An Approach to Debugging Grid or Web Services

Qiang YUE, Zhiwei XU, Haiyan YU, Wei LI, Li ZHA
Research Center for Grid and Service Computing
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100080, P.R. China
{qiangyue, zxu, yuhaiyan, liwei, char}@ict.ac.cn

Abstract

In this paper, we first introduce some issues that are encountered in building a service debugger and briefly describe our approach to addressing them. Next, we outline some debugging modes and components of a simple composite debugger. Then, we mainly describe its message-based front-end and back-end, which are a co-existing, self-identifying, and non-intrusive. Finally, we preset some experimental results of our latest prototype.

Categories and Subject Descriptors -- D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids, Distributed Debugging, Dumps, Monitors, Tracing*. D.4.1 [**Operating Systems**]: Process Management – *Concurrency, Threads.* D.4.9 [**Operating Systems**]: Systems Programs and Utilities.

General Terms – Management, Design, Experimentation.

Keywords – *Composite debugging, Behavioral trace, Context inspection, Behavior controller, Debugging migration, Co-existing, Self-identifying, Non-intrusive, System debugging service.*

1. Introduction

Services are autonomous, platform-independent computational elements that can be described, published, discovered, orchestrated and programmed using protocols to bind networks of collaborating applications distributed within and across organizational boundaries [1], [2], [3], [4]. In contrast, the problem of debugging for grid or web services has received less attention.

Debugging a service at internet-based environments is a more difficult task due to many factors including the nature of a service itself, the nature of a run-time environment and specific features of the descriptive language. These factors have made traditional methodologies and tools not powerful enough. In particular, a need often arises for the focused, on-demand debugging of a certain service, where the focus of one or more programmer(s) may refer to inspect a message interaction, a single behavior, a sequence of services. For example, in a testbed of grid or web computing, a remote debugger built on special standard debugging protocols maybe violate a run-time environment by making its client dependent upon the specific implementation, e.g., imperatively turning on a debugging infrastructure (controlled by a -debug or –trace command line argument [5], [6]), and disabling a firewall for successful service debugging as it takes at least one independent port. Furthermore, a container for a collection of registered services could not usually turn on or restart with such an infrastructure, once a bug occurs in a service. That is, in such a case the traditional debugging infrastructure is unavailable.

To tackle these problems and make conventional debugging techniques and tools easily applying to service inspections, the vega service debugger (vdb) takes an approach to a composite mode, both message-based and source-level debugging for services, and lets programmers do some debugging activities in both levels. In the vdb, service debugging is organized based on behaviors of message processing. Detecting and localizing bugs typically require programmers to inspect dynamic behaviors and related contexts during the problem execution, and to check consistency between the observed behaviors and the expected behaviors in message processing. Focusing on these behaviors would allow programmers employing different debugging strategies in understanding and communicating the expected behaviors in a service.

In this paper, we draw out approaches to capturing typical scenarios, in which the key paradigms are used to debugging grid or web services. In the remainder of this paper, we describe a composite mechanism and its components in detail. In section 2, we present an

2007 IEEE International Conference on Web Services (ICWS 2007)
0-7695-2924-0/07 $25.00 © 2007 IEEE

overview of some debugging modes, and the architecture of a simple composite debugger. Section 3 and 4 give components of message-based front-end and back-end respectively. Section 5 gives experimental results of our latest prototype. Conclusions are presented in section 6.

2. Overview of composite debugging

2.1. Some debugging modes

The traditional source-level debugging mode is usually that one front-end connects or attaches one back-end with a specific debugging infrastructure. This means a debugged process or a service container or a process must always turn on the debugging mode for inspecting a service. In a composite debugger, programmers could use new modes to debug grid or web services, including:

1. **Message-based mode**. It allows programmers debugging one or a sequence of services in a normal (un-debugged) container. In this mode, debugged services can be examined separately. A debuggee dynamically identifies itself, both the client caller and service in a message interaction, with a self-identifying behavior in its service deployment configurations. It can determine debugging activities that positively affect on it. Operations of message-based debugging facilities agree on a debugged service and a behavior they manipulate. Such debugging facilities could co-exist with debugged services in the same system and support a composite debugging mode.

2. **Composite mode**. It could be classified into two types: mixed debugging and layered debugging.

a) Mixed debugging. It is assumed that a local client program, an original caller, is easy to start upon with a conventional debugging infrastructure. Therefore, a mixed mode is that the caller can be debugged in source-level, and the remote service can be inspected with a message-based mode. That is, different modes can be separately applied to the caller and its service.

b) Layered debugging. With message-based debugging, a programmer could also inspect a single behavior of the service in source-level, where a source-level back-end either local or remote can start up and create a new debugged process. A special debugger could connect or attach it, and then source-level debugging activities can be performed.

Furthermore, a composite mode is limited in that a message interaction can be suspended at a behavioral boundary and the behavior can be loaded dynamically and executed with a message context and its own options independently. This limitation is not a serious obstacle in service-oriented systems with a high-order abstraction.

2.2. Simple composite debugger

The architecture of a simple composite debugger may include a composite front-end, a message-based back-end, and/or a source-level back-end, shown as Fig. 1.

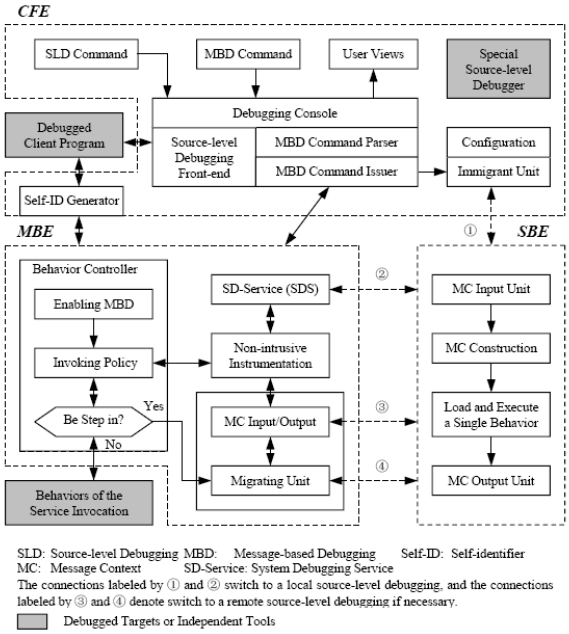

Figure 1. The architecture of a simple composite debugger for grid or web services

In a composite front-end, a debugging console can manage a set of debugging commands and a group of user views for both a source-level debugging front-end and a message-based one. A source-level front-end can be used to debug a client program, a client caller for a service. It is same as other local source-level debugging. For remote debugging, the caller identifies its own request message with a self-identifier generated by the *self-ID Generator*. A message-based front-end consists of a command parser and a command issuer to deal with remotely debugging activities. It could also trigger a local source-level debugging for a moved behavior come from the normal service container with appropriate environmental configurations.

A message-based back-end is used to communicate between a message-based front-end and a sequence of debugged behaviors, and even to move a single

behavior to a new debugged environment when a programmer wants to step in the behavior.

It has following components: a behavior controller, a system debugging service for non-intrusive instrumentation, and a migrating unit for message context input and output. The behavior controller is used to control execution of a debugged service, a sequence of behaviors in a remote message procedure, with behavioral steps, e.g., waiting when a behavioral breakpoint hits, and skipping or retrying a behavior when a control command request is received. With interfaces of a system debugging service, non-intrusive instrumentation can also co-exist with debugged services in a normal container. The migrating unit is used to isolate a single behavior in source-level debugging if necessary.

A source-level back-end either local or remote first gets a set of necessary context elements from the message-based back-end. According to the received information, it can reconstruct its own message context. Then, it loads and executes the single behavior, which could be controlled by a special source-level debugger. When source-level debugging completed, the source-level back-end could output necessary context elements to the message-based back-end. These elements could update the current message context of a debuggee.

As above, we can see that a composite debugger with some new debugging modes could extend conventional debugging techniques and tools for service inspections. Since there are many literatures to study techniques and tools of source-level debugging and the limitation of this paper, in following sections we mainly discuss about message-based debugging, especially for remote services.

3. Message-based front-end

3.1. MBD command line

A command line inputted from a debugging console means to perform a debugging activity. Two important characteristics of a message-based debugging mode are co-existence and self-identification. The co-existence means that debugging facilities do not need violate the current run-time environment, especially a remote service container. The self-identification denotes that a debugged service could determine itself and a debugging activity that could inversely control it.

Debugging command message interactions can be treated in a uniform mode across proper abstractions. The processing mechanism could format a request and sends it to a system debugging service via same standard web-messaging protocols for services without other more special requirements for the environment.

For remote inspections, the name of a debugging command is related to an operation of a system debugging service. To identify a debugged service, an address of the system debugging service and a unique identifier of the debuggee always follow a command name, which are used to determine the service and the target debuggee. Moreover, in order to observe or modify a context element, an expression must be provided to represent the element.

Therefore, for remote debuggees (D) and co-existing system debugging services (S) with a group of operations (O) for message-based debugging, the structure of a command line (*cmdl*) can be defined as:

For $\forall(d,s) \in D \times S$, $\exists o \in O, O \in s$ in a normal container, $cmdl = (nm, < addr, uid >, Expr)$, where, *nm* is the name of a debugging command (o=*nm*), *addr* is the address of a system debugging service (s), *uid* is the unique identifier of a debuggee (d), and *Expr* is the expression to represent a target object (including the empty for a default one or if not needed).

For example, to continue a debuggee (*vdcont*) at a behavioral breakpoint, we can input a command line as follows:

```
vdcont
-r
http://10.61.0.67:8080/axis/services/VegaDebugService
-m MID-11678837283091
```

where, *-r* URL denotes a remote system debugging service, *-m* MCID-xxx sets the unique identifier of a debuggee. Another one for calling a method in the current message context (*vdinvoke*) can be input by:

```
vdinvoke
 -r
http://10.61.0.67:8080//axis/services/VegaDebugService
 -m MID-11678837283091
requestMessage.getSOAPPartAsString()
```

where, the *Expr*, *requestMessage. getSOAPPartAsString()*, following the address and identifier can represent a method name.

A composite front-end provides a set of commands for message-based debugging that enable programmers could manipulate one or more debugged services. These commands is classified in three groups: control, query and update, and helpful.

1. **Control commands**. They are mainly used to set executing states of a remote message procedure of the debugged service in a normal container for message-based debugging, e.g., to continue a suspended debuggee, to step into a behavior.

2. **Query and update commands**. They invoke corresponding operations of a system debugging service to inspect context elements when a debuggee is

suspended or hits a breakpoint, e.g., to observe or modify a value of a variable or a context element.

3. **Helpful commands**. They are used to display helpful information, and to set or get properties about a debugging console and a group of user views, which may be called by other commands as default properties.

3.2. Command procedure

The command parser in a message-based front-end first buffers a command line, which is input from its debugging console, and verifies the command. Then, it processes arguments within a command line. It constructs a set of elements of a debugging command request to a system debugging service. Moreover, because some debugging activities can be processed asynchronously, e.g., a delayed observation of some context elements, and a periodic check the executing state, the parser must distinguish these activities and create proper sub-threads for them.

Similar to a normal client caller, the command issuer takes elements for a debugging command request from its command parser. To distinguish this message interaction from that of a debugged service, it sets the caller as a "system" role in its executing context. The procedure of a command request is shown as Fig. 2.

```
Get elements for a debugging command request;
Construct a new instance of the client service CS;
    Create a new call instance C∈ CS;
    Set the endpoint address of C;
    Set the operation name of C;
    Declare types of parameters in C;
    Set a system role in the message context of C;
    Set an expression parameter (Expr) if it is not empty;
    Invoke the system debugging service with parameters;
If it is a local stepping-in request, then
    Construct new elements of C;
    Get the description of a moved behavior;
    (make another remote invocation
          of the system debugging service);
    … …
    Trigger Immigrating Unit for debugging the behavior;
Endif;
Transform the result object to string text to display;
End of the debugging command request.
```

Figure 2. The procedure of a debugging command.

In general, for a response result of a debugging activity, the issuer transforms a result object to a string text and then sends it to its user view. When a command request requires locally debugging a remote behavior in source-level, it is also responsible for getting information of the moved behavior before triggering an immigrant unit. To get the information, it should make a new call for a special operation of the system debugging service.

3.3 Immigrant unit

The most important feature for composite debugging is that it can switch debugging between a message-based mode and a source-level one, when a programmer wants to step in a behavior moved from a normal container.

An immigrant unit triggered by its command issuer first reads a local configuration file, which defines properties of a run-time environment for debugging a moved behavior. As above, to satisfy such a debugging migration, these conditions can be represented by $(< BDesc, Conf >, Elms)$:

1. Arguments for a new debugged process.

a) The behavioral description (*BDesc*). It includes: (1) a class name or a library name with a fixed method, and (2) a set of options of the behavior. For example, the following XML configuration given a handler description in WS-Axis deployment configuration [3], where, in a request flow there is a *URLMapper* behavior with a default "invoke" method.

```
<requestFlow>
<handler type="java:org.apache.axis.handlers.
    http.URLMapper"/>
</requestFlow>
```

b) Configurations (*Conf*). It contains: (1) a main name or a main class for the new locally process, (2) arguments with a command line to start the debugging infrastructure (e.g., for debugging a java process, *-agentlib:jdwp=transport=dt_shmem,address=test,server=y,suspend=y*) [5], (3) a related configuration of the service deployment, (4) relative library directories, and (5) a working directory.

The command issuer requires a behavioral description and a set of configurations as command line arguments to start a debugged process.

2. A group of necessary elements (*Elms*).

Because the behavioral abstraction for services is programming language-neutral, elements for the migration can also be limited with the current message context. For example, in WS-Axis, it includes (1) descriptions of the current service and operation, (2) XML data of the current SOAP message, (3) actor roles of context, and (4) some attachments if existed [3].

When a debugged process starts up, it can implicitly call some operations of a system debugging service, in order to exchange these elements at the beginning or the end for source-level debugging of a single behavior. These messages commute between the debugged service and the new debugged process. For example,

the *MC Input Unit* at a source-level back-end invokes an operation to get related elements from the remote debuggee. Then, the back-end could construct its message context (called *MC Construction*) according to these elements. Another service occurs at the end of source-level debugging. Likewise, a migrating unit could create a new process at the server site for remote source-level debugging.

4. Message-based back-end

4.1. System debugging service

A system debugging service provides a channel to communicate between a message-based front-end and a behavior of a debugged service. One of its operations calls a function of non-intrusive instrumentation when a debugging command request arrives. Generally, it could be considered as interfaces of the instrumentation. Because of using a system actor role, it could not be broken or interrupted.

For each debugged service, it provides control, examining operations in order to response debugging activities. These operations are classified as follows:

1. **State operations**. Set or update executing states for a debugged service, e.g., step-in, next, continue, and waiting. Some compensable operations are also provided to alter execution of a debuggee, e.g., skipping, compensating after, and retrying a behavior (see also section 4.3).

2. **Breakpoint operations**. Insert or delete a behavioral breakpoint, and enable or disable behavioral break;

3. **Query and update operations**. Get or generate a behavioral trace, and dump information of a context element (e.g., to buffer or construct an inspected context, to set arguments for a method, and to invoke a reflective method).

4.2 Non-intrusive instrumentation

Non-intrusive instrumentation consists of three components: a generator of behavior trace, a manager of debugging information and an expression evaluator, and a set of mapping spaces for related context objects and other debugging data.

1. **Generator of behavioral trace**. It could dynamically create a set of behavioral descriptions for a debugged service. The descriptive set is mapped to a mapping space, with a key of the unique identifier. Each behavioral trace is independent, so that the debuggee could be separately controlled [10].

For a one-way message exchange, there exists a self-contained trace:

$T = \{T_c, T_s\}$, where,

T_c denotes a sequence of behavioral descriptions of the client caller, and T_s does one of the service. For other message exchange patterns, the trace is:

$T = \{T_{c,rep}, T_{s,rep}, T_{s,res}, T_{c,res}\}$, where,

behaviors of both the caller and a service are divided into two groups: the message request ($T_{c,req}$ and $T_{s,req}$) and the message response ($T_{c,res}$ and $T_{s,res}$).

2. **Manager of debugging information**. It is responsible for updating executing states of debugged services, setting or deleting behavioral breakpoints, and mapping these debugging data to related mapping spaces.

In each mapping space, a set of debugging information for co-existing debuggees is marked with their own identifiers. In a normal service container, all executing states for debugged services could be mapped into a state mapping space (*SMS*) in non-intrusive instrumentation with identifiers. It can be also denoted as: for all these debugged services (*D*), $\forall d \in D$, $\exists SMS$, then all $S \mapsto SMS$, where the $\mapsto$ symbol means "map into". Similarly, behavioral breakpoints could also be mapped into a breakpoint space [10].

3. **Expression evaluator**. It performs to a computing function, which evaluates the return value of an expression for a context element.

For service debugging of multi-programmers (*P*), there exists a uniform mapping space (*UMS*) to map their simultaneously inspected contexts (*C*) in a normal service container. It can be also denoted as: for a programmer, $\forall p \in P$, $\exists UMS$, then all $C \mapsto UMS$ [10].

As an expression can be considered as a high-level description of a reflective context element, the relationship between them could be similar one between an interface and a particular implementation. An expression can be decomposed with some functions in non-intrusive instrumentation, and encapsulates its implementation with local language reflections [7], [8], [9]. With WS-Axis systems [3], to observe an operation description in the remote message context, we can directly give the field name, *currentOperations* in the *MessageContext* class. With the java programming language, locally reflective capabilities are centered on the class *java.lang.Class*, extended by the core reflection package, *java. lang.reflect*. The *java.lang.reflect.field* supports local introspection [9]. Therefore, we can query the field. Similarly, other core reflections can also represented

by context expressions, e.g., methods, constructors, and member classes.

4.3. Behavior controller

With a well-form abstraction, in a normal service container the operational model of debugged services can be seen labeled transition systems $(B, S, \rightarrow)$, where B is a set of labels, S is a set of executing states, and $\rightarrow \subseteq P(B \times S)$ is a transition relation. Furthermore, for the behavioral controller we have following propositions: behavioral step, compensable policies.

1. **Behavioral step**. For all $b, b' \in B$, and $s \in S$, $< b, s > \rightarrow < b', s' >$ and $< b, s > \rightarrow < b', s'' >$ implies $s' = s''$. The behavioral step is much weaker than requiring that a debugged service be deterministic. Any transition will satisfy it after some simple local transformations at each behavior in the service that fails. For example, if $< b, s > \rightarrow < b', s' >$ and $< b, s > \rightarrow < b', s'' >$ but $s' \neq s''$, we can simply create a new b'' and change $< b', s'' >$ to $< b'', s' >$. Essentially, the behavioral step states that from a $< b, s >$ there can be many possible behavioral steps or compensable policies. In practice, a behavioral step could consist of one or more conditions that determine when a service should be interrupted. A behavior controller verifies a message-based debugging mode, controls its debugged service at each behavioral boundary, and triggers its migrating unit if needed. At each behavioral boundary, the state diagram of invoking policies is shown as Fig. 3.

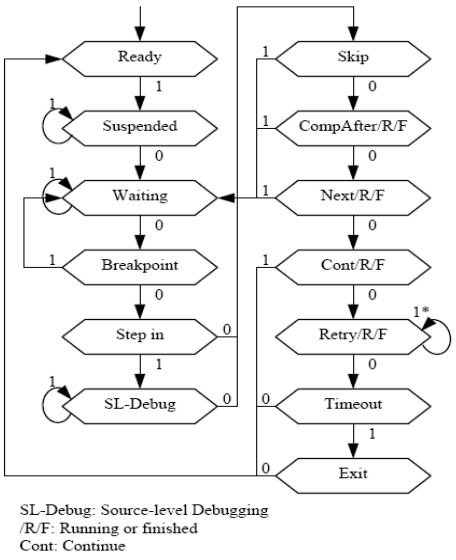

Figure 3. The state diagram of invoking policies in a behavior controller.

A behavioral breakpoint is an intentional stopping or pausing place in a service, put in place for a debugging purpose. At a behavioral breakpoint, a service enters into a waiting state. If the state switches to step-in, it goes in a source-level debugging state. By the self-identification, it could satisfy to capture a single control flow for the service. In the service container, many services may invoke the same behavior, e.g., one in the scope of a global request or response flow; thus mutual interference between their services may be avoided if setting a breakpoint in the behavior.

2. **Compensable policies**. At a behavioral breakpoint in message-based debugging, compensable policies consist of forward and backward invoking strategies. There are three basic policies with the executable behavior semantics.

a) **Skipping**. It will disable execution of a behavior and make forward to the immediate next behavior if it exists, or else stop the service.

b) **Compensating-after**. It is compensating a behavior just after finishing its execution and before initiating the next behavior, e.g., to temporally insert an additional behavior, to restore the previous message context.

c) **Retrying**. It will make backward and repeat a finished behavior.

Both the compensating-after and the retrying could occur after the completion of a behavior and before the termination of a debugged service.

Other states including ready, suspended, continue, next, timeout, and exit, are imperatively or explicitly used to manipulate a debugged service. A behavior controller could also initialize and construct a behavioral trace by calling the generator in non-intrusive instrumentation. It can also put the current message context to a global mapping space when the executing state changes or a breakpoint hits.

In this section, we discuss components of a message-based back-end. Together with a composite front-end, we could provide a mechanism of composite debugging for grid or web services. In the next section, we give some experimental results of an implementation, called vega service debugger (*vdb*).

5. Vega service debugger

In the *vdb*, the source-level front-end is based on specifications of java virtual machine and java™ platform debugger architecture, which use the java debugger wire protocol for communications [5]. The message-based front-end is built on the same service-oriented architecture, and could call a system

debugging service to inspect a remote service. Because of using different specifications and connected protocols, debugging activities could be also divided into two levels clearly. Using the *vdb*, a programmer could trace behaviors of a debugged service in the normal container. Like inspecting activities as found in source-level debuggers, the executing context can be observed and modified through non-intrusive instrumentation. It allows for multi-programmers online inspecting their service with same environments.

Our latest prototype of the *vdb* has supported to debug grid or web services within Axis-java v1.x, Vega GOS v2.0, and GT4 v4.0.1 platforms [3], [4], [11], in both message-based and source-level debugging.

5.1. Experimental results

In the following discussion, we inspect a web service, *echo*, of Web Services-Axis 1.2, running on the Apache Tomcat servlet container [1]. The container address is "*http://10.61.0.67:8080*". The *echo* service deployed in "*/axis/services/echo*" has a group operations including, *echoString, echoInterger, echoStringArray, echoStruct, echoNestedStruct, echoMap, echoMapArray* etc., to transmit different types of received data back to a client invoker. All related java codes and service descriptions could be found in the release package. The system debugging service for non-intrusive instrumentation is deployed in "*/axis/services/VegaDebugService*".

The following code can give a simple client program of a test case for invoking the *echoString* operation.

```
public static void main(String[] args) {
    try {
    String endpoint =
       "http://10.61.0.67:8080/axis/services/echo";
    /* get the argument. */
    String str = args[0];
    /* Construct new client service. */
    Service service = new Service();
    Call call = (Call) service.createCall();
    call.setTargetEndpointAddress(
       new java.net.URL(endpoint));
    /* set the operation name. */
       call.setOperationName(
          new QName("http://soapinterop.org/",
          "echoString"));
       /* invoke the operation with a string object.*/
    String ret = (String) call.invoke(
       new Object[] {str});
    System.out.println("Sent"+str+", got '"+ret+"'");
  }
  catch (Exception e) {
    System.err.println(e.toString());
  }
}
```

With inserting a handler for the Self-ID generator in the client-config.wsdd file and a handler for the behavior controller (*org.ict.vdebug.arch.MLDebugHandler*) in the service-config.wsdd file, we can inspect the service in a message-based debugging mode. Some results for debugging the service are illustrated in Fig. 4.

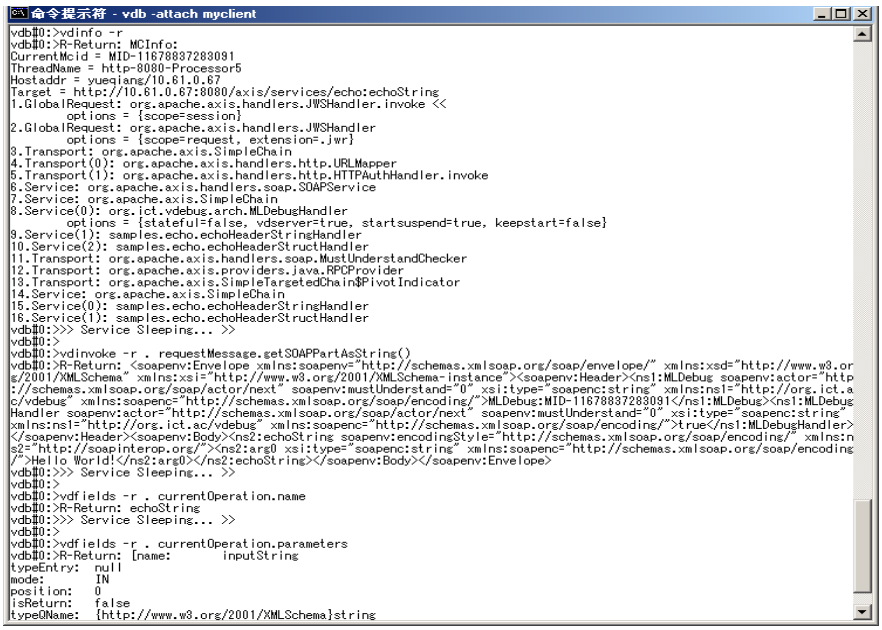

Figure 4. A screenshot for debugging the echo service with vdb

With the *vdinfo –r* command line, remote behaviors can be traced at a breakpoint. The trace consists of the unique identifier with *CurrentMcid*, the remote thread name (*ThreadName*) in the container, the host address (*HostAddr*), the target service with a operation name (*Target*), and following a group of all behavioral descriptions with indexes. Each item of behavioral descriptions includes a scope, e.g., *GlobalRequest*, *Transport*, and *Service*, a class name, and zero or more option(s) for the behavior. The current behavior is marked by "<<".

In message-based debugging, context elements can be observed and modified through some debugging activities. For example, a remote SOAP message can be dumped with the command line:

vdinvoke –r . requestMessage.getSOAPPartAsString(),

where, *-r .* means to use the default address and unique identifier, and the following expression consists of a method name (*getSOAPPartAsString*) declared in the *requestMessage* class, which is a member of the default inspected context, *MessageContext* class. We can find that its body encapsulates the "Hello World!" argument for the *echoString* operation. We can also observe the operation name in a *currentOperation* field with the command line:

vdfields –r . currentOperation.name,

and its parameters, with:

vdfields –r . currentOperation.parameters.

All the results are given in Fig. 4, within a vdb debugging console.

In fact, above message-based debugging activities inputted from the *vdb* console invoke related operations of a system debugging service, which call corresponding functions in non-intrusive instrumentation. It shows that the implementation makes our mechanism suited to inspecting web services in a normal container.

6. Conclusions

A major area of concern in the usability of the debugger described here relates to its composition. Certainly composite issues concerning message-based and source-level debugging have been broached. Equally important, if not more, is the issue of this tool's usability for debugging a sequence of services in normal containers. A solution we are looking at moves the focus of source-level debugging towards behavior debugging based on the message interaction. The idea is to capture not only all behaviors in a debugged service, but in addition context elements for debugging activities. A debugging migration, in this scenario, would inspect only one behavior if needed. In this way the behavior could be effectively inspected as part of a service and debugged in isolation.

Acknowledgments

This work is supported in part by the China Ministry of Science and Technology 863 Program (Grant No. 2006AA01A106), and the China National 973 Program (Grant No. 2003CB317000 and No. 2005CB321807), and the Omii China (Grant No. 2005AA119010).

References

[1] M. P. Singh, M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, West Sussex, England, 2005.

[2] F. Berman, G. Fox, T. Hey, *Grid Computing Making the Global Infrastructure a Reality*, John Wiley & Sons, West Sussex, England, 2003.

[3] Apache Software Foundation, Web Services-Axis, http://ws.apache.org.

[4] Z. Xu, W. Li, L. Zha, H. Yu, D. Liu, *Vega: A Computer Systems Approach to Grid Computing*. J. Grid Comput. 2(2), pp. 109-120, 2004.

[5] *Java Platform Debugger Architecture Documentation*, http://java.sun.com/products/jpda/, 2004.

[6] D. Neri, L. Pautet, S. Tardieu, *Debugging distributed applications with replay capabilities,* In Proceedings of the conference on TRI_Ada'97, St. Louis, Missouri, USA, 189-195, 1997.

[7] P. Maes, *Concepts and Experiments in Computational Reflection*, In Proceedings of the ACM Conference on Object-Oriented Programming System Languages and Applications, Orlando, Florida, USA, 147-155, 1987.

[8] M. Fahndrich, M. Carbin, J. Larus, *Reflective Program Generation with Patterns,* In the Fifth International Conference on Generative Programming and Component Enginerring (GPCE'06), Portland, Oregon, USA, 275-384, 2006.

[9] *Java Core Reflection*, http://java.sun.com/j2se, 1998.

[10] Q. Yue, Z. Xu, W. Li, *Message-based Instrumentation for Service Debugging*. Submitted the International Journal of Web Services Research, 2006.

[11] Globus Alliance, *Globus Toolkit*, http://www-unix.globus.org/toolkit.