Performance Improvement in Web Services Invocation Framework

DIST - University of Genoa, Via Opera Pia 13, 16145, ITALY
Roberto Podesta' and Mauro Migliardi
ropode@laser.dist.unige.it
om@dist.unige.it
Tel. +39 010 3536549 Fax +39 010 353 2154

abstract: *The two main threads in distributed computing that recently show noticeable progress are Grid Computing and the development of Web-based services. It is our opinion that the adoption of the standards fostered by the web-service community could improve the level of usability and interoperability of grid systems and lead to the development of metacomputing systems based on reusable, interoperable components. Up to now the adoption of Web Services in Grid projects has never penetrated the core computational model. This is mainly due to the fact that Web Services use by default the SOAP protocol which is extremely ill suited to computational science applications. In this paper we show how we leveraged the Web Service Invocation Framework (WSIF) to develop a new high performance binding based on XDR coding and how we improved the general WSIF performance by developing an extension based on a fully static service invocation.*
Keywords: Metacomputing, Web Services, stub, WSIF

1- Introduction

Web Services [1] technology and high performance, heterogeneous, distributed computing exhibit quite a few common features. Both Web Services and high performance distributed computing aim at building software architectures capable to work with resources that are shared among multiple domains. However, while the former uses widely spread and standardized communication mechanism, the latter adopts ad-hoc methodologies tightly bound to the framework implementation. This fact imposes strong limitation to the interoperability with other systems and other technologies.
In order to overcome these problems, Web Services technology seems to be a very attractive solution because of its high level standardization. However, the adoption of Web Services in meta-computing and Grid Computing is very limited [2] and has not penetrated the core computational model. In fact, the computational model is still tightly tied to message passing based on software systems such as MPI [3] or PVM [4]. It is our opinion that making Web Services technology paramount in the set of application at the core of metacomputing, would allow developing a new generation of metacomputing system based on highly reusable components.
However, the full adoption of Web Services technology is currently unsuitable to a high performance and dynamic context like Grid Computing. There are three fundamental issues that prevent such an adoption. The first problem comes out from the static nature of the general services model provided by Web Services. In fact, while it is natural for commercial and business oriented services to be rather stable, services targeted at Grid Computing systems are highly dynamic and typically handle volatile information that quickly becomes obsolete. Thus, the service deployment mechanism offered by Web Services is too slow for a HPC environment.
Furthermore, the amount of data involved in computational science is orders of magnitude larger than the data required in business oriented transaction. This fact, turns the use of XML based message system such as SOAP/HTTP [5], that is the most commonly adopted Web Services wire protocol, into a significant liability. In fact, such a messaging scheme introduces a huge encoding overhead that severely hampers its usage in computational science applications [6].
The last issue regards the redundant use of the TCP/HTTP/SOAP stack in case of communication between components residing on the same host. As a matter of fact, in order to minimize the communication

overhead, it should be possible to bypass completely the memory to memory copies introduced by the above mentioned stack when both parties reside into the same address space.
In this paper we approach the encoding problem, in fact, it is our opinion that the adoption of Web Services inside the computational core of Grid Computing applications is totally unfeasible as long as this problem has not been solved.
In order to overcome this problem, we decided to maintain the Web Services Description Language (WSDL) [7] description of service separating it from the use of SOAP based message system. Thus, we defined a new XDR based WSDL binding [8] targeted to the transport of large amounts of numerical data. Such a WSDL binding could be adapted to the requirements of grid applications. Our binding leverages the capabilities of Web Services Invocation Framework (WSIF) and is embedded in this infrastructure. The WSIF project originates in 2001 [9] as an IBM design. In 2002 IBM donated the sources to the Apache Software Foundation that took care of further development.
WSIF APIs provide a simple way to invoke Web Services, no matter how and where they are published. WSIF extends the "service oriented" model to services that are reachable through different protocols. The only pre-condition is that the service must be described by means of a WSDL document. In the WSIF model, WSDL becomes a standardized way to describe software components.
However, the WSIF framework is not targeted at high performance computing. In fact, WSIF introduces a significant overhead for the run-time parsing of the WSDL document.
In this paper we will describe how we improved the performance of the WSIF framework without renouncing to the WSDL service description. It is our opinion that our improvement to the performance of the WSIF framework performance, together with the reduction of the coding overhead we obtained with the XDR based binding, constitute a significant step in the direction of making Web Services technology well suited to the requirements of metacomputing and Grid Computing systems and applications.
This paper is structured as follows: in section 2 we provide a general overview of the WSIF architecture and of the key role of WSDL; in section 3 we describe our grid oriented binding embedded in WSIF; in section 4 we describe how and where we changed WSIF to get over the overhead introduced by the run-time WSDL parsing task; finally, in section 5 we provide some concluding remarks and some preliminary performance results.

2. Web Services Invocation Framework

The main goal of WSIF is the integration between heterogeneous applications. By providing a simple set of APIs, WSIF allows to build client routines that do not need to change according to how the service is reachable in terms of protocol used and where the service is located. WSIF is a client side framework written in Java, it aims at acting as a proxy between the caller and the service, and it aims at hiding the details of the communication to the client. The client can always use the same APIs. At present, WSIF supports bindings for SOAP, JMS, JCA, EJB and local Java objects (i.e. caller and callee inside a single JVM). WSIF leverages the implicit separation of a WSDL document in three different sections: the *portType* section, the *binding* section and the *service* or *port* section.
The first part defines the set of operations that are offered by the service: it is the abstract interface of the service. This part includes three types of tag: the *operation* tag, the *message* tag and the *type* tag. The *operation* tag defines the names of the methods that can be invoked by the clients to request services. The *message* tag allows describing the method invocation in terms of data types exchanged. Finally, the *type* tag allows defining new complex types.
The binding part allows mapping the abstract definition of the set of operations offered by the service (the *portType* section) with the real service through a specific protocol. The association between the *binding* and the *portType* tells how the service can be actually reached from the client. This section includes information about the mapping of the data types and the operation but in general depends on the wire protocol chosen.
Finally the *service* or *port* section defines where the service is located in terms of physical network neighborhood. Actually, it defines the URL where the service is available.

WSIF is the first framework that extends the Web Services model removing the practical constraint of having SOAP like transport protocol. In fact, the WSIF APIs are based on the abstract WSDL description of the service and so they mainly operate at a protocol-independent level. This feature, together with its high level of customizability, makes WSIF very attractive in order to maintain the WSDL description in metacomputing environment. Thus, WSIF appears to be a good base onto which to develop a binding capable of overcoming the inefficiencies typical of SOAP in the common Web Services architecture. In Figure 1 we show an example of generic WSIF usage.
The invocation of an abstract operation offered by a service has to be converted to a specific mechanism of communication for contacting the service endpoint. This means that WSIF needs a piece of software that enables the invocation through a specific protocol. This component is named *WSIFProvider*. Actually, there must be a *WSIFProvider* for each binding supported by your installation of WSIF. At present (i.e. release 2.0) there are WSIF Providers for SOAP, JMS, JCA, EJB and local Java. As an example, in order to invoke a service with a SOAP binding, it is necessary to use a *WSIFProvider* that supports such a binding. In case of SOAP binding there are two Providers, one based on Axis and one based on Apache SOAP.
A *WSIFProvider* is a container that offers the capability of using one of the specified communication mechanism. For dynamically handling the WSDL document, WSIF uses the services provided by the WSDL4J project, namely a Java implementation of a WSDL parser. WSDL4J natively supports only the standard WSDL1.1 binding: SOAP, HTTP and MIME. So, in order to allow using other binding inside the WSIF framework, it is necessary *i)* to define WSDL binding extensions, *ii)*to define the corresponding WSDL4J extensibility elements with WSDL4J and *iii)* to add the serializer and deserializer objects for these WSDL extensions with the WSDL4J Extension Registry. It is very important to notice that WSIF allows both dynamic and static invocation. The former invocation type gets all the service information by parsing the WSDL document at run-time. The latter type is able to use a stub tied to the *portType* section. However, this mechanism does not allow avoiding the WSDL parsing, in fact the information belonging the stub replaces only the information obtained from the parsing of the *portType* section.
When a client component invokes a service of one of the supported binding, an appropriate provider is looked up. The logic to do this matching is in the class *WSIFPluggableProviders*. This class is linked to a file that contains a list of the registered providers. These providers must implement the *WSIFProvider* interface. The choice of the provider takes place through the methods of *WSIFService* that receive the binding description by the WSDL parsing. Thus, the actual service invocation takes place without involving the client into the details of the specific transport.

3. WSDL Grid Binding: an XDR based binding

Among the bindings introduced by WSIF, we found the local Java binding [10] very promising. This binding allows invoking a WSDL described service using a Java-like wire protocol. The Java binding leverages the Java reflection and largely reduces the encoding overhead. Obviously, it is able to call only services located into the same Java Virtual Machine of the caller. However, it seems to be the first step toward obtaining "Web Services sans SOAP" and it is a viable solution to the third performance issue we described in the introduction section. A new binding focused onto high performance must have a behavior similar to the Java binding for the data encoding. Moreover such a binding needs to be untied to a single language and must allow communicating between software components distributed over the entire Internet.
In order to implement a binding oriented to the passage of large amounts of numerical data, we chose to adopt eXternal Data Representation (XDR) [11] as the wire level coding on top of which to develop the new WSDL Grid Binding.
XDR provides a way to exchange data between heterogeneous machines as it defines a platform independent data representation. Therefore XDR seems to be extremely well suited to Grid Computing. By leveraging the high interoperability of WSDL and the lightness of XDR, we developed a binding that allows accessing a Web Service without compromising the system performance.

WSIF is closely bound to the WSDL4J project [12], therefore the first step has been the implementation of a XDR extensibility element for WSDL4J and appropriate serializer and deserializer objects with the WSDL4J extension registry.
Our implementation is oriented to numerical applications, so it supports only primitive numeric types and structures like arrays of primitive numeric types. These numeric data are converted in byte arrays multiple of four bytes, following the XDR encoding rules.
In order to test our extension we implemented a simple server (written in Java) that is able to receive XDR data, converts them to Java types and invokes the requested method through the Java reflection.
WSDL Grid Binding implements the *WSIFProvider* interface and follows the general model proposed from WSIF to implement a specific *WSIFPort* and *WSIFOperation*. The data package sent to the server contains the following data: the IP address of the service provider, the physical class path of the service, the number of the listening TCP port, the remote method name, the formal parameters and the actual parameters. Such information is passed to a client proxy that opens a TCP connection with the server and waits for the method return value.
The results of our tests onto the WSDL Grid Binding embedded in WSIF demonstrate that such an invocation is an order of magnitude faster than SOAP, especially for the transport of big arrays of data. In fact the performance results for the exchange of little chunks of data (up to 10 KB) says that WSDL Grid Binding is almost 15-20 faster than SOAP Axis. When the dimension of the data sent increases over 10 KB the comparison becomes impossible. In fact while the transport time with Axis tends to diverge till the system crashes, the WSDL Grid Binding time remains almost constant. As a final remark, it is important to notice that the use of WSIF introduces a constant latency due to the WSDL parsing time. This time is independent from the transport provider and invocation type (dynamic or static) chosen by the client.

4. Improvement in WSIF Stub Invocation

In the previous paragraph we mentioned the performance hit that still persists when using our WSIF extension in a high performance scenario. This problem comes out from the completely dynamic service invocation in WSIF due to the WSDL parsing at run-time.
Even the stub-based invocation requires parsing the WSDL document, in fact the information obtained from the stub replaces only the information obtained by parsing the *portType* section of the service in the description document. This is not a mistake of the WSIF authors because their target is the integration between heterogeneous application, not the extension of Web Services to Grid Computing. The fundamental effort of WSIF is to make WSDL an extensive way to describe software component and to use it directly in the service invocation without taking care of the transport protocol needed.
The weight of the WSDL parsing obviously depends on the speed of client machine, however, with the level of performance of current PCs it always takes a significant amount of time and in case of WSDL Grid Binding this time is 10-15 times longer than the time required to actually move the data over the wires[1]. Removing this latency would represent a significant improvement to the WSIF architecture and thus would make WSIF and our extension really attractive for Grid Computing.
In order to overcome this problem, we focused our attention on the stub invocation. The stubs are software component that are able to provide the same information contained in a WSDL file in the form of Java classes or interfaces. The classical use of SOAP Axis [13] without WSIF takes advantage of the adoption of the stubs. In the Axis framework, the stubs can be generated by *WSDL2Java* tools, describe the interface of the service and the location of the service and take care of communicating directly with the service provider actually acting as a network proxy.
In the case of our WSIF extension, the stubs could just have the role of providing a complete service description in the form of a Java interface instead of a WSDL document. They won't need to operate at network level as this task is delegated to WSIF.

[1] This is true for data set up to ten megabytes.

The general WSIF routine client takes place with a call to the method *getService* of the class *WSIFFactoryImpl*. This call instantiates all the supported *WSIFProvider* classes and starts the WSDL parsing. Then it chooses the correct binding for the service. The next step changes if the invocation is completely dynamic or semi-static. In the former case, the correct implementation of the *WSIFPort* and *WSIFOperation* are instantiated directly and the service is invoked through their methods. In the latter case, the caller invokes the method *getStub* of the *service* object returned by the *getService* method. This method loads an interface that describes the services in term of the name of the method required, and then calls a proxy, *WSIFClientProxy*, that takes care of the actual invocation using the correct implementation of the *WSIFPort* and *WSIFOperation*. In figure 2 is presented the core code of the two client routines.
Our WSIF extension aims at obtaining a fully static service invocation. We introduce a new class, called *WSIFFactory4Stubs*, that presents a *getServiceWS* method too. Such a method, after having instantiated all the supported transport providers, takes the information about the portType, the binding and the service end point from Java classes generated off-line, taken as *Class* parameter. This method includes the call to a routine that takes care of invoking the service, by setting of the correct *WSIFPort* and *WSIFOperation.* The return type of *getServiceWS* ("WS" stays for "With Stubs") is an *Object* element and it is the actual result of the service invocation. In Figure 3 we show the general operation model of the use of our fully static invocation.
At the moment the classes that represent a service description are not automatically generated and, in general, must be coded by hand. In fact, the original *WSDL2Java* tools only support the Axis Provider. However, we are currently developing a tool capable of automatically generating stubs for our WSIF extension starting from the WSDL document.
It is important to notice the difference between the binding stubs use in Apache Axis and in our WSIF extension. When *WSDL2Java* generates the stubs starting from a WSDL document, a *NameOfServiceSOAPBinding* class is created. Such a class works as a proxy and is responsible of the physical call of the service. Besides, it contains the invocation of the method *Call()*, that allows the typical SOAP-Web Services invocation. In our extension the binding stub does not act as a network level proxy, because this task is obviously delegated to WSIF. The binding stub is only a descriptor of the transport mechanism used and allows passing to WSIF the correct parameters to set the correct transport provider.
With our completely static invocation the WSDL parsing is skipped, and the invocation time depends only from the speed of the protocol on which the chosen *WSIFProvider* is based. The invocation remains transparent to the user although the code for building the client routine is different to the standard WSIF samples. However, it is far simpler than the WSIF dynamic invocation and equivalent to WSIF original stub invocation.

5. Concluding Remarks

In this paper we have described our extension and modification to the Web Services Invocation Framework. We developed a new remote method invocation based on the XDR standard and targeted to the transport of large amounts of numerical data. This protocol can be embedded in WSIF and with the WSDL description of the service it is possible to invoke the service in a completely transparent manner. Moreover, our performance experiments showed us that WSIF suffers from a heavy overhead due to the systematic run-time parsing of WSDL documents. To remove this performance cost we developed a WSIF extension that is able to invoke a service without the WSDL parsing. This extension uses off-line generated stubs. The stubs are generated directly from the WSDL document. The adoption of our completely static invocation allows reducing the total invocation time just to the actual communication time. WSDL Grid Binding and the WSIF static stub based invocation extension demonstrate that it is possible to use Web Services technology *i)* without renouncing to high performance and *ii*) without loosing its high level of standardization. The full paper will include a more detailed description of our WSIF extensions and the actual results of our benchmarking experiments.

References

1 M. Champion, C. Ferris, E. Newcomer, D. Orchard, Web Services Architecture, available on-line at http://www.w3.org/TR/2002/WD-ws-arch-20021114/

2 The Globus Project Team, Open Grid Services Architecture, available on-line at http://www.globus.org/ogsa/

3 Message Passing Interface Forum, Message Passing Interface, documentation available on line at www.mpi-forum.org

4 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam, PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.

5 D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Frystyk Nielsen, S. Thatte and D. Winer, Simple Object Access Protocol (1.1), available on line at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

6 M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, D. Gannon **(**Department of Computer Science Indiana University Bloomington), Requirements for and Evaluation of RMI Protocols for Scientific Computing, http://www.extreme.indiana.edu/xgws/papers/sc00_paper/

7 E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, available on line at http://www.w3.org/TR/wsdl

8 Roberto Podesta' and Mauro Migliardi, WSDL Grid Binding, Proc. of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications, pgg. 43-49, Vol I, Las Vegas, Nevada (USA), June 23-26, 2003.

9 M. J. Duftler, N. K. Mukhi, A. Slominski and S. Weerawarana, Web Service Invocation Framework Proposal, http://www.research.ibm.com/people/b/bth/OOWS2001/duftler.pdf

10 WSIF Team, WSDL Java Extension, documentation available on line at http://ws.apache.org/wsif/providers/wsdl_extensions/java_extension.html

11 R. Srinivasan, RFC 1832 XDR: eXternal Data Representation, documentation available on line at http://www.faqs.org/rfcs/rfc1832.html

12 WSDL4J IBM Open Source Project, http://www-124.ibm.com/developerworks/projects/wsdl4j/

13 Apache Axis Open Source Project, http://ws.apache.org/axis/

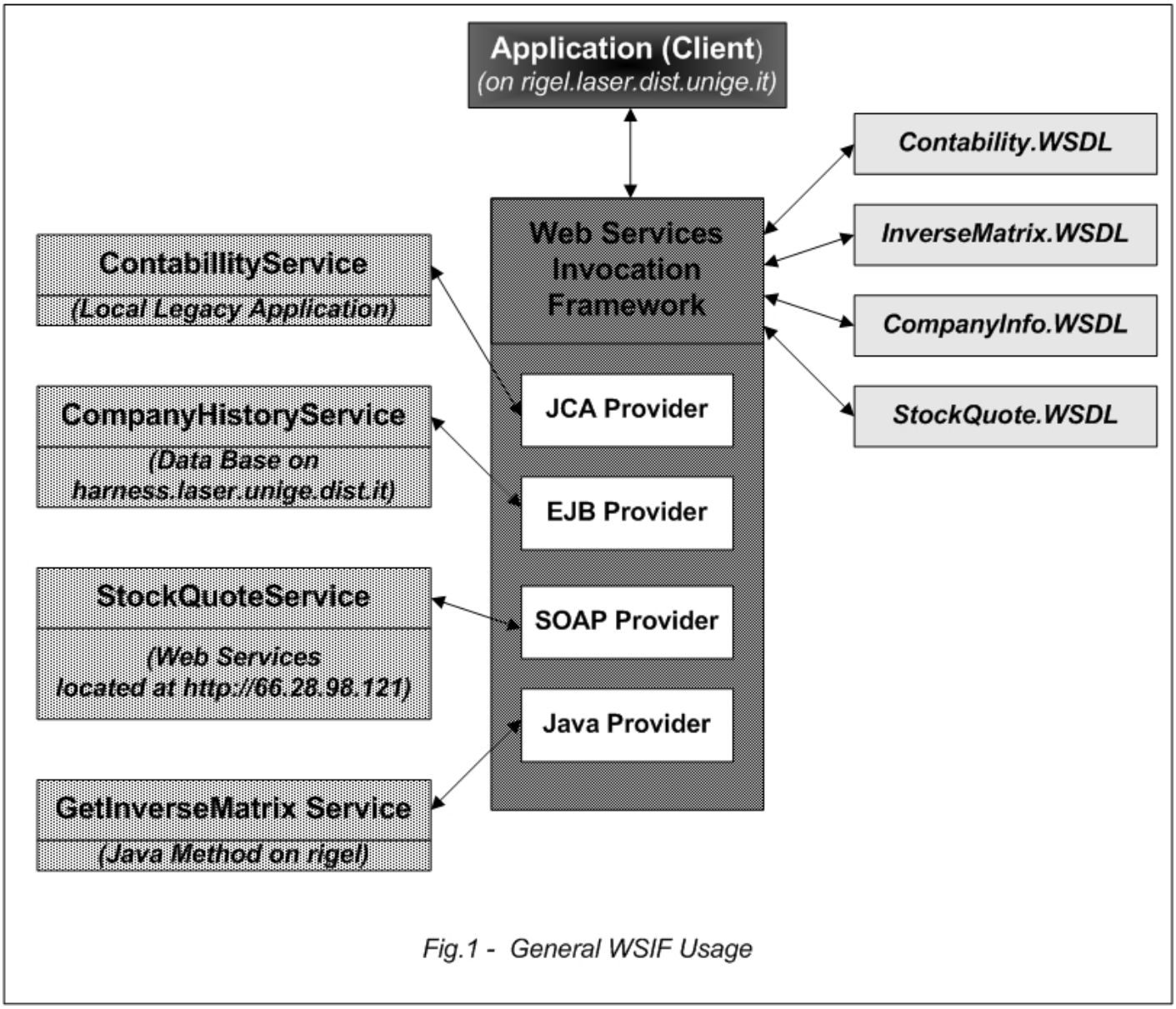

Fig.1 - General WSIF Usage

```
//WSIF Stub Invocation

//(…)
 WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
 // parse WSDL
 WSIFService service = factory.getService(args[0],
                                          null,
                                          null,
                                          "http://wsifservice.intservice/",
                                          "IntServicePortType");
// create the stub
IntServiceStub stub = (IntServiceStub) service.getStub(IntServiceStub.class);
int result = stub.retrieveInteger(2003);
//(…)
```

```
//WSIF Stubless Invocation

//(…)
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
// parse WSDL
WSIFService service = factory.getService(args[0],
                                         null,
                                         null,
                                         "http://wsifservice.intservice/",
                                         "IntServicePortType");
// create the stub
WSIFPort port = service.getPort();
WSIFOperation operation = port.createOperation("retrieveInteger","GetIntRequest", null);
WSIFMessage input = operation.createInputMessage();
WSIFMessage output = operation.createOutputMessage();
WSIFMessage fault = operation.createFaultMessage();
Integer theNum = new Integer(2003);
input.setObjectPart("Number", theNum);
if (operation.executeRequestResponseOperation(input, output, fault)){
    Integer res = (Integer)output.getObjectPart("Result");
}
//(…)
```

Fig. 2 - Core code of WSIF Stub Invocation and WSIF Stubless Invocation

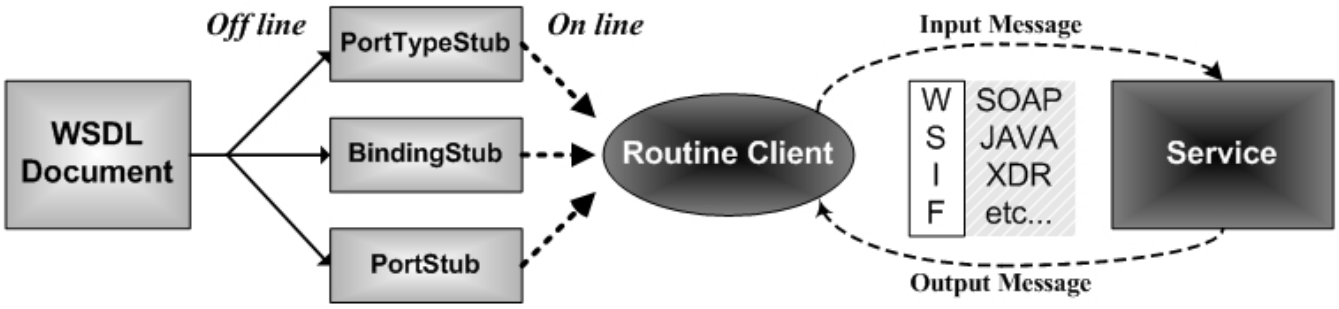

Fig.3 - General Operation Model of fully static WSIF invocation