Debugging Distributed Object Applications With the Eclipse Platform

Giuliano Mega and Fabio Kon
Department of Computer Science
University of São Paulo
{giuliano, kon}@ime.usp.br

Abstract

Debugging distributed applications is a well-known challenge within the realm of Computer Science. Common problems faced by developers include: lack of an observable global state, lack of a central location from where to monitor possible states, non-deterministic execution, heisenbugs, and many others. There are currently many good techniques available which could be employed in building a tool for circumventing some of those issues, especially when considering widespread middleware-induced models such as Java RMI, CORBA or Microsoft .NET based applications.

In this paper, we introduce an extended symbolic debugger for Eclipse which besides usual source-level debugging capabilities, adds to the abstraction pool a distributed thread concept, central to causality in any synchronous-call distributed object application.

1 Introduction

Debugging a distributed system can be a daunting task. In addition to normal debugging issues (the old isolate-extirpate paradigm), the developer of a distributed system must also cope with the fact that there might be multiple chains of states and events evolving independently and across multiple machines.

To get some minimum insight as to how the execution of the system actually took place (something that is crucial for detecting and isolating bugs), the developer must somehow gather and correlate trace data from the various components of the distributed system, hoping that this approximate view of the execution will be enough for tracking down misbehaviors. If one has to do it all manually, things get even more difficult.

Following the trend of Object Oriented Programming, traditional socket and RPC-based Distributed Systems have evolved over the past few years into modern Distributed Object Systems (DOS). Those are in essence distributed systems which employ some sort of middleware layer that allows the use of object interfaces for publishing and accessing distributed services.

The net effect is that the developer is enabled to think of his distributed system "as if" it were not distributed[1]. Most middleware design efforts are targeted at improving transparency and abstraction. It is only though these powerful and well-known concepts that application developers can build complex distributed object applications – by putting aside the complexity handled by middleware services, one can focus on managing its own application complexity.

Unfortunately, much of the complexity hiding provided by middleware at development time is lost when debugging and, despite the natural pressure to evolve, it seems that debuggers have lagged behind in this area. Traditional symbolic debuggers have been stuck on basic source-level and language-level abstractions for decades, whilst middleware and other similar tools have been continuously extending language abstractions. Even though there are currently many debugging tools available for collecting behavioral information from running distributed applications

OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

[1] Within some reasonable limits.

those are, in general, very different from what the average developer could expect, since they are mostly based on purely monitoring techniques and/or data analysis and bear little or no resemblance to symbolic debuggers. The goal of our project is to extend the Eclipse debugger so that developers can symbolic-debug their distributed applications without losing the abstractions the underlying middleware provides, while applying convenient and solid distributed debugging techniques found in other tools and described in the literature. This should translate into an effective mean of symbolic-debugging distributed applications (something that current tools do not allow) while enjoying all the benefits the Eclipse debugger model provides.

The remainder of this paper is organized as follows. In the following section we present the basis and motivation for our particular choice of semantic extension from the set of possible extensions to the Eclipse symbolic debugger. In Section 3 we underline the basic architecture and current state of implementation of our tool, exposing its current capabilities and explaining a few of its mechanisms. In Section 4 we summarize our results and comment on ongoing and future work.

2 Motivation

There are basically three issues to discuss when considering our current state of work. Those are addressed in the paragraphs that follow.

1. Centralized Presentation: It is a known fact [3,4] that an observed error might be on the tip of a large chain of previously misbehaved or mispredicted events. The difference when considering distributed systems is that this chain of events might span multiple machines, making events difficult to observe and correlate without appropriate tools. Since two distributed executions are hardly equal, if we are to have any hope in finding out the cause of an error then we must somehow capture event information for later reconstruction (postmortem analysis). Though this process of observation is distributed, we want to be able to somehow visualize the complete chains of events. This brings the need for tracing events and forwarding them to a central location where they can be later correlated and presented to the developer, either after his application has ran or while it is running.

By allowing the distributed debugger to contain a central piece at Eclipse from where the developer can easily access collected information we are actually making his life easier as he will not have to jump from machine to machine manually scanning trace files whenever an error occurs.

2. Causality: Another issue that arises in the context of distributed systems (not just distributed debugging) is related to the notion of event ordering. Since most real-world distributed systems have no global clock, unless we find ways of imposing an ordering on collected events we will just end up with a load of scrambled information that tell us nothing about which event happened before which.

Lamport [3] brought this up in his classical paper about logical clocks and causal order, but Lamport's paper discusses message-passing systems and that is where employing middleware changes things a bit. Synchronous call mechanisms such as the one Java RMI and CORBA provide makes remote objects appear as if they were local. This means we effectively have, through middleware and synchronous calls, a local flow-of-control abstraction. In other words, calls made to remote objects transfer the thread of control to callees much like the ones made to local objects do. Since the outcome of a computation in a multithreaded or shared-memory parallel environment is only determined by the dynamic data dependencies formed among concurrent threads at runtime [6], we feel we can work causality on DOSs much like we do in multithreaded systems, provided a few precautions are taken [2].

What we propose is to describe causal relations on DOSs through the dynamic data dependencies formed among concurrent *distributed threads* - hence their importance to our project. Initially, however, we are only interested in tracking *distributed threads* through successive synchronous remote method calls, or the so-called caller/callee relationship [2] (as tracking arbitrary dynamic dependencies would be a lot of work for an initial implementation).

3. Semantics: As discussed earlier, we wish to present the developer with some sort of middleware view of his distributed system. The most notorious abstraction not provided by conventional debuggers is that of a *distributed thread* (see sec. 3.2 for details). This "middleware view" allows the developer to treat his distributed object application "as if" it were a multithreaded application, much like the middleware does.

3 The software

We now present the basic building blocks of our implementation and explain their origins and roles.

3.1 Architecture

Our software is based on a widely adopted architecture for distributed debuggers [8], depicted in Fig. 1.

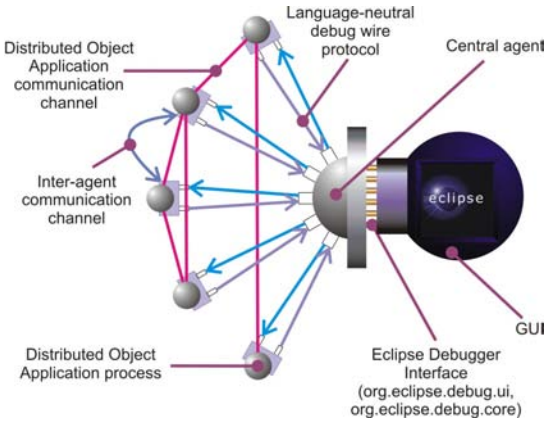

Figure 1: architectural view

Apart from being a requirement to any centralized debugger, this architecture favors decoupling. In Fig. 1, the small grey spheres represent application processes to which are attached local debugging agents, which in turn communicate through a language-neutral wire protocol with a central debugging agent living in Eclipse.

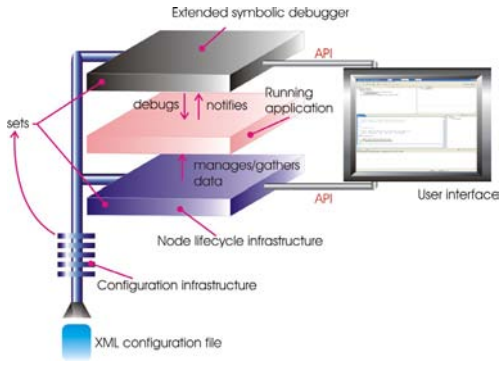

Figure 2: interacting parts

Though our system does not yet reflect this architecture in its plenitude (see Sec. 3.2 for details), our intention is evolving the implementation until it does.

From an software division point of view (shown in Fig. 2), there are four interacting parts that build up our system to which (almost) all classes can be traced to - namely two infrastructure components (node lifecycle and configuration), one debugger component (extended symbolic debugger) and one (could be more) user interface. All components with the exception of a few parts in the extended symbolic debugger (the local agents) run in the same JVM as Eclipse.

Each of those components play a specific role in the debugging process - the configuration infrastructure is responsible for reading information from the various XML-based configuration files, instantiating object proxies, assembling instances and setting up managers that will later on be used to set up nodes and debugger resources.

The node lifecycle infrastructure provides fine-grained control over individual nodes at the process level and allows for data harvesting and interaction (more details in Sec. 3.3). The extended symbolic debugger acts as an omniscient entity that hovers over the running system, monitoring and interacting with it from the outside through a wire debug protocol. The running application is composed of the user application code itself and is included in the picture for clarification purposes only (not part of the debugger).

3.2 Choices and Threads

We have decided to start working with restricted Java/CORBA environments since the Java Platform Debug Architecture [7] offers support for remote debugging by means of a well-defined and established interface (the JDI, or Java Debug Interface), allowing us to skip (most of) the local agent developing efforts as well as the debug wire protocol implementation. Also, our aim at language-independence led us to prefer CORBA over Java RMI at this early stage of our work.

While we tried to build the highest level of the debugger in the most language-neutral fashion we could, our model must yet grow mature before it can accommodate all the organizational differences between the languages we may wish to support in the future (besides C++ and Java).

As mentioned earlier, our primary concern at this point is tracking distributed threads. More than just tracking, we wish to extend the Eclipse symbolic debugger so that it can cope with distributed threads much the same way it does with “normal” threads. That would include, for instance, allowing the user to step into a CORBA call and popping at some remote machine without

further difficulties – "as if" it were a local call.

In order to do so, however, we must first define a distributed thread, then we must map it to the language level and then somehow manage to identify, at any point of a given execution, which local threads map to which distributed threads. This mapping to the language level must be as unintrusive as it can, ideally relying exclusively on the CORBA specification for portability and making use of as little code instrumentation as possible. That said, defining *distributed threads* is easy enough since they are mainly a corollary of the synchronous-call mechanism CORBA (and other middleware) implements. Defining more precisely, all local threads are either distributed threads by themselves or are encompassed by a larger distributed thread to which they are components.

If a local thread is involved in making a (possibly remote) synchronous call, then the thread which services this request at the server-side is part of the same distributed thread. Note that, under normal circumstances, there must be at most one component thread running per distributed thread at any given point in time[2].Therefore, our task for tracking distributed threads consists of tagging local threads with some sort of system-wide identifier as soon as they get created. This identifier, once assigned, must get carried along with the distributed thread across nested call chains of arbitrary depth.

The thread id assignment is accomplished through a classloading instrumentation scheme that modifies all classes implementing the *Runnable* interface. A code snippet containing our custom registration code is inserted at the beginning of each *run* method, causing new threads to register themselves automatically with a local registry (or *tag repository*) when started. This approach does not break even if the user calls the *run* method directly.

The propagation of the thread id, on the other hand, is accomplished at the interceptor level through the use of service contexts, providing for a low-intrusion and portable solution. More details about this mechanism will be given in a few paragraphs. Details concerning interceptors and service contexts can be found in [5]. Our other (highly related) task - enabling the Eclipse debugger to cope with distributed threads – consisted of devising a mechanism for automatic placement of breakpoints at the CORBA servant level whenever a request is made to "step into" a remote object stub[3]. A schematic of our full solution is given in Fig 3.

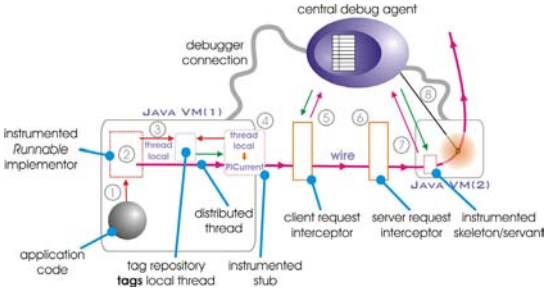

Figure 3: distributed thread tracking mechanism

In Figure 3, the application code requests a new thread, eventually firing an instrumented *run* method and executing our code snippet **(1)**, which automatically registers the newly created thread with a local registry **(2),** and associates it with a system-wide locally-generated thread tag id **(3)**.

If a step request is made into a local stub, the global agent marks the current distributed thread as "remote stepping" and resumes the local component thread, causing it to eventually hit an instrumented stub, **(4)** who then queries the local registry for the current thread tag id and inserts it into a preallocated *PICurrent* slot [5]. The request then resumes its flow until it reaches an installed client-side interceptor **(5),** who is responsible for extracting the current distributed thread id from the *PICurrent* slot and transforming it into a service context that can go through the wire. The client-side interceptor may also communicate with the central agent for non-stepping mode event tracking purposes. As soon as the request reaches the server-side it triggers another interceptor **(6)** who reads the transported context and reinserts it into a *PICurrent* slot, to be accessed later on by an instrumented servant **(7),** responsible for informing the global agent that a distributed thread has reached its boundaries. The global agent then decides whether or not to insert a breakpoint at the servant method and, at the same time, it gains knowledge of where the distributed thread will be passing next (for tracking purposes). When the breakpoint is finally hit **(8)** the central

[2] We could consider that a safety property.

[3] We actually invented a new stepping mode, the "step remote", which works just like "step into" for local object calls but jumps to remote machines whenever it detects that the local object is also a CORBA stub.

agent works together with the runtime instrumented skeleton **(7)** to avoid possible race conditions. The central agent then resumes step mode at the servant object method boundary, giving an illusion to the user sitting in front of Eclipse that he has "stepped into" a remote object.

3.3 Node Lifecycle

Our system also implements a node lifecycle infrastructure that allows for flexible management and data harvest from remote processes. It allows the user to configure, via XML, multiple ways of launching remote JVMs as well as providing facilities for on-the-fly harvesting of text data from remote processes standard output (stdout) and remote error output (stderr). One of the main objectives of this infrastructure is providing the user with means for easily setting debug scenarios and controlling remote processes (launching and then taking down nodes for simulating failures, for example). It is useful for automated testing and could assist in cyclic debugging (for an explanation of cyclic debugging you can refer to [6]). We currently support those features in integration with the GNU Secure Shell Client and plain rlogin.

4 Conclusion and Future Work

We have taken the first step towards building an extensible symbolic distributed debugger for Eclipse which levels middleware abstractions and allows the developer to think the same way while debugging and developing. Our work is novel in the sense that it allows the live tracking of distributed threads and also in the sense that it provides an infrastructure for easily setting distributed debugging scenarios, including capabilities for launching and killing remote processes, simulating node failure and communicating with remote processes I/O. The plugin source code is available for download at http://eclipse.ime.usp.br/projects/DistributedDebugging.

This small step is part of a much more ambitious project – there are still many unclosed gaps and many other distributed debugging issues left unhandled. Our main focus shall shift from now on to issues like improving non-intrusiveness, seeking better ways of instrumenting classes, implementing distributed predicate detection [1] and, finally, distributed execution replay. Also, Eclipse integration is still very limited and must be handled properly above all else.

Acknowledgements

This work is supported in part by an Eclipse Innovation Grant from IBM and by a grant from CAPES-Brazil.

About the Authors

Giuliano Mega is a graduate student and **Fabio Kon** is an Assistant Professor, both in the Department of Computer Science at the University of São Paulo.

References

[1] I. Tomlinson and V.K. Garg. Detecting Relational Global Predicates in Distributed Systems. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*. pages 21-31, USA.

[2] J. Li. Monitoring and Characterization of Component-Based Systems with Global Causality Capture. *23rd ICDCS*. Providence, Rhode Island, May 19-22, 2003.

[3] L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM,* 21(7) :558-565, July 1978.

[4] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. Workshop Parallel and Distributed Algorithms*, Elsevier Science Pub., Amsterdan, 1989, pages 215-226

[5] Object Management Group. *The Common Object Request Broker : Architecture and Specification*, Revision 3.0.3, March 2004.

[6] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. Workshop on Parallel and Dsitrbuted Debugging*, pages 1-10. 1993.

[7] Sun Microsystems. *The Java Platform Debugger Architecture*. http://java.sun.com/products/jpda/index.jsp

[8] IBM Distributed Debugger : Overview. http://web.ccr.jussieu.fr/ccr/Documentation/Calcul/usr-share/html/idebug/en_US/concepts/cbcddovr.htm