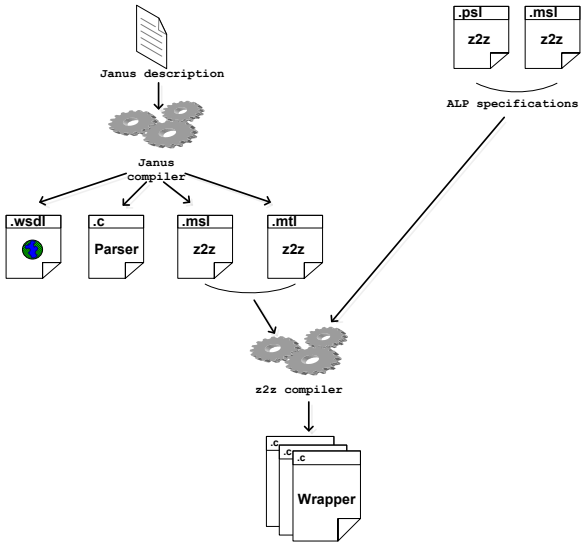

Fig. 8. Overview of the generated code

4 Code Generation

Based on the z2z descriptions of ALP behaviors and message structures and the Janus descriptions of WS message structures and the translation between WS messages and ALPs, the Janus compiler generates various documents, specifications and program code, as shown in Fig. 8, to create a complete wrapper implementation. In this section we describe the generation of these artifacts.

4.1 WS framework

From a WS wrapper specification written in Janus, the compiler generates code for publishing the service operations in a WS framework and for processing WS messages.

Publishing the service operations. In the WS framework, a WS is accompanied by a WSDL document that makes information about the operations provided by the WS available in a machine-readable form. A WSDL document also specifies concrete bindings that describe how the abstract service description is mapped to a specific service access protocol. Nevertheless, WSDL, as a machine-readable format, is not well suited to being written by hand, especially for a service that defines multiple operations.

Based on the Janus service operation descriptions and compiler arguments indicating the endpoint where the service is to be deployed, the Janus compiler creates the WSDL description of the wrapper. The generated WSDL document includes *types*, which are data type definitions specified using the XML Schema language, *messages*, which are typed definitions of the data to communicate, and *operations*, which are abstract descriptions of the actions supported by the service. Furthermore, the WSDL

```
1  message soap {
2    read {
3      mandatory public fragment subject;
4      mandatory public fragment from;
5      ...
6      mandatory public int smtpPort;
7    }
8    ...
9  }
```

a) message view for a SMTP message

```
1  #include "msg_soap.h"
2  ...
3  void xml_data(void * d, const char * data, int l) {
4      struct IGDdatas * datas = (struct IGDdatas *)d;
5      char buf[l];
6      if ( !z2z_strcmp(datas->elt_name, "subject") ){
7          sprintf(buf, "%.*s",l, data);
8          msg_soap_view_set_subject(datas->view,
9                                    make_string(buf));
10     }
11     ...
12 }
```

b) Excerpt of the SOAP parser for SMTP

Fig. 9. Generation of a SOAP message view and the associated parser

```
1  import protocol.imap.*;
2  service ImapServer(String hostname, int port) {
3   ...
4   serverResponse imapCreateFolder(String login,
5                 String passwd, String folderName) {
6    response<imap> resp; request<imap> req;
7    ...
8    req = new createFolder(folderName);
9    resp = req.send();
10   ...
11   return new serverResponse(resp.line);
12  }
13 }
```

a) imapCreateFolder operation with Janus

```
1  template createFolderResponse {
2   magic = "SEP"; newline ="\r\n";
3   public fragment serverResponseLine;
4  --SEP
5  <soapenv:Envelope ...>
6   ...
7    <cli:createFolderResponse>
8      <resp><%serverResponseLine%></resp>
9    </cli:createFolderResponse>
10  ...
11 </soapenv:Envelope>
12 --SEP
13 }
```

b) z2z generated template

Fig. 10. Message template generation

document specifies the endpoint address where the service is available as well as the protocol (e.g., SOAP) to be used for invoking the service. The Janus compiler also includes in the WSDL document the WS specifications that are required by the wrapper. Once created, the WSDL document is made available via a web server, thus allowing client programs to call any of the operations that are listed.

WS message structures and processors. Once the wrapper is exposed to potential WS clients, it may begin to receive WS messages. It must parse these messages and may need to construct WS messages to send in response. The structure of these messages is determined by the parameter and return type specifications in the signatures of the Janus operation descriptions and the associated Janus data type specifications. To provide support for processing received WS messages, the Janus compiler generates a z2z message view and a dedicated SOAP parser from the data types representing the parameters of each WS operation. The message view contains all the fields of the data type, including those of any data type it inherits. The SOAP parser is a C program that extracts invocation parameter values from an XML document, embedded inside an incoming HTTP message or on top of UDP, and uses these values to initialize the message view fields listed in Fig. 9 (a). Fig. 9 (b) shows an excerpt of a generated parser that recovers the

subject to use in a *sendMail* operation from a WS message. For each data type that is used to describe a WS operation return value, as in the example of Fig.10a (line 11), the Janus compiler generates a z2z SOAP template whose public fields correspond to the primitive types that compose the return data type (Fig.10b). This template is used to create a SOAP message that carries the operation result.

4.2 Wrapper implementation

The wrapper implementation is based on the z2z specifications of protocol behaviors and message structures, and the Janus operation descriptions. These are translated by the Janus compiler into the corresponding lower level z2z message translation logic code, which is then translated into C code by the z2z compiler. Furthermore, the Janus compiler adds into the translation logic of the operation descriptions the code necessary for taking into account any WS specifications that are used to address issues that are supported by Janus, but are not handled by the basic WS standards. Developers need not be aware of the details of these specifications.

To support client sessions, Janus relies on the WS-Reliability [15] specification. Using this specification, a WS message is identified by a message ID, consisting of a group ID and a sequence number. In our design, a WS wrapper recognizes messages from the same session by the shared message group ID. Based on this information, the wrapper code has access to the values of the global variables corresponding to the session, which it can then use to construct ALP messages containing appropriate session identifiers. The Janus compiler automatically detects the use of ALP sessions in the operation descriptions and generates the corresponding code to manage WS sessions.

When a Janus service is declared as `multicast`, the Janus compiler generates code to parse and create SOAP messages directly over UDP. This generated wrapper code uses the SOAP-over-UDP [17] and WS-Addressing [22] specifications. The wrapper listens to a multicast group address and does not include HTTP processing capabilities.

The C code generated by the composition of the Janus and z2z compilers is supported by a dedicated runtime system. Janus relies on a enhanced version of the z2z runtime system that provides a framework for processing SOAP messages and currently supports the WS specifications WS-Reliability, SOAP-over-UDP and WS-Addressing.

5 Assessment

To assess our approach, we have implemented wrappers for the various ALP-based service functionalities described in the case studies of Section 2. For each considered case study, Figure 11 compares how many lines of source code the developer needs to manually write against how many lines of source code are generated by the Janus compiler. Among our examples, only 204 (SMTP) to 582 (RTSP) lines of source code, including ALP specifications (z2z), operation descriptions (Janus) and ALP parsers (C), need to be written by hand to implement the WS wrapper. Using only z2z, but not Janus, the WS wrapper for SMTP requires 642 lines of source code, including SOAP parsers, WSDL documents, and z2z specifications, and the WS wrapper for RTSP requires 882

		Developer code (lines of code)			**Generated code**						
		ALP specifications	Janus descriptions	ALP Parsers	z2z specifications (lines of z2z code)	SOAP parsers (lines of C code)	WSDL document (lines of code)	Wrapper source code (lines of C code)	WS wrapper (size in KB)		
									Wrapper	Runtime System	Total
IMAP Server wrapper	IMAP	161	79	102	370	208	102	1861	44	80	124
SMTP Server wrapper	SMTP	129	48	27	216	222	75	918	24	80	104
Media Server wrapper	RTSP SDP	193 41	97	153 98	297	227	122	2488	48	80	128
UPnP service discovery	UPnP	58	13	304	115	312	113	1877	32	80	112

Fig. 11. The size of specifications and the generated WS wrapper

lines of source code. This comparison of code size furthermore does not fully take into account the amount of WS and network expertise that is required to implement a wrapper without using Janus. Compared to the final generated C code, excluding the runtime system, the Janus compiler provides around 77% of the code. Moreover, as illustrated in Figure 11, the size of the wrappers does not exceed 128KB, including 80KB for the runtime system. Thus, Janus wrappers can be embedded in constrained devices.

To fully evaluate the performance of Janus generated wrappers, we have carried out three experiments involving an IMAP and an SMTP service, a UPnP service, and an ALP-based echo protocol. Our experiments were carried out on a 2GHz Intel Core 2 duo with 4GB of RAM. In each case, to reduce the impact of the network latency on the response time, the client, the wrapper, and the service are all collocated on the same machine and interact using the loopback interface.

WS wrappers for IMAP and SMTP services. We evaluate the capacity of wrappers to manage both WS to IMAP and WS to SMTP translations under stress tests. To this end, several WS clients simultaneously invoke either a WS *pullMails* operation on a user folder to retrieve two mails, or a WS *sendMail* operation to send five mails to a remote mailbox. The IMAP wrapper translates WS invocations into IMAP messages, that are sent to a Dovecot IMAP server (*http://www.dovecot.org*) The SMTP wrapper similarly translates WS invocations into SMTP messages that are sent to a POSTFIX SMTP server (*http://www.postfix.org*).

Figures 12 and 13 compare, respectively, the performance of the Janus IMAP wrapper with that of a manually developed IMAP wrapper, and the Janus SMTP wrapper with that of a manually developed SMTP wrapper. The handmade wrappers, using neither Janus nor z2z, are implemented in Java with the JAX-WS Reference implementation and Tomcat. The execution time is measured from the time when the wrapper receives a WS invocation to the time when the corresponding response is sent back to the WS client. The performance is expressed in terms of CPU cycles, to be independent of the CPU frequency.

In our experiments, two parameters may impact the performance: (i) the number of simultaneous clients, and (ii) the number of simultaneous invocations performed by each client. Consequently, our test procedure involves several simultaneous clients and consists of a set of rounds, successively increasing the stress on the wrapper in each round. In the first round each client fetches or sends all mails once, resulting in two

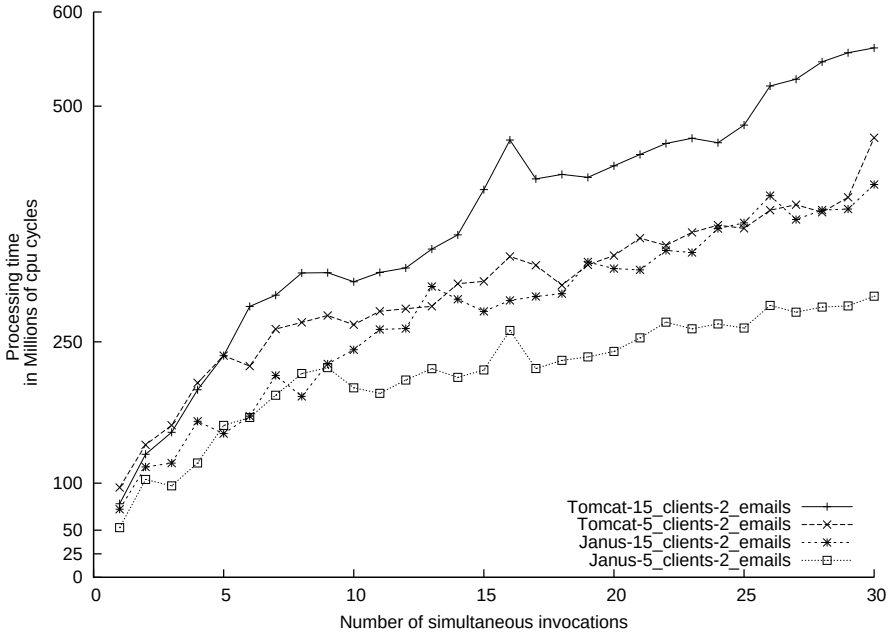

Fig. 12. IMAP server wrapper

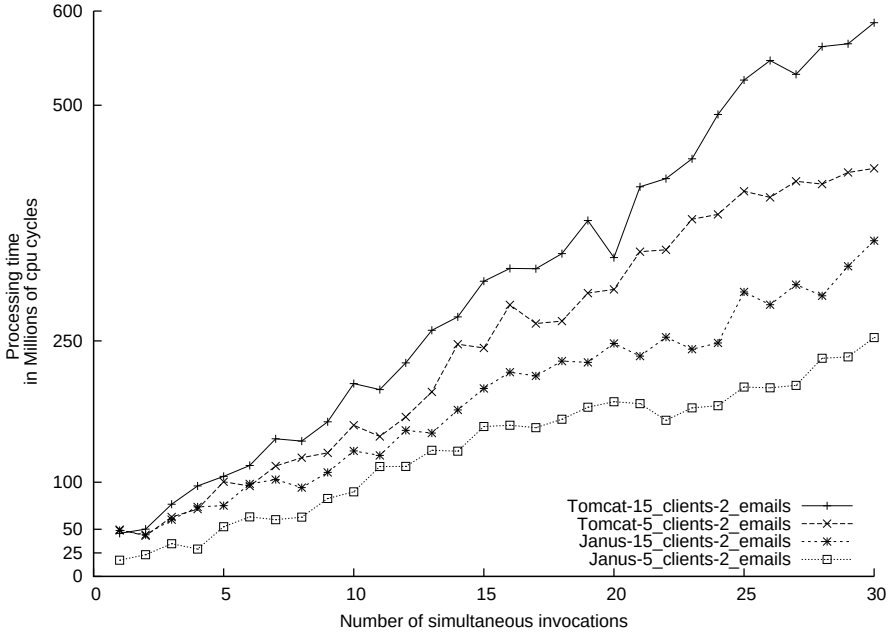

Fig. 13. SMTP server wrapper

or five simultaneous requests per client for the IMAP or SMTP tests respectively. In the second round each client fetches or sends all mails twice, resulting in four or ten simultaneous requests per client. This pattern continues until the thirtieth round where each client fetches or sends all mails 30 times, resulting in 60 or 150 simultaneous requests per client. The test procedure is undertaken 30 times. Figures 12 and 13 show only median values.

As expected, the graphs show that the higher the number of simultaneous clients (*i.e.*, 5 or 15 clients for IMAP and SMTP wrappers), the higher the response times. Janus wrappers perform better than the wrappers developed by hand, because the Janus wrappers can rely on a fine grained runtime support that includes generated code that is dedicated to mapping IMAP and SMTP messages into SOAP messages and vice versa. Specifically, Janus wrappers include a WS stack stripped down to the bare essentials according to the Janus description given to the Janus compiler. In contrast, handmade wrappers use a general-purpose WS stack and runtime that offer no particular optimizations for wrappers.

WS wrapper for UPnP service. The Janus-generated wrapper for UPnP service relies on multicast addressing both in the WS realm, requiring support of extra WS-* standards, and in the UPnP native domain. We have therefore carried out an experiment to estimate the overhead introduced by the wrapper processing layers. To evaluate the Janus UPnP wrapper, we compare the response time required by a WS client to discover a UPnP service with the time required for a UPnP client to discover a UPnP service. In both cases, we measure, at the client side, the time taken between an initial discovery request and the corresponding successful response. Any standard WS toolkit can be used to generate the WS client from the WSDL Document published by with Janus wrapper. We have chosen the gSOAP[6] toolkit for its efficiency, as it is developed in C. The UPnP client and the service are developed with the C implementation of the CyberLink[7] stack.

In our experiment, the response time of the native UPnP client reaches 1220 million CPU cycles whereas for the WS wrapper client it takes 1805 million CPU cycles, amounting to a slowdown of 50%. This slowdown results from the cost of the various steps of the translation logic. The Janus wrapper needs to: (i) listen on a multicast group address, dedicated to the WS realm, to intercept incoming SOAP-over-UDP requests, (ii) deserialize received SOAP requests to generate the corresponding UPnP requests, (iii) forward the newly generated UPnP requests and listen for potential UPnP responses from the multicast group address dedicated to UPnP, and (iv) transform UPnP responses to SOAP messages to send them back to the WS client. In comparison, the UPnP client interacts directly with the UPnP service, it does not need to listen on two different multicast group addresses, and it does not need to serialize and deserialize SOAP messages.

WS wrapper for an ALP-based echo service. We have implemented a micro benchmark to evaluate the performance of the SOAP serialization/deserialization performed by Janus wrappers. The experiment involves a Janus wrapper for an ALP-based echo service that echoes primitive data types such as integers and strings. In this case, a

[6] http://www.cs.fsu.edu/~engelen/soap.html
[7] http://www.cybergarage.org/cgi-bin/twiki/view/Main/CyberLinkForC

WS client, generated by the gSOAP compiler from the Janus WSDL document, sends SOAP primitive data types to a Janus wrapper that extracts the data value to forward it, without any XML tags, to an echo service. Messages from the echo service are similarly encapsulated into SOAP messages by the Janus wrapper and are sent back to the WS client. The micro benchmark measures the execution time from when the wrapper receives a WS invocation to when the corresponding response is sent back to the WS client. We consider the median time over 50 executions. We find that with serialization and deserialization, this takes around 1.1 million CPU cycles for an integer value, and around 1.4 million CPU cycles for a string of 50 characters. Without serialization and deserialization, we find that the time is around 0.3 million CPU cycles for an integer value and around 0.5 million CPU cycles for a string of 50 characters. Although these results show that the cost of serialization is high, they represent a worst case due to the simplicity of the echo server. Normally the total treatment time would be dominated by the server computations.

6 Related Work

Alternative forms of Web Services This paper has focused on the SOAP/WS-* stack for Web Services. In the last decade, RESTful Web Services [8] have been increasingly used. RESTful Web Services are praised for the simplicity of their design and implementation, in comparison with WS-* standards which are increasingly complex and often not implemented. Nevertheless, as extensively discussed by Pautasso *et al* [19], SOAP/WS-* remains the most appropriate choice in many contexts. Our goal is to provide interoperability between existing services. The use of SOAP/WS-* allows these services to remain outside the Web; the web is only used as a message enchange interface. On the other hand, RESTful Web Services exist only within the Web, requiring more reengineering and preventing other kinds of accesses. Furthermore, contrary to REST, SOAP/WS-* technology comes with a fairly robust body of standards for QoS. Thus, when advanced functionalities, such as multicast, are needed, existing WS-* standards can deliver the appropriate capabilities. RESTful WS would have to be extended to support these capabilities in an ad hoc manner [19]. Finally, the variety of formats that can be used to represent a RESTful WS can hinder interoperability, as WS clients may not be able to process all types of payloads. In the rest of this section, `WS` refers to web services constructed with the SOAP/WS-* stack.

WS implementation. As WS technologies mature, many research projects [5,7,10] have focused on devising methods and tools that help with WS development, testing and deployment. Kelly *et al.* have analysed existing programming languages and development environments used by SOA programmers and have identified a number of limitations [10]. For instance, they point out that since object-oriented and scripting languages were originally designed for standalone environments, the extra functionalities that have been added to them for network and distributed processing fail to provide a simple means for designing and implementing services that are invoked remotely. They stress that a good language for WS development should support static typing, so that WSDL definitions can be automatically generated from function definitions. In addi-

tion, all data should be serializable so that it can be sent within SOAP messages. The Janus language meets these requirements.

Kelly *et al.* also propose the GridXSLT execution engine for exposing programs as WS. GridXSLT relies on a language that extends the XSLT programming language for specifying WS operations. GridXSLT only supports functions that are side-effect free, meaning that a service may not maintain state. This constraint makes impossible to implement WS wrappers that involve sessions, such as our RTSP media service wrapper. Janus does not place any restrictions on the type of applications that can be wrapped.

Legacy services migration. Several companies have succeeding in re-packaging their legacy services as WS so as to enable better integration of Web information. For instance, in 2002, Amazon.com released a WS interface (*http://aws.amazon.com*) linked to their existing query engine to provide to computer programs the same service that their primary keyword-based search interface has been offering to humans. Google Web APIs are another example of the migration of human-oriented Web Site interfaces to web services. Lately, many researchers have proposed tools for extracting from web documents information that can then be requested by WS clients [9,13].

The idea of migrating a legacy service to WS has been explored in the literature [3,4,11,21]. Almoanaies *et al.* have recently published a survey of the various existing approaches [1]. Attempts to move legacy services to the SOA environment have been motivated by issues ranging from software reuse and maintenance to interoperability. Our work is dedicated to migrating services built on top of incompatible ALPs to WS so as to benefit from the many features provided by SOA.

Migration solutions that are close to our work have been proposed by Sneed [20] and by Canfora *et al.* [3,4]. Sneed has designed a tool for extracting and wrapping individual functions from legacy code. Canfora *et al.* have devised a method for constructing a wrapper that interprets a Finite State Automaton that models the interaction between the client and the legacy system. In our approach, a wrapper developer is allowed to adjust the characteristics of this interaction so as to fulfill other requirements. For instance, in the case of IMAP server case study, we have designed a WS wrapper with asynchronous operations while messages are sent synchronously in IMAP. Thus the WS client is not required to actively wait while all mail is collected from the server.

Other projects have re-designed and re-implemented ALP-based application functionalities using WS standards. Though such re-engineering solutions can provide flexibility in design and ensure performance, their invasive aspect prevents their wide adoption. WSEmail [14] replaces the existing protocols for email (i.e., SMTP, POP, IMAP, S/MIME) with protocols based on SOAP, WSDL, and other XML-based formats. However, WSEmail does not fully exploit existing email infrastructures and thus fails to recover the logic perfectly [25]. Similarly, Chou *et al.* [6] have proposed an entirely WS-based protocol, WIP, to replace SIP in their WSIP [12] endpoint for converged multimedia/multimodal communication over IP. All communicating entities, however, must support the new protocol.

A compromise approach to legacy service migration is to provide wrappers that interface different parts of a service. The legacy service is thus broken up to several parts, each implementing one or more functionalities. This approach avoids reimplementing these functionalities in the WS, and thus yields practical and less invasive solutions.

Zhang and Yang [25] have presented a service-oriented approach that uses a hierarchical algorithm to understand the legacy code and extract independent services from it. Janus on the other hand lets the developer choose the granularity of the functionalities to expose.

7 Conclusion

In this paper, we have introduced an approach for migrating ALP-based service functionalities to Web Services using wrappers. To this end, we have designed the Janus language, which provides dedicated constructions and operations to hide low-level ALP and WS details from the wrapper programmer. We have also developed a compiler for Janus that automatically generates the corresponding wrapper code in C and the wrapper's associated WSDL service description. Finally, we provide a Janus runtime system that is to be linked with the generated wrappers and that encapsulates the required low-level networking and systems code.

We have successfully used Janus to develop a number of wrappers, including wrappers for IMAP and SMTP servers, for a RTSP-compliant media server and for UPnP service discovery. Our experience in using Janus for wrapper construction shows that our approach drastically reduces the level of expertise required. By freeing the wrapper developer from manually managing both WS details and ALP-based communication issues, Janus bridges the gap between the WS realm and ALP-based services.

Preliminary experiments show that Janus-based WS wrappers have performance comparable to manually written wrappers. Furthermore, the size of the executable code of our Janus-based wrappers, including the runtime system, is small, not exceeding 128KB, which is acceptable in contexts where code size must be minimized, as in some embedded systems. We are currently extending the Janus approach to support WS specifications such as WS-Notification, WS-Eventing, and WS-Security. We are also developing Janus wrappers for other application domains such as network supervision.

Availability: The source code of z2z is available at http://www.labri.fr/perso/reveille/projects/z2z/. The source code for Janus is available on request.

References

1. Almonaies, A.A., Cordy, J.R., Dean, T.R.: Legacy system evolution towards service-oriented architecture. In: SOAME 2010: International Workshop on SOA Migration and Evolution. pp. 53–62. Madrid, Spain (March 2010)
2. Bromberg, Y.D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware. pp. 21–41. Urbana Champaign, IL, USA (2009)
3. Canfora, G., Fasolino, A., Frattolillo, G., Tramontana, P.: Migrating interactive legacy systems to Web services. In: 10th European Conference on Software Maintenance and Reengineering. pp. 10 pp.–36 (Mar 2006)
4. Canfora, G., Fasolino, A.R., Frattolillo, G., Tramontana, P.: A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures. Journal of Systems and Software 81(4), 463–480 (2008)

5. Cho, E., Chung, S., Zimmerman, D., Muppa, M.: Automatic web services generation. In: HICSS '09: Proceedings of the 42nd Hawaii International Conference on System Sciences. pp. 1–8 (2009)
6. Chou, W., Li, L., Liu, F.: WIP: Web service initiation protocol for multimedia and voice communication over IP. In: ICWS '06: Proceedings of the IEEE International Conference on Web Services. pp. 515–522. Chicago, IL, USA (2006)
7. Feuerlicht, G., Meesathit, S.: Towards software development methodology for web services. In: Proceeding of the 2005 conference on New Trends in Software Methodologies, Tools and Techniques. pp. 263–277. Tokyo, Japan (2005)
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
9. Han, H., Kotake, Y., Tokuda, T.: An efficient method for quick construction of web services. In: Proceeding of the 2009 conference on Information Modelling and Knowledge Bases XX. pp. 180–193. Amsterdam, The Netherlands, The Netherlands (2009)
10. Kelly, P.M., Coddington, P.D., Wendelborn, A.L.: A simplified approach to web service development. In: ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research. pp. 79–88. Darlinghurst, Australia (2006)
11. Lewis, G., Morris, E., Smith, D., O'Brien, L.: Service-oriented migration and reuse technique (SMART). In: STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice. pp. 222–229. Budapest, Hungary (2005)
12. Liu, F., Chou, W., Li, L., Li, J.: WSIP - web service SIP endpoint for converged multimedia/multimodal communication over IP. In: ICWS '04: Proceedings of the IEEE International Conference on Web Services. p. 690. San Diego, CA, USA (2004)
13. Lu, Y.H., Hong, Y., Varia, J., Lee, D.: Pollock: automatic generation of virtual web services from web sites. In: SAC '05: Proceedings of the 2005 ACM symposium on Applied computing. pp. 1650–1655. Santa Fe, NM, USA (2005)
14. Lux, K.D., J. May, M., Bhattad, N.L., Gunter, C.A.: WSEmail: Secure internet messaging based on Web services. In: ICWS '05: Proceedings of the IEEE International Conference on Web Services. pp. 75–82. Orlando, FL, USA (2005)
15. OASIS: Web Services Reliable Messaging TC. WS-Reliability 1.1 (Nov 2004)
16. OASIS: Web Services Business Process Execution Language Version 2.0 (Apr 2007)
17. OASIS: SOAP-over-UDP version 1.1 (Jul 2009)
18. Papazoglou, M.P., Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. The VLDB Journal 16(3), 389–415 (2007)
19. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. "Big"' Web Wervices: Making the Right Architectural Decision. In: Proceedings of the 17th International World Wide Web Conference. pp. 805–814. Beijing, China (2008)
20. Sneed, H.M.: Wrapping legacy COBOL programs behind an XML-interface. In: WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01). p. 189. Stuttgart, Germany (2001)
21. Sneed, H.M.: Integrating legacy software into a service oriented architecture. In: CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering. pp. 3–14. Bari, Italy (2006)
22. W3C: Web Services Addressing (WS-Addressing) - W3C submission (Aug 2004)
23. W3C: Web Services Choreography Description Language Version 1.0 (Nov 2005)
24. Walsh, A.E. (ed.): UDDI, SOAP, and WSDL: The Web Services Specification Reference Book (2002)
25. Zhang, Z., Yang, H.: Incubating services in legacy systems for architectural migration. In: APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference. pp. 196–203. Busan, Korea (2004)