A Model of Message-based Debugging Facilities for Web or Grid Services

Qiang Yue, Xiaoyi Lu, Zhiwei Xu, Haiyan Yu, Li Zha
Key Laboratory of Network Science and Technology
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, 100190, P.R. China
({qiangyue, luxiaoyi, zxu, yuhanyan, char}@ict.ac.cn)

Zhiguang Shan
Department of Information Technology
State Information Center of the People's Republic of China
Beijing, 100045, P.R. China
(shanzg@mx.cei.gov.cn)

***Abstract*—Message-based debugging facilities for Web or Grid Services are separated from an infrastructure of source level debugging and can work in a self-identifying and coexisting mode within a normal services container. In this paper, we discuss problems for services debugging and approaches we take. We present the operational model and context inspection of message-based debugging facilities. The facilities are able to trace service behaviors, dump debugging information, and manage states and behavioral breakpoints of debugged services. This model supports a mechanism of multi-user and multi-site service debugging without requiring programmers or developers to one by one duplicate full scenarios in multiple servers.**

Keywords-Debugging facilities, mutual limitation, coexistence, breakpoint, mapping space, behavior trace, context expression, System Debugging Service.

I. INTRODUCTION

Web Services are autonomous, platform-independent computational elements that can be described, published, discovered, orchestrated and programmed to bind networks of collaborating applications distributed within and across organizational boundaries [1]. By well-defined interfaces and specific conventions, Grid Services are concerned, particularly with behaviors related to the management of transient service instances [2]. Generally, Web Services defined by W3C, are stateless. Grid Services, on the other hand, are stateful [1-2].

Specifications are put in place for major aspects of Web or Grid Services. Most SOAP engines follow these common standards and protocols, e.g., in Web Services-Axis, Vega GOS, and Globus Toolkit [3-5]. As Service-Oriented Computing has emerged as a cross-disciplinary paradigm for distributed computing, a need for tools and techniques to build and debug such software applications has grown. However, it still lacks suitable facilities for service debugging.

Debugging techniques have evolved over the years in response to changes in programming languages, implementation techniques, and user requirements. A new type of implementation vehicle for software has emerged that, once again, requires new debugging techniques [6]. Several papers on theoretical and practical aspects of new debugging and monitoring have appeared in the literature [6-11]. These research efforts have succeeded in creating a vast and valuable body of knowledge and demarcating what is feasible from what is not.

In particular, a need often arises for the focused, on-demand debugging of a certain service, where the focus of one or more users may want to inspect a message interaction, a single behavior, a sequence of services. Essentially, the message processing provides a framework to support messaging or service behaviors. Such a framework prompts the need for more effective approaches to trace these behaviors, and dump service contexts during problem execution.

The focus of this paper is on modeling message-based debugging facilities in a normal service container. To gain an understanding of important issues addressed in the paper and approaches we take, consider a scenario that two users want to debug grid services deployed in two containers. We discuss problems with a source level debugger and give our solution of message-based debugging. Then we discuss ideas of mutual limitation, coexistence, and service context. Mutual limitation is applied to identify a debugged service and a debugging activity. Coexistence means the facilities do not violate a run-time service container and not interfere with other services on it. Behavior descriptions as labels are taken to trace a debugged service in the operational model of the facilities, which are programming language-neutral. The service contexts consist of structural run-time data and persistent resources for message processing and management for a service invocation. We give design and implementation of the facilities that provide capabilities to trace behaviors, dump debugging information, and manage execution states and behavioral breakpoints of a debugged service. The facilities are separated from a source level debugging infrastructure and can work in a self-identifying and coexisting mode within a normal service container.

In [12], we have given some debugging modes and structure of debugging Web or Grid Services. In this paper we want to clearly give a formal description of message-based debugging facilities in the debugging back-end. The rest of this paper is organized as follows: Section 2 presents a case study to debug grid services, followed by a more general analysis. In Section 3, we discuss mutual limitation with a debugged service and a debugging activity and give

978-0-7695-3708-5/09 $25.00 © 2009 IEEE
DOI 10.1109/SERVICES-I.2009.20

an operational model of message-based debugging. Section 4 then deals with context inspections for multiple users in multiple containers. The design of message-based debugging facilities are given in Section 5. We introduce related work in Section 6 and conclusions with future work in Section 7.

II. CASE STUDY

Consider the following scenario. Two users want to debug the grid services deployed in two containers. To use a source level debugger, a service container must run on a debugged mode, in which the debugging infrastructure works, and only one user can connect or attach to it at one time. This procedure for debugging a single grid service can be shown as Fig. 1.

In order to debug a GT4 service [3], a container must be executed on a debugged Java Virtual Machine (JVM) and Java Debugger should connect the JVM at first. Then *CounterClient* in the client side starts to call *CounterService* at the debugged container. After these preparations, one can use the debugger to remotely inspect the whole JVM on which the service runs.

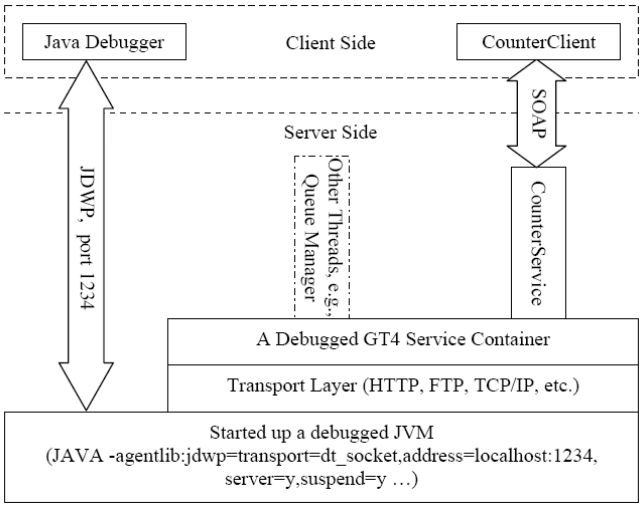

Figure 1. The procedure for debugging a single grid service

When programmers debug the services in the classical debugging mode, there are several problems existing. First and foremost, one user could not simultaneously debug services run on a debugged container connected or attached by the other. For example, *Service2Service* in *Container1* invokes another service *EmbeddedService* in *Container2*, shown as Fig. 2. In order to avoid mutual conflicts, *User1* could not debug *EmbeddedService* that is run on *Container2* connected by *User2*. Similarly, *User2* could not debug *CounterService2* on *Container1* connected by *User1*.

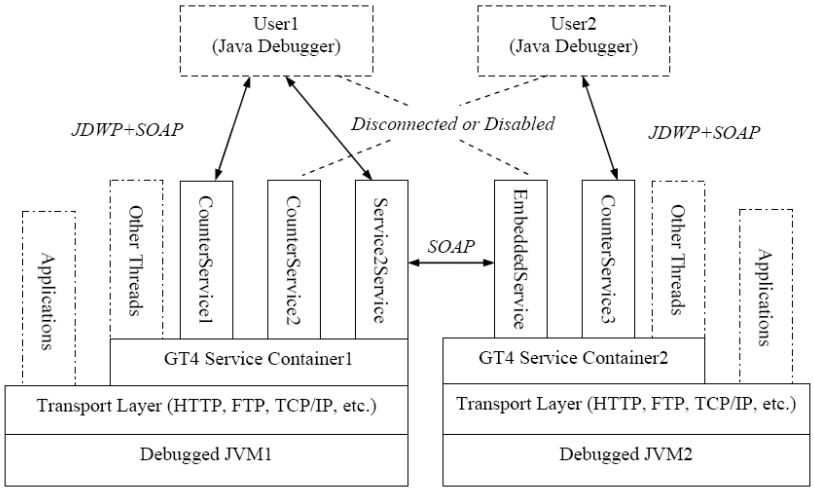

Figure 2. An example for service debugging disabled with the Java debugger

Besides, since a Java debugger must control the whole debugged JVM as a service container, the second problem is such a debugging mode may interfere with other java threads, services run on the JVM, e.g., *Queue Manager*. It may also affect execution states of other applications on the JVM.

Last but not the least, for a testbed of Service-Oriented Computing, such a debugging mode could not start upon with a normal run-time container. Even if the container can restart with a debugging mode, the run-time environment would be destroyed. That is, one may imperatively turn on the debugging infrastructure for debugging interactions.

Therefore, to tackle these problems and make conventional debugging techniques and tools easily applying to service debugging, we takes an approach to a composite mode, both message-based and source level debugging [12]. In our facilities, service debugging can be organized on behaviors of message processing. It checks consistency between the observed behaviors and the expected behaviors. Focusing on these behaviors allows programmers to take different debugging strategies in understanding and communicating the expected behaviors of a service, especially in a normal container. Moreover, a debugging migration could apply to a single behavior if necessary. In this way the behavior could be effectively inspected as part of a service and debugged in isolation. A message-level debugging example that two users could simultaneously debug services deployed in two containers is illustrated in Fig. 3.

Both the message-based front-end and back-end are built on the same architecture to support Web or Grid Services. A message-based debugging command request calls System Debugging Service (SDS), the API of the debugging back-end, to inspect a debugged service remotely. The message-based debugging mode is independent of a source level debugging infrastructure, so that other non-debugged services and applications can be executed normally and continuously in the JVM. In this mode, a user could break and resume a debugged service invocation, and trace the behaviors in a normal run-time container. Like inspecting activities as found in a source level debugger, some contexts can be observed and modified. Our implementation allows multiple users online to inspect their services in run-time environments without mutual interference [12].

As all above, this paper safely draws the conclusion that a separation of debugging infrastructure, at the source level, and message-based debugging facilities, at the service level, is necessary to fully support service debugging. This separation leads to a model of message-based debugging facilities for Web or Grid Services, which is discussed in following sections.

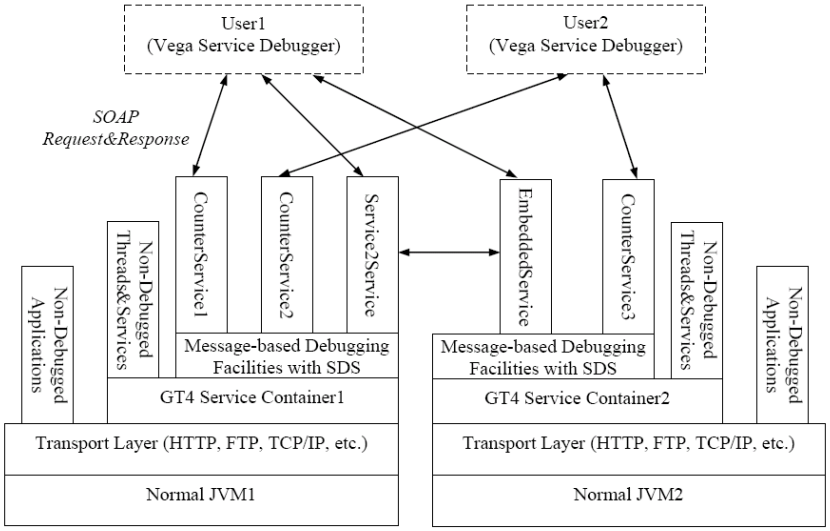

Figure 3. An example for service debugging with Vega Service Debugger

III. OPERATIONAL MODEL

A command line input from a debugging console means to perform a debugging activity. Two important characteristics of message-based debugging are coexistence and self-identification. Coexistence makes message-based debugging facilities work in a normal service container. Self-identification denotes that a debugged service could determine itself and a debugging activity inversely controlling it.

A. Debugging Request

In a message mode, for a set of debugged services (D) and a set of system debugging services (S) with a group of operations (O), the structure of a command line ($cmdl$) can be defined as [12]: For $\forall(d,s)\in D\times S$, $\exists o\in O, O\in s$, $cmdl=(nm,<addr,uid>,Expr)$, where, nm is the name of a debugging command ($nm= name(o)$), $addr$ is the address of a system debugging service (s), uid is the unique identifier of a debugee (d), and $Expr$ is the expression to represent a

target object including the empty for a default one or if not needed.

For service debugging, message interactions of a debugging command can be processed by a uniform method across proper abstractions. This could format a request to a System Debugging Service through the standardized protocols for Web or Grid Services without other more specific requirements. Therefore, we have the definition of a debugging request.

Definition 3.1. In a message-based debugging mode, the format of a debugging command request can be defined as: *C* = (*Sa, Op, Po*), where,

- *Sa*: an endpoint address of the system debugging service;
- *Op*: a description of the operation, *Op* = (*NS, nm*), with
  - *NS*: a naming space of the operation;
  - *nm*: an operation name;
- *Po*: parameters of the operation, *Po* = (*uid, Expr*), with
  - *uid*: a unique identifier for the debugged service;
  - *Expr*: an optional expression for the debugging command request.

For a debugging command request, the endpoint address (*Sa*), operation (*Op*) and unique identifier (*uid*) are used to detect a debugged service from other services in a container. That is, the *uid* and *Sa* identify a debugged service and the *Sa* and *Op* do a debugging command request or activity.

Theorem 3.2. (Mutual Limitation) In a normal runtime container, a debugged service identifies itself not only as controlled by a debugging activity but also as determining the activity. That is, for a debugee (*d*) and a related debugging activity (*a*), $\exists uid$ (a self-identifier), then $d \xleftrightarrow{uid} a$, where the $\leftrightarrow$ symbol means "mutual limited by a self-identifier".

Proof. According to the definition 3.1.

In a container, the *uid* identifies a debugee and a debugging activity, e.g., a request from debugger to continue it, or one to dump debugging information. In other words, a debugee itself and a set of related debugging activities are in mutual limitation. All behaviors of a debugee lie in the assertion of its identifier. To be sure, the identifier must be generated from an original client caller for an on-demand debugging mode, and the debugged service that is identical with the very existence of the same. With this limitation, a debugee and a debugging activity could recognize each other.

Therefore, message-based debugging facilities could coexist with debugged services in the system. Any debugger that follows this is, by definition, a coexisting debugging system, in which the structural correspondence substantially holds. Since an interactive debugging procedure does not interfere with other services running on the same system, users could simultaneously debug their service invocations.

B. Operational Semantics

In a normal service container, the operational model of debugged services can be seen as labeled transition systems $(B, S, \rightarrow)$, where *B* is a set of labels, *S* is a set of execution states, and $\rightarrow \subseteq P(B \times S)$ is a transition relation. Therefore, we have following propositions: behavioral step, behavioral breakpoint, state and breakpoint mapping space, self-contained trace.

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 $s' = s''$. Note that the behavioral step is much weaker than requiring a debugged invocation to be deterministic. Any transition will satisfy it after some simple local transformations at each behavior in the invocation 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 shows that from a $< b, s >$ there can be many possible behavioral steps. In practice, a behavioral step could consist of one or more conditions that determine when a service invocation should be interrupted.

2) State and Breakpoint Mapping: In a normal service container, all execution states for debugged services could be mapped into a state mapping space (*SMS*) in message-based debugging facilities with their own identifiers. It can be also denoted as: for all these debugged services (*D*), $\forall d \in D$, $\exists SMS = \sum_{i=1}^{n} \{uid_i, S_i\}$, then all $S_i \mapsto SMS$, where, *n* is the count of simultaneously debugging services in a container, the $\mapsto$ symbol means "map into". Similarly, a set of behavioral breakpoints ($BP_i \in B$) could also be mapped into a breakpoint mapping space (*BMS*). That is for all these debugged services $(D), \forall d \in D$, $\exists BMS = \sum_{i=1}^{n} \{uid_i, BP_i\}$ then all $BP_i \mapsto BMS$.

With the mutual limitation, a behavioral breakpoint could satisfy to capture a single control flow for a debugged service invocation. Such a breakpoint is self-identifying, and hits in the only sequence of behaviors of the invocation. In a service container, many services may invoke the same behavior, e.g., one in the scope of a global request or response flow; thus the mutual interference may be avoided if setting a breakpoint in the behavior with a self-identifier.

IV. Context Inspection

Service contexts include structural run-time data and persistent resources in the message processing for a service invocation. In the message-based debugging mode, debugging information is built on top of a current message context and a set of service behaviors which make a framework to support the service execution. We can give a general definition of service contexts as follows.

A. Service Context

1) Message Context: For a service invocation, the structure of a message context can be defined as: *MC* = (*DSO, RRM, RPE, AR*, M*$_0$), where,

- *DSO*: descriptions of a service invocation and its operation;

- *RRM*: the data structure of the current request and response messages. A message contains following parts: the main part, attachments, and object's methods, thus *RRM*=(*R, A*, M1*), with
  - *R*: the message structure of a request or a response, e.g., SOAP message
  - *A**: zero or more attachments used for behaviors or services;
  - M_1: a group of members to deal with *R* and *A**. These members include some properties, fields and methods, which provide basic identifying data and interfaces of messages.
- *RPE*: a local reference for the execution engine. An engine provides common functions for the procedure, e.g., service registries, behavioral description maps, and class loading properties;
- *AR**: null or a set of actor roles. Actor roles are associated with an execution of service behaviors, which are invariant during the message processing;
- M_0: a group of members to handle *DSO*, *RRM*, *RPE* and *AR** in a message context. In the message-based debugging mode, a method can be invoked by a debugging command with correct arguments.

In addition, we should manipulate various contexts dynamically built or constructed through the current message context. Message-based debugging facilities should also support inspecting such resource contexts if necessary. For example, persistent resources with a stateful service should be exchanged between the local and remote sides, when a behavioral emigration occurs for single step-in [12].

2) Resource Context: For a service invocation, a resource context can be defined as: $RC = (MC, Cons, M_{rc})$, where,

- *MC*: a message context defined as above;
- *Cons*: a constructor of a resource context with the current message context;
- M_{rc}: a group of members defined in a resource context.

Therefore, in a normal container the related service contexts for a debugged service can be denoted as:

$C = MC \cup (\sum RC)$.

As the following discussion, all debugged services could establish a mapping of those contexts into a group of global maps in a service container.

B. Global Maps

In a container, assume that a one-level map could contain all references of current message contexts for debugged services whose keys are the self-identifiers. For a debugged service, message-based debugging facilities can make a reference of its current message context into a global map (*GMAP*1). By calling a system debugging service, it is easy to locate a current message context (*mcobj*) with a self-identifier parameter (*uid*). The global map in a service container can be denoted as:

$$GMAP1 = \sum_{i=1}^{n} \{uid_i, mcobj_i\} \quad (1)$$

where, *n* is the count of programmers for simultaneously debugging services in a container. Then a well-format expression (*Expr*) within a debugging request can be computed in the context by an expressional evaluator (see also section 5.3).

For a resource context, some debugging command requests could construct or save a context object (*rcobj*) into a two-level global map, where it is mapped with a *uid* key and a specific name (*sname*) assigned to it, denoted as:

$$GMAP2 = \sum_{i=1}^{n} \{uid_i, \sum_{j=1}^{m_i} \{sname_{i,j}, rcobj_{i,j}\}\} \quad (2)$$

where, *n* is the same one of map (1). To the i^{th} debugged service, m_i is the count of resource contexts temporally mapped. For such a context inspection, an operation of the system debugging service can locate an object with two steps by the identifier and specific name. It first searches the sub-map with a key (*uid*), where the related context object is mapped with a name (*sname*) if exists, and then locates it.

Arguments for a debugging command request, which calls a method of the context, can be orderly buffered to a list in a service container. They could be set with one or more debugging commands before a method called. Indeed, they are automatically cleared after the call completed. Similarly, in a container, a global map (*GLST*1) for temporally buffering argument lists (*alst*) does also a one-level map with unique identifiers (*uid*), denoted as:

$$GLST1 = \sum_{i=1}^{n} \{uid_i, [alst_i]\} \quad (3)$$

where, *n* is the same one of map (1) and (2).

Theorem 4.1. (Inspective Satisfiability) In a message-based debugging mode, for a group of programmers (*P*), there exists a uniform space to map simultaneously inspected contexts in a service container. That is, for a programmer, $p \in P$, $\exists MS$, then $C \mapsto MS$, where the $\mapsto$ symbol means "maps into".

Proof. In a service container, references for inspected message context objects of concurrent debugged services can be mapped into a global map: $GMAP1 = \sum \{uid, mcobj\}$, and the inspected resource context objects into a global map: $GMAP2 = \sum \{uid, \sum \{sname, rcobj\}\}$, and a temporal argument's list for called methods into a global map: $GLST1 = \sum \{uid, [alst]\}$. Therefore, the facilities can establish a mapping of the simultaneously inspected references and objects in a container with a space:

$MS = (GMAP1 \cup GMAP2) \times GLST1$.

Hence, for $\forall p, \exists MS$, then inspected service contexts, $C \mapsto MS$. □

Lemma 4.2. (Inspective Reachability) In a message-based debugging mode, for a sequence of debugged services within one or more service containers or sites, there exists a product mapping space that could establish a mapping of debugged service contexts for inspections, denoted as:

$$PS = \prod_{k=1}^{l} MS_k$$

or $PS = \prod_{k=1}^{l}((GMAP1_k \cup GMAP2_k) \times GLST1_k)$,

where, l is the count of service containers. Service contexts related to these debugged invocations should be mapped with a unique identifier and/or an assigned name into this product mapping space.

Proof. According to formulae (1)-(3), theorem 3.2 and theorem 4.1, it is easy to prove. □

Therefore, in message-based debugging facilities, all the contexts for debugged services in one or more containers ($\forall c \in C, \exists C \mapsto MS, MS \subseteq PS$) can be independently inspected or dumped without a source level debugging infrastructure.

C. Context Expression

The context expression (*Expr*) is another element within a debugging command line. It is a parameter of a debugging command request. Such an expression is often an appropriate presentation of a reflective member in the inspected context. In message-based instrumentation, a reflection (*r*) for the element (*m*) in a service context (*c*) can be represented with an expression (*Expr*). That is, $\forall m \in c, c \in MS$, $\exists Expr$, then $Expr \uparrow m_c^{\ r}$, where $\uparrow$ symbol means "can represent".

A context expression can be considered as a high level description of a reflective context element. The relationship between them is a similar one as that between an interface and a particular implementation. Consequently, the expression can encapsulate its implementation and can also be decomposed with some inspective functions of the facilities. Typically, in an object oriented language with reflections (e.g., Java, C#, Smalltalk), we can query a context instance for its class with reflective APIs provided by the core of a system.

For example, the java reflection consists of a set of programming interfaces through which the software module in a java system can discover the structure of classes, methods and their associations in the system [13-14]. In the java programming language, reflective capabilities are centered around the class *java.lang.Class*, extended by the core reflection package, *java.lang.reflect* [13]. All classes, including the *Class* itself, are instances of the *Class*. In addition to the *Class*, several other classes that support reflections are defined, e.g., *java.lang.reflect.Constructor, java.lang.reflect.field, java.lang.reflect.Method*. Classes have methods that support introspection. We can query a class for its superclass, superinterfaces, fields, methods, constructors and member classes.

As above, when a debugging command request arrives, the corresponding operation of a System Debugging Service is called, which can locate the service context in a product mapping space and evaluate an expression in a non-intrusive mode based on the APIs of language reflection.

V. DESIGN

Message-based debugging facilities can reply a debugging command request through a system debugging service [12]. In the facilities, a system debugging service provides a channel between the message-based front-end and back-end. With debugging interactions, a behavior controller verifies the message-based debugging mode, manipulates its debugged service at each behavioral boundary, and triggers its migrating unit if needed. Non-intrusive instrumentation provides capabilities dynamically and interactively to trace service behaviors, manage debugging information, and evaluate the value of a context element [12].

A. System Debugging Service

In message-based debugging facilities, operations of a system debugging service are of two types: control, query and update. The control one consists of the following three groups [12]:

1) State Operations: Set or update some execution states for a debugged service, e.g., step-in, next, continue, and waiting.

2) Breakpoint Operations: Insert or delete a behavioral breakpoint, and enable or disable behavioral control of a debugged service.

3) Compensable Operations: Temporally adjust the control structure of a debugged service at a behavioral breakpoint, e.g., skipping, compensating after, and retrying a behavior.

The query and update operations include:

1) Trace Operations: Generate a behavioral trace of a debugged service.

2) Dump Operations: Dynamically evaluate and dump debugging information of a context expression.

Mentioned previously, these operations corresponding to debugging activities can be processed in a uniform mode. The format of each operation contains three parts: an operation name, a parameter of the unique identifier, and an optional expression of a context element. With the mutual limitation, a called operation of the system debugging service first identifies a debugged service, and processes states and data of the debugee. The expression can represent its implementation, the context reflection.

B. Execution State

The states can be divided into two types: internal (Int) and external (Ext). An internal state is set by the debugging engine, including: *Ready*, *Suspended*, *Running(R)*, *Waiting*, *SL Debug*, *Timeout* and *Exit*. An external state is set by a debugging command request or activity, including: *Suspended*, *Waiting*, *Breakpoint*, *Next*, *Cont*, *LStep*, *Step*, *Skip*, *Compafter*, *Retry*, and *Exit*. The explanation of execution states for a debugged service is listed in Table 1.

C. Syntax of Expressions

The service debugger provides a set of inspective commands for message-based debugging. We can use them to examine elements of a context at a behavioral breakpoint [12]. The syntax of expressions for inspecting context elements is given in Fig. 4.

Notes: The *construct* option denotes to construct a context object with the current message context if needed. The *asarg* is used to save the result object as an argument for invoking the next reflective method. The *save* denotes to save the result object to the context mapping space. The

nomc indicates that the inspected object is not a default current message context. The *fput* or *fget* means to put or get a resource file to or from the service-site.

TABLE I. A LIST OF EXECUTION STATES FOR A DEBUGGED SERVICE

State	Explanation	Int/Ext
Ready	The service behavior is ready	Int
Suspended	Suspend a service	Int/Ext
Running	Execute a service behavior	Int
Waiting	Stop at the current behavior	Int/Ext
Breakpoint	Hit at a behavioral breakpoint	Ext
Next	Go into the next behavior and then stop	Ext
Cont	A waiting service is being continued	Ext
LStep	Step into an immigrating behavior locally	Ext
Step	Step into an emigrating behavior remotely	Ext
SL Debug	Enter a source-level debugging mode	Int
Skip	Skip execution of the current behavior	Ext
CompAfter	Compensate after execution of the current behavior	Ext
Retry	Re-execution of the current behavior	Ext
Timeout	The service is expired	Int
Exit	Terminate the service	Int/Ext

QULExpr→Empty | Variable | QULExpr Dot Variable | QUMark QULExpr
Dot→.
Minus →-
Empty→null
QUEqu→QUMark QULExpr Equal ConstString
QUExpr→QULExpr | QUEqu
Equal→=
QUMark→[*Minus QUType | Minus QUOption*]
QUType→boolean | byte | int | char | long | float | double | string | list
QUOption→construct | asarg | save | nomc | fput | fget
Variable→Identifier

Figure 4. The syntax of context expressions

VI. RELATED WORK

Debugging tools in general have a very long history. The three basic forms of debugging, trace, dump, and break date back to the EDSAC computer of the 1940's [16]. In a survey of on-line debugging technique, Evans and Darley remark that computer use is becoming debugging-limited rather than limited by memory size or processor speed. In this section, we introduce the relevant research results for debugging Grid or Web Services. They can be divided into four broad categories: extension mechanisms of source level debugger, low-level system virtualization, mirror-based technique, manual instrumentation. Here we discuss some instances of the four categories.

A. Extension Mechanisms

In [17], Kurniawan and Abramson proposed a WSRF-Compliant Debugger for Grid Applications. One important component in it is the middleware compatibility layer. It is written according to an extension mechanism of the middleware, which acts as a translation layer between Grid debugging service and generic debug interface. This extension allows a GDB/GDBServer [18] to be utilized as the back-end debug engine. In [19] and [20], the NASA Ames's researchers build a debugger for applications running on heterogeneous computational Grids, called p2d2, which uses the client-server architecture to isolate the platform-dependent code in a debugger server. In the g-Eclipse project [11], it also provides plug-ins to support debugging and program understanding on MPI applications running on Grid sites. Generally, these extension mechanisms depend on some conventional debugging facilities. Moreover, they focus on Grid applications but on the Grid Services.

B. Low-level Virtualization

In [21] and [22], researchers in Computer Laboratory of University of Cambridge propose the pervasive debugging approach for debugging distributed systems, called PDB. Based on the Xen Virtual Machine Monitor to virtualize the system resources of a single machine, PDB can eliminate the probe effect, and can reproduce the exact behavior of a distributed system. By using Xen to virtualize Grid resources, one can deterministically debug Grid applications. However, this debugging technique is difficult to attain for applications that need to be run on a large-scale distributed system.

C. Mirror-based Technique

In [13] and [14], designers and developers in Sun Microsystems propose the Java Platform Debugger Architecture (JPDA) based on mirror-based systems. All these interfaces are subtypes of the interface *com.sun.jdi.Mirror*. A mirror is always associated with a particular virtual machine as a whole, in which the entities being mirrored exists, so the implementation of a virtual machine often requires command line options to load the debugging agent or infrastructure at first. For debugging Grid or Web Services, more elaborate facilities have appeared with some systems, but the source level debugger still mainly provides only technique for accessing the dynamic state of an entire container. Therefore, programmers are continuing to debug their service applications in their original or demo machines, but a testbed even or multiple servers in different geographical locations.

D. Manual instrumentation

Often software programmers trace the execution of their programs by manually inserting instrumentation statements that may print out code locations and values of one or more variables. Thus a program can generate its own trace that can be inspected to understand or verify its behavior as it executes on the actual environment or platform. One approach to manually instrumentation at source code is those inserted statements to dump logging information, e.g., logging services [23], and some programming languages support program annotation, e.g., Java and CLR. Code annotations constitute a powerful mechanism that enables passing information between programmers, tools, and the runtime, from the source code level up to the execution time [24]. Manual instrumentation and annotation to measure a

program should be done when one gathers profiling information by inserting code to track various quantities. The presence of such code can often influence the execution speed and introduce imprecision in the measurement.

VII. Conclusion and Future Work

A major area of concern in a model of message-based debugging facilities described here relates to service debugging in Internet-based environments. Certain issues concerning service debugging have been broached. Equally important, if not more, is the issue of this model for dynamically inspecting a sequence of services without violating normal service containers. A solution we are looking at moves the focus of a source level debugging mode towards a message-based debugging mode. The model supports a mechanism of multi-user and multi-site service debugging without requiring programmers or developers to one by one duplicate full scenarios in multiple servers.

We consider several directions for future research. First, learnt from some conventional debugging techniques and tools, we want to improve our debugger with some extended functionalities, including remote event handling, just-in-time behavior debugging. Secondly, we will investigate to debug composite or orchestrating business process and multi-language applications within Service-Oriented Architecture, e.g., non-intrusive and automatic message instrumentation for workflow and business applications, supporting for other types of reflective programming languages. Thirdly, we plan to integrate our debugger into a grid-programming environment for debugging services in large-scale distributed systems.

Acknowledgment

This work is supported in part by the National Science Foundation of China (Grant No. 90412010, 60873243, 60673054), the Hi-Tech Research and Development (863) Program of China (Grant No. 2006AA01A106, 2006AA04Z158, 2008AA01Z140).

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] Globus Alliance, Globus Toolkit 4, http://www-unix.globus.org/toolkit.

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

[5] 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.

[6] C. Jeffery, J.D. Choi, R. Lencevicius, Proceedings of the Sixth International Symposium on Automated & Analysis-Driven Debugging. ACM Press, California, USA, 2005.

[7] L. Baresi, C. Ghezzi, S. Guinea, Smart Monitors for Composed Services. In Proceedings of the Second International Conference on Service Oriented Computing (ICSOC'04), New York, USA, pp. 193-202, 2004.

[8] M. Pellegrino, CLR Debugging: Improve Your Understanding of .Net Internals by Building a Debugger for Managed Code. MSDN Magazine, Microsoft Corporation, Nov. 2002.

[9] X. Zhang, R. Gupta, Y. Zhang, Effective Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. IEEE International Conference on Software Engineering, pp. 502-511, Edinburgh, UK, 2004.

[10] A. Zeller and R. Hildebrandt, Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 28(2), pp. 183-200, 2002.

[11] C. Klausecker. T. Kockerbauer, R. Preissl and D. Kranzlmuller, Debugging MPI Programs on the Grid using g-Eclipse. In Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, HLRS, Stuttgart, pp.35-45, 2008.

[12] Q. Yue, Z. Xu, H. Yu, W. Li, An Approach to Debugging Grid or Web Services. In Proceedings of IEEE 2007 International Conference on Web Services (ICWS'07), Salt Lake City, Utah, USA, pp. 330-337, 2007.

[13] G., Bracha, D., Ungar, Mirrors: design principles for meta-level facilities of object-oriented programming languages. In Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vancouver, BC, Canada, pp. 331-344, 2004.

[14] T. Lindholm, F. Yellin, The Java Virtual Machine Specification. Addison-Wesley, 2nd edition, 1999.

[15] M. Campbell-Kelly, The Airy Tape: An Early Chapter in the History of Debugging. IEEE Annals of the History of Computing, 14(4), pp. 16-26, 1992.

[16] T. Evans, D. Darley, On-Line Debugging Techniques: A Survey. In Proceedings of the Fall Joint Computer Conference 29, pp. 37-50, 1966.

[17] D. Kurniawan, D. Abramson. A WSRF-Compliant Debugger for Grid Applications. In the 21st International Parallel & Distributed Processing Symposium (IPDPS 2007), Long Beach, California, pp. 1-10, 2007.

[18] Free Software Foundation inc. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[19] R. Hood. The p2d2 Project: Building a Portable Distributed Debugger. In Proceedings of SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96), Philadelphia, Pennsylvania, USA, pp. 127-136, 1996.

[20] R. Hood, G. Jost. A Debugger for Computational Grid Applications. In Proceedings of the 9th Heterogeneous Computing Work (HCW2000), pp. 262-270, 2000.

[21] P. Barham, et al. Xen and the Art of Virtualization. In Proceedings of the 9th ACM Symposium on Operating Systems Principles, Session: Virtual Machine Monitors, Bolton Landing, NY, USA, pp. 164-177, 2003.

[22] R. Mehmood, J. Croecroft, S. Hand, S. Smith. Grid-Level Computing Needs Pervasive Debugging. Grid 2005 - 6th IEEE/ACM International Workshop on Grid Computing, pp. 186-193, 2005.

[23] S. Gupta. Pro Apache Log4j, Second Edition (Pro). Apress Berkley, CA, USA, 2005.

[24] M. Biberstein, et al. Instrumenting Annotated Programs. In Proceedings of the First International Conference on Virtual Execution Environments (VEE'05), ACM SIGPLAN, Chicago, Illinois, USA, pp. 164-174, 2005.