Extensible Symbolic Debugging for Distributed Object Systems

Giuliano Mega[1] **e Fabio Kon**[2]

[1]Dipartimento di Ingegneria e Scienza dell'Informazione
Università degli Studi di Trento – Trento, TN – Italia

[2]Departamento de Ciência da Computação
Universidade de São Paulo (USP) – São Paulo, SP – Brasil

mega@disi.unitn.it, kon@ime.usp.br
http://www.ime.usp.br/~giuliano/dissert.pdf

Abstract. *After over thirty years of distributed computing, debugging distributed applications is still regarded as a difficult task. While it could be argued that this condition stems from the complexity of distributed executions, the fast pace of evolution witnessed with distributed computing technologies has also played its role by shortening the lifespan of many useful debugging tools. In this Masters thesis summary, we briefly summarize our incursion in building an extensible tool which puts distributed threads and symbolic debuggers together, resulting in a simple and useful debugging tool/technique. The tool is extensible and backed by a technique supported by features that are common to synchronous-call middleware implementations, making them suitable candidates for surviving technology evolution.*

1. Introduction

Debugging distributed applications can be a remarkably difficult task. While it is true that part of these difficulties stem from the inherent complexity of distributed executions, both the diversity and the fast pace of evolution of distributed computing technologies (including hardware, operating systems, and middleware) have played a prominent role in making the lifespan of debugging tools extremely short. Indeed, we are not the first ones to identify heterogeneity as a major contributing factor to the slow progress witnessed with mainstream debugging tools. Hondroudakis had done it before [Hondroudakis 1995], as had Cheng [Cheng and Hood 1994], and many other researchers before us. Be as it may, the end result is the noticeable lack of a set of useful, effective debugging tools, even on mainstream technology and middleware platforms.

The main contribution of this work is the research, design and implementation of a **simple**, **useful**, and **portable** debugging tool. This debugging tool is supported by a simple instrumentation technique, and can be applied to a large and important class of distributed systems – the distributed object systems. One of the main characteristics of our debugging technique is that it is conceived to run on top of features commonly present in Distributed Object Computing (DOC) middleware, making it suitable for heterogeneous environments, and resilient to technology evolution. Other relevant contributions of this work include a formal characterization for *distributed threads* (DTs)

[Mega and Kon 2007] (discussed in Sec. 2), and a comprehensive survey, in Portuguese, on existing tools and techniques for debugging of concurrent systems [Mega 2008].

The rest of this text presents a short summary of the main contribution of this work – our debugging tool and its associated technique. In order to better motivate our particular choices and design, however, it helps to turn to the history of computing. From a historical perspective, one of the most popular abstractions for interprocess communication in distributed systems has been that of the remote procedure call (RPC) [Coulouris et al. 2005]. So much that, in the late eighties, Andrew Tanenbaum published an article [Tanenbaum and van Renessee 1987] which criticized the "holly cow" status attained by the model within the distributed operating system research community, in an allusion to its indiscriminate employment. RPCs have been widely employed for over two decades, either in a procedure-oriented fashion or, somewhat more recently, as remote method invocations in object-oriented middleware systems such as CORBA [Object Management Group 2004], Java/RMI [Sun Microsystems 2004] and many others. Although RPCs and Distributed Objects have been largely criticized in recent years (e.g. [Vinoski 2008]), the model still enjoys significant popularity [Birman 2004], and systems built on top of these technologies still represent a highly relevant class.

On the debugging side, symbolic or source-level debuggers represent perhaps one the most popular and reinvented debugging concept of all times. We believe this popularity can be explained by the fact that symbolic debuggers do a very good job at bridging the low-level mental images that developers form in their minds while coding with the actual execution of their systems, and also because this debugging concept came to life very early in the history of computing [Evans and Darley 1966]. Apart from a kind of "cognitive appeal", therefore, symbolic debuggers are a *de facto* standard: these tools represent the only kind of debugging aid, apart from the print statement, which is available for almost every language, operating system, and hardware platform currently in existence.

The gist of this work lies in how we put together this highly relevant class of distributed systems – RMI-based Distributed Object Systems – and those highly popular debugging tools – symbolic debuggers – in such a way that our requirements of portability, simplicity, and usefulness can be satisfied.

2. Distributed Threads

Distributed threads (DTs) are at the core of our debugging concept. To understand what they represent, recall that RMI-based Distributed Object Systems work by providing an illusion, to clients, that objects residing in distinct machines actually share one single addressing space [Coulouris et al. 2005]. One of the crucial points of this illusion is the fact that method invocations performed on remote objects appear to clients as if they were local. Particularly, DOC middleware provides an abstraction of a "virtual" thread that can cross machine boundaries. At the implementation level, this effect is achieved through the use of *proxies* [Gama et al. 1994] on the client side, which act as a bridge between application code and middleware. The intuitive notion behind a DT is illustrated in Fig. 1 (a). The DT is depicted as a thread that can cross the network and reach a remote object.

Fig. 1 (b) shows the mechanism that actually provides this illusion. In a typical Distributed Object Computing (DOC) middleware implementation, the client-side thread performs a method invocation on a proxy object (1) that has the same interface as the

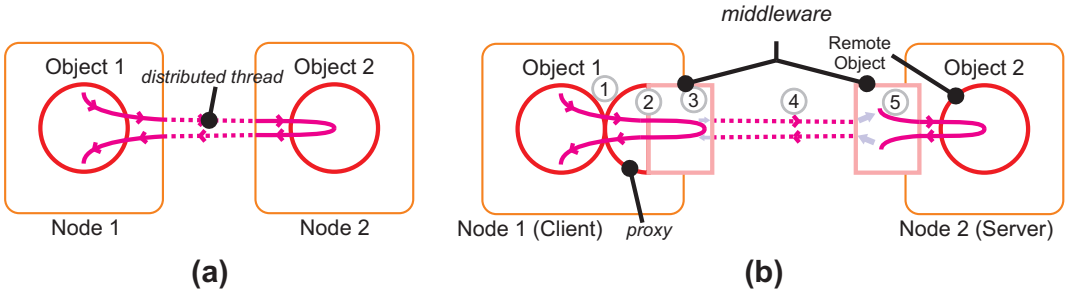

Figure 1. (a) Intuitive notion of a DT and, (b) typical mechanism for implementation.

remote object it is mimicking. This proxy redirects the call to the middleware (2), which then takes care of dispatching it to the appropriate server on the network. Upon receiving a request, the server-side middleware component locates the correct remote object, and assigns a local thread to service the received call. Finally, it performs the adequate request on the remote object on behalf of the client (5). An important piece of this mechanism is that, in most DOC middleware implementations, the client-side thread remains blocked (3) while the server processes the request, thus emulating the transfer of control that would actually occur if remote and local objects were in fact residing in the same address space.

Now note that, although the DT in Fig. 1 spans only two machines, it could well span more if the second node made a remote call to a third server. We will discuss the general structure of a DT in the context of the process of obtaining its "virtual call stack". Fig. 2 shows a DT spanning three nodes. At each of these nodes, the DT is mapped by the middleware into some local thread, represented in the picture as l_1, l_2, and l_3. These threads are logically part of the same DT T, so we say that they **participate** in T. Local threads are represented in Fig. 2 together with their respective call stacks (1),

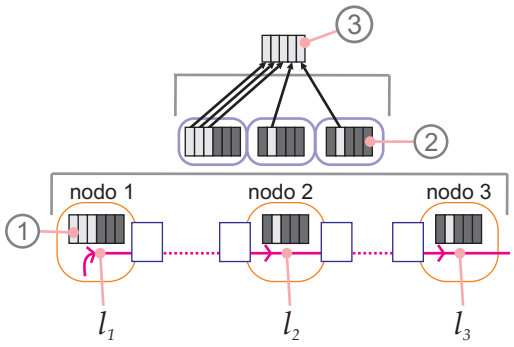

Figure 2. Assembly of a virtual call stack.

with each little rectangle representing an activation record (stack frame). Lighter gray frames represent calls into application code, whereas darker gray frames represent calls into middleware code. Note that this essentially means that a DT T can be seen as a sequence of local threads $\{l_1, l_2, l_3, ...\}$. Now note that this sequence changes in time: at each instant t, a DT T can be mapped into a possibly different sequence $\{l_1, ..., l_n\}$ of local threads. Also, for $1 \leq i < n$, l_i, is a local thread that is blocked, waiting for a remote request that is being handled by server-side local thread l_{i+1} to complete.

Finally, note that it is possible that some thread l_i is **both** servicing a request from

l_{i-1} **and** blocked waiting for the remote request being serviced by l_{i+1} to complete. In fact, this is the case for all local threads in the sequence, except for l_1 (which is blocked, but is not servicing any requests) and l_n (which is servicing a request from l_{n-1}, but is not blocked). We call the sequence of local threads that *participate* in T at time t a **snapshot** of T at t. That said, the "virtual call stack" of a DT T at an instant t can be produced by taking the call stacks of all threads in the *snapshot* of T at t (represented by (2) in Fig. 2), and gluing them together. Since we are interested in abstracting away the middleware, we strip the darker gray stack frames out and get to the call stack depicted in (3) in 2.

This is just a high level overview of the formalization we have developed on what DTs are and how we expect them to behave. Due to space constraints, we will not present it here, but rather refer the interested reader to one of our publications [Mega and Kon 2007], or to the full thesis text [Mega 2008]. This formalization lays down the foundations to our debugging technique.

2.1. Distributed threads as debugging tools

Although DOC middleware provides, within its limitations, a very convenient abstraction for developers, debugging these systems is usually not straightforward. This happens in part by the reasons pointed by Rosenberg [Rosenberg 1996]; namely, that debugging support trails systems development. Meaning here that vendors are remarkably effective at pushing out new technology while delaying, or even neglecting, debugging support. Some remarkable examples of this behavior can be found in the literature [Cargill and Locanthi 1987]. At the same time, as we already hinted to in Sec. 1, vendor-specific tools, when provided, fail to be useful on heterogeneous environments where systems are built on top of multiple platforms. This leads many developers to turn to the simple, well-known, and widely available symbolic debuggers, or even the pervasive print statement. In any case, there are several shortcomings to using symbolic debuggers with DOC middleware, the most important of which we outline in the following paragraphs.

Abstraction mismatch: Symbolic debuggers are inherently connected to the semantics of the underlying language. Since remote method invocations are extensions to this semantics, they are, almost by definition, out-of-scope for symbolic debuggers. This means that the useful abstractions provided by DOC middleware and RMIs fall apart as soon as the developer steps into the first bit of automatically generated proxy code.

Causal relations: capturing causality [Schwarz and Mattern 1994] is a task that is out of scope for most symbolic debuggers. For DOC middleware, this means that users will not be able to see the order in which events have happened. Also, they will not be able to see which local threads participate in which DTs.

Distributed and self-deadlocks: Like with multithreaded applications, DTs can deadlock when acquiring the same locks in different orders. Distributed deadlocks can be tough to spot with plain symbolic debuggers, as the cause-and-effect implied by the caller/callee relationship is not properly captured [Mega 2008].

Conventional symbolic debuggers present a running system as a collection of local threads with which the user may interact in order to further explore the running states of their systems. A distributed symbolic debugger is a natural generalization to this model: instead of "local" threads, we have DTs, and instead of a single process, we have the whole distributed system. This is precisely what we have designed and built: a debugger

that works like a regular symbolic debugger, where users can suspend, resume, do step-by-step execution, and set breakpoints on DTs. Moreover, it can detect distributed deadlocks, as well as perform some other forms of limited automatic analysis [Mega and Kon 2007].

The usefulness and helpfulness argument for our approach stems from the observation that working at the abstraction level of DTs provides several advantages. First, it enables the user to focus his attention on the state and the control flow of his own application, instead of forcing him to manually extract that information from the merged state of his application and the middleware platform. This is crucial to avoid a phenomenon known as the *maze effect* [Gait 1986], where the programmer is overwhelmed with useless information, making debugging very hard, if not impossible. Second, DTs convey some form of causal information which, although limited, allows us to detect situations such as distributed deadlocks much more easily.

3. A portable distributed symbolic debugger

Our debugger works, in essence, by tracking DT snapshots (Sec. 2). Fig. 3 (a) gives an overview of its architecture. The debugger is composed by **(1)** a set of *local agents*, which are responsible for collecting relevant runtime information, and for interacting with the various processes in the distributed system on behalf of a **(2)** *global agent*. The global agent, in turn, acts as a global observer of the distributed computation, piecing together the relevant runtime information collected by the local agents into an approximation of the distributed execution. The global agent also hosts the user interface, and coordinates the various local agents during interactive processes.

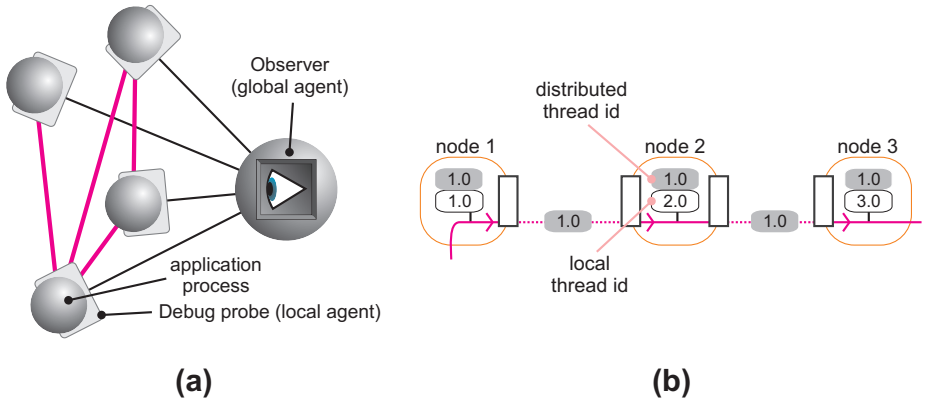

Figure 3. (a) Architecture of our debugger, and (b) ID propagation.

We identify and track DTs by assigning a unique, 48 bit ID to each of the local threads in the distributed system. These IDs are actually composed by a pair (ID_{node}, ID_{thread}) of smaller IDs, where ID_{node} is a 16 bit ID assigned to each node, and ID_{thread} is drawn from a local sequential counter. Whenever a local thread l with ID ID_l initiates a remote call, its ID gets propagated along the call chain, "tainting" all of the subsequent local threads. This is shown in Fig. 3 (b). Note that this means that local threads have actually two IDs: one that identifies the local thread itself, and another one that identifies the DT in which it takes part at a given instant. DTs are tracked by having **local agents** emit tracking events at certain key points in the request processing mechanism of DOC middleware. These key points are exactly the points where a

local thread enters (*server receive*) and leaves (*server return*) a remote object, as they capture precisely the instants when a local thread l starts and finishes handling a remote request (and thus, begins and ends participating in a given DT T). The tracking protocol is illustrated in Fig. 4. These events are propagated from the local agent to the global agent through a language-neutral protocol named *Distributed Debugging Wire Protocol*, or DDWP [Mega and Kon 2007, Mega 2008]. Note that "language-neutral" means that DDWP is not coupled to any specific runtime. The rest of this section will be focused on the local and global agents, presenting in a rather synthetic form our portability argument.

Local agents. Local agents are in fact composed by three distinct parts: a symbolic debugger, a debugging library (deployed together with the application), and a set of interceptors woven into the application. The symbolic debugger is just a conventional debugger that can be operated remotely, such as the Java Platform Debugger, or the well-known GNU Debugger (GDB). Interceptors are inserted into each proxy method, as well as into their remote object counterparts. These interceptors call specific methods in the debugging library whenever a proxy or remote method invocation is performed, allowing us to assign and propagate thread IDs, as well as generate DDWP *server receive* and *server return* events. A high level overview of how it all fits together is given in Fig. 4.

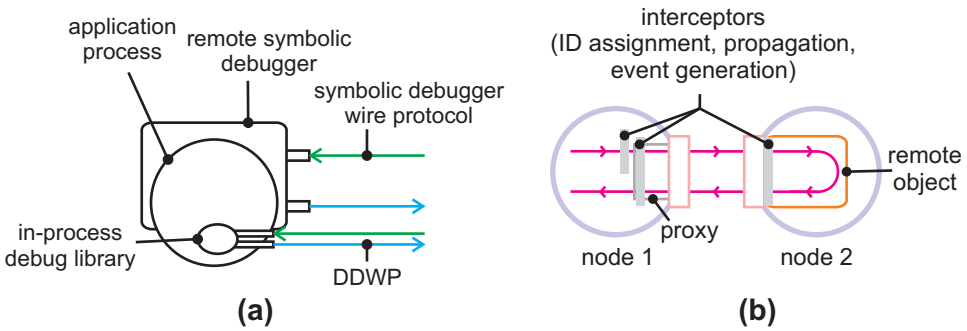

Figure 4. (a) A closer look a the local agent, and (b) interceptors.

The important thing to know here is that the only requirement imposed by the tracking mechanism over the underlying middleware is that it should provide some support for passing metadata with each request. This is a very reasonable requirement, as almost every DOC middleware implementation currently in existence supports it. Also, the only requirement imposed by the tracking mechanism over the underlying programming language is that it should be possible to insert interceptors in each proxy and remote object. We have accomplished this in Java by using bytecode instrumentation [Mega and Kon 2007]. Similar results could be achieved in other languages through the use of reflection or source-code instrumentation. Also, there must be a symbolic debugger available for the language, and this debugger must be remotely operable. Note from Fig. 4 (a) that local agents speak two distinct protocols: the DDWP, which is language-neutral, and a debugger-specific protocol which is used to drive the symbolic debugger remotely. Currently, we provide local agents for Java software running CORBA middleware.

Global agent. The global agent is implemented both as an extension to the Eclipse debugging framework [Wright and Freeman-Benson 2004] and as a regular debugger for Eclipse. Eclipse provides a generic, flexible and language-neutral metamodel for representing entities in a running computation. That said, we essentially implement the

metaphor described in Sec. 2.1 – we provide a debugger implementation that actually aggregates existing debugging clients and, by using the complementary information provided by DDWP events, presents the entire distributed system as a single virtual process, composed by a collection of DTs. Our DTs implement the same Eclipse *IThread* interface as regular threads do, meaning Eclipse sees our distributed debugger as just another debugging client. The architecture of the global agent (in dark gray) and its insertion in the Eclipse framework are presented in Fig. 5. The main point of this architecture is

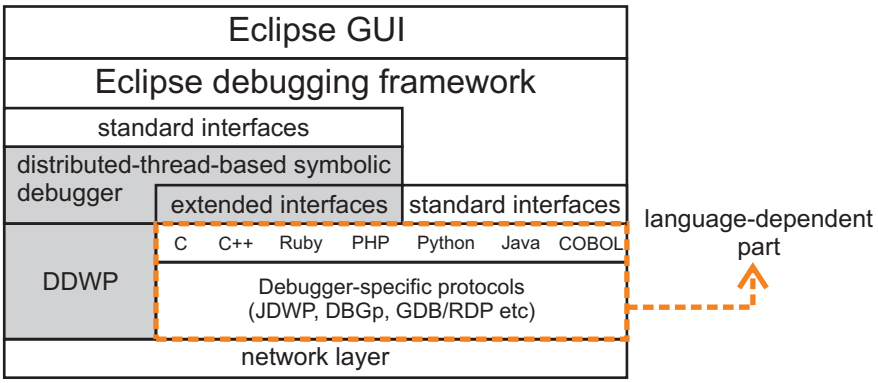

Figure 5. Architecture of the global agent.

that it allows us to add support for new languages by simply extending existing symbolic debugging clients with a set of simple interfaces (extended interfaces in Fig. 5). These interfaces are mostly concerned with registering listeners for some thread-specific events, so that the state of DTs and local threads can be kept consistent, and so that interactive commands issued to DTs can be dispatched to the correct local threads (e.g. if the user decides to suspend a DT, or execute it in step-by-step mode). Also, the only portion in this whole architecture that is coupled to the actual programming languages are the actual language-specific symbolic debugging clients and their communication protocols, but this does not leak to the upper layers, making for a modular and extensible architecture.

4. Conclusion

This paper summarized our Masters research on developing a novel debugger that has been conceived with portability, simplicity, and usefulness in mind. Apart from a highly modular and decoupled architecture, our debugger is backed by a rather simple debugging technique, which requires minimal instrumentation of the target application and imposes very few requirements on the underlying middleware and programming language. Most of the results of this work have been previously published both nationally and internationally: at the 2004 OOPSLA Eclipse Technology eXchange [Mega and Kon 2004], in the 2006 Brazilian Symposium on Computer Networks [Mega and Kon 2006], and in the 2007 International Symposium on Distributed Objects and Applications [Mega 2008]. A demonstration of the initial prototype has been carried out in the tools track of the 2005 Brazilian Symposium on Software Engineering (SBES) [Mega and Kon 2005]. Moreover, both the implementation (free software) and a demonstration screencast of our tool are available at `http://god.incubadora.fapesp.br/portal/screenshots`.

References

Birman, K. P. (2004). Like it or not, web services are distributed objects. *Communications of the ACM*, 47(12):60–62.

Cargill, T. and Locanthi, B. (1987). Cheap Hardware Support for Software Debugging and Profiling. *ACM SIGARCH Computer Architecture News*, 15(5):82–83.

Cheng, D. and Hood, R. (1994). A portable debugger for parallel and distributed programs. In *Proc. of the 1994 ACM/IEEE conf. on Supercomputing*, pages 723–732.

Coulouris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed Systems: Concepts and Design*. Addison-Wesley, 4th edition.

Evans, T. G. and Darley, D. L. (1966). On-line debugging techniques: a survey. In *Proc. of the Nov. 7-10 AFIPS fall joint computer conference*, pages 37–50.

Gait, J. (1986). The Probe Effect in Concurrent Programs. *Soft.: P & E*, 16(3):225–233.

Gama, E., Johnson, R., Helm, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hondroudakis, A. (1995). Performance analysis tools for parallel programs. Technical report, Edinburgh Parallel Computing Centre.

Mega, G. (2008). Depuração Simbólica Extensível para Sistemas de Objetos Distribuídos. Master's thesis, Universidade de São Paulo.

Mega, G. and Kon, F. (2004). Debugging Distributed Object Applications with the Eclipse Platform. In *Proc. of the 2004 OOPSLA Eclipse Technology eXchange*, pages 42–46.

Mega, G. and Kon, F. (2005). GOD: Um Depurador Simbólico para Sistemas de Objetos Distribuídos. Anais Eletrônicos do Salão de Ferramentas do Simpósio Brasileiro de Engenharia de Software de 2005 (SBES'05).

Mega, G. and Kon, F. (2006). Depurando Sistemas de Objetos Distribuídos da Forma que Gostaríamos. In *Proc. of the 24th SBRC*, pages 1131–1346.

Mega, G. and Kon, F. (2007). An Eclipse-based Tool for Symbolic Debugging of Distributed Object Applications. In Meersman, R. and Tari, Z., editors, *Proc. of the 2007 Symposium on Distributed Objects and Applications (DOA'07)*, volume 4803 of *LNCS*, pages 648–666. Springer.

Object Management Group (2004). *The Common Object Request Broker Architecture*.

Rosenberg, J. B. (1996). *How Debuggers Work: Algorithms, Data Structures, and Architecture*. Wiley, 1st edition.

Schwarz, R. and Mattern, F. (1994). Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174.

Sun Microsystems (2004). *Java RMI Specification*. Sun Microsystems.

Tanenbaum, A. S. and van Renessee, R. (1987). A critique of the remote procedure call paradigm. In *Proc. of the 1987 EUTECO*, pages 775–783.

Vinoski, S. (2008). Convenience Over Correctness. *IEEE Internet Computing*, 12:89–92.

Wright, D. and Freeman-Benson, B. (2004). How To Write an Eclipse Debugger. http://www.eclipse.org/articles/Article-Debugger/how-to.html.