An Eclipse-based Tool for Symbolic Debugging of Distributed Object Systems*

Giuliano Mega[1] and Fabio Kon[1]

Department of Computer Science, University of São Paulo, Brazil
{giuliano, kon}@ime.usp.br
http://god.incubadora.fapesp.br

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 life-span of many useful debugging tools. In this paper we present an extensible Eclipse-based tool which brings distributed threads and symbolic debuggers together, resulting in a simple and useful debugging aid. This extensible tool is based on a technique that is supported by elements that are common to synchronous-call middleware implementations, making it a suitable candidate for surviving technology evolution.

1 Introduction

After over thirty years of distributed computing, debugging distributed applications is still a difficult task. While it is true that this could be partially blamed on the fact that distributed executions are complex and difficult to handle, a major contributing factor to this situation has been the fast pace at which new distributed computing technologies (including hardware, middleware, and operating systems) have emerged, making the life-span of debugging tools somewhat short. Be as it may, the net result is the noticeable lack of a set of effective, widely adopted debugging tools, even on mainstream middleware platforms.

We are not the first ones to identify heterogeneity as a major contributor to the slow progress witnessed with mainstream debugging tools. Cheng had already identified it in 1994 [2], and so had Pancake [15], as well as many other researchers and industry specialists. This points out to the fact that portability - not just among hardware platforms, but also among middleware and operating systems - is of paramount importance if a tool is to be useful within today's highly heterogeneous environments, and also if this tool is to remain useful for more than a couple of seasons. One way to achieve portability is through standardization. In the High-Performance computing arena, there have been some rather important efforts - like the High Performance Debugging (HPDF) forum,

* This work was partially supported by a grant from CAPES-Brazil, by an IBM Eclipse Innovation Grant, and by a Google Summer of Code grant.

the OMIS project, and the Parallel Tools Consortium - which attempted to push the development of a set of standards for debugging and performance analysis tools. These efforts were mainly targeted at the needs of the high performance computing community, however, and, to the best of our knowledge, no such efforts have ever been attempted within the distributed object community. This puts distributed object application developers in a difficult situation as far as compatibility, longevity and, by consequence, availability and usability of debugging tools is concerned.

In this paper, we present a new framework for distributed debugging that can be applied to multiple middleware platforms and programming languages. The framework has been developed as an Eclipse plugin, which allowed us to reuse the rich debugger user interface provided by the platform. This framework is validated through a Java/CORBA instantiation that supports features such as distributed application launching, breakpointing, distributed stack and state inspection, multiple extended stepping modes, and distributed deadlock and stack overflow detection, among other features.

Motivation: Distributed Object Middleware and Symbolic Debugging

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) [13]. 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, DCOM, Java/RMI and .NET/Remoting.

In an equal ground as far as popularity is concerned, we have the symbolic or source-level debuggers (like GDB [17], for instance). Symbolic debuggers have been around for almost as long as higher-level languages themselves, and still constitute one of the most widely used and accepted tools for helping programmers remove bugs from programs. We believe that this popularity can be explained by the fact that symbolic debuggers are the only kind of debugging tool, apart from the print statement, that is available for almost every language, runtime environment, operating system, and hardware currently in existence. In a sense, we could say there is a standard at work here - albeit not a formal one. Distributed applications (including Distributed Object Applications) are, in fact, frequently debugged with symbolic debuggers capable of operating remotely.

Our approach to the distributed debugging problem attempts to unite the essence of what makes synchronous calls and symbolic debuggers so convenient. At the same time, we try to augment the latter with functionality so that they can become more useful in the context of multithreaded, distributed object applications (DOAs), while keeping in mind the goals of extensibility, portability and simplicity. The philosophy that actually backs this whole work is already quite simple - we want to bring symbolic debugging of distributed object applications as close as possible to the debugging of centralized, multithreaded applications. We also want to support the user of Distributed Object Computing (DOC) middleware in accomplishing debugging tasks that result from the

insidious complexities of synchronous calls; that is, we wish to help the user of DOC middleware to overcome some of its inherent complexities. That said, symbolic debuggers are convenient for the following reasons:

1. Ubiquity: it is very rare to see a language development toolkit shipping without a symbolic debugger – this is even more true with mainstream languages. It is therefore not unreasonable to take for granted that the languages and runtime environments on top of which a heterogeneous distributed object system will be built on will have symbolic debuggers available for them.

2. Cognitive appeal: the visualization mechanism used by symbolic debuggers - animation over the source code - is quite intuitive, and matches the (low-level) mental images that the programmer produces while coding. Although this visualization mechanism does not scale particularly well, it is something developers are familiarized with, and it is definitely helpful.

Synchronous-call DOC middleware present the distributed system as a collection of virtual threads that are capable of spanning multiple machines. We will call these *distributed threads* (or DTs). Symbolic debuggers and the underlying execution environment, including its libraries, however, are myopic to such high-level constructs. This brings on a number of issues (some of which are discussed in [10]), which we present in the following paragraphs:

1. Abstraction mismatch: middleware platforms allow developers to treat remote and local objects similarly. Middleware implementations accomplish this transparency by replacing the implementation of remote object references accessible from clients with code that is generated automatically, either during compilation or at runtime. The problem happens when such an application is subjected to the eye of a symbolic debugger, as all of this automatically generated code will be exposed to the user, together with the code of the middleware platform, defeating the benefits of transparency. Not only that, but the user will also not be able to track the flow of control of his application (by stepping) into remote objects like he does with local objects, simply because symbolic debuggers are not aware of this arrangement. In other words, the view of the underlying execution environment that is provided by the debugger does not match the abstract view provided by the DOC middleware.

2. Causal relationships are not properly captured: capturing causality [16] is a task that is out of scope of 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 distributed threads.

3. Distributed and self-deadlocks: Like with multithreaded applications, distributed threads can deadlock when acquiring the same locks in different orders. Distributed deadlocks can be tough to spot with plain symbolic debuggers. Also, although a distributed thread logically represent a single thread, it is actually composed of multiple, "local" threads. Since the concurrency mechanisms of the underlying execution environment are normally oblivious to the existence of distributed threads, this may lead to situations where a distributed thread deadlocks with itself (we call that a *self-deadlock*).

Besides the issues already mentioned, there are a number of other issues that constitute classical problems in distributed debugging [7]. For the sake of completeness, we outline these as follows:

1. Non-determinism and the probe effect: distributed executions are intrinsically non-deterministic due to their partially ordered nature [12]. The interleaving of instrumentation code with application code for gathering of runtime information may lead to timing distortions which affect (and bias) the partially ordered, distributed execution. This is known as the probe effect [5] and may lead to "heisenbugs" (erroneous behavior that disappear under observation).

2. Maze effect: the maze effect [4] manifests itself whenever the user of a debugging tool is overwhelmed with information. Representations of the distributed execution that are too fine grained are one common cause. Tools that are unable to selectively display execution information using some relevance criteria, thus overwhelming the user with data, are another cause.

We believe synchronous calls and symbolic debuggers to be a good starting point for three main reasons: their popularity among developers, their pervasiveness among middleware platforms/programming languages, and because the resulting tool – a distributed symbolic debugger – is something most developers will be familiar with, even though they might have never seen or used one. The source code for all implementation efforts described in this article can be obtained as free software at http://god.incubadora.fapesp.br.

2 Debugging With Distributed Threads

This section begins by attempting to describe in more precise terms *what* are distributed threads (DTs), and by laying a formal framework that tells how we expect them to behave. We then proceed by describing how DTs can help us cope with a number of debugging issues, and present a general idea of how a tool that explores them works.

Before that, however, we would like to reach an informal agreement on what a non-distributed – i.e., a "local" – thread is. A local thread is a "regular" thread. Local threads are restricted to a single addressing space and to a single processing node. Examples of local threads include traditional lightweight processes, such as those implemented at the kernel level in operating systems like GNU/Linux and Microsoft Windows, for instance, as well as heavyweight processes and "green" (non-OS, usually non-preemptive) thread implementations. This informal definition should be enough for our purposes.

For the sake of simplicity, we assume time to be a continuous and linear set of instants that is isomorphic to the set of real numbers $\mathbb{R}$ (i.e., we represent time instants as real numbers). For every distributed computation C, we assume that there are two real-valued instants t^s_C, t^d_C that represent the instants in which C begins and ends, respectively. We say that $[t^s_C, t^d_C]$ is the *lifetime interval* of C. We call L_C the set of all local threads that ever took part in C. Since C has a lifetime interval, all local threads in L_C also have lifetime intervals. Therefore, we can assign a pair of real-valued time instants t^s_l and t^d_l to each local thread

$l \in L_C$, such that these values represent the instants when local thread l starts and dies, respectively, and $[t_l^s, t_l^d] \subseteq [t_C^s, t_C^d]$.

Definition 1 (Distributed Thread Snapshot). *A distributed thread snapshot over a distributed computation C is defined to be a sequence $s = \{l_1, ..., l_m\}$ of local threads, where $l_i \in L_C$ ($i \in \mathbb{N}$ and $1 \leq i \leq m$).*

By analogy to L_C, we will define S_C to be the set of all snapshots that can be formed with threads drawn from L_C. We are now ready to define a distributed thread.

Definition 2 (Distributed Thread). *Let C be a distributed computation. A distributed thread T is a sequence of snapshots $\{s_{t_1}, ..., s_{t_n}\}$, where each t_i ($1 \leq i \leq n, i \in \mathbb{N}$) represents a real-valued time instant ($t_i \in \mathbb{R}$), and all s_{t_i} are drawn from S_C. Also, if $i < j$, then $t_i < t_j$; that is, T is totally ordered with respect to time. For every distributed thread $T = \{s_{t_1}, ..., s_{t_n}\}$, the following properties must hold:*

1. *There exists a local thread $l_1 \in L_C$ such that $s_{t_1} = \{l_1\}$, and l_1 is the first element of s_{t_i}, for all i. We will say that l_1 is the* **base** *of distributed thread T.*
2. *Let $i \in \mathbb{N}$ and $1 < i \leq n$, and let $s_{t_{i-1}} = \{l_1, ..., l_m\}$. Then, in the absence of failure, exactly one of the following must hold:*
   (a) *$s_{t_i} = \{l_1, ..., l_m, l_{m+1}\}$, where $l_{m+1} \in L_C$. In this case, l_m have initiated a remote request at instant $t_{i-1} + \delta \leq t_i$ (where $\delta \in \mathbb{R}$ and $\delta > 0$), and thread l_{m+1} have begun handling this remote request at instant t_i.*
   (b) *$s_{t_i} = \{l_1, ..., l_{m-1}\}$. In this case, thread l_m has finished handling, at instant t_i, the remote request that had been previously initiated by l_{m-1}.*
3. *Let t_T^s and t_T^d be the instants in which T starts and dies, respectively. Then $t_1 = t_{l_1}^s = t_T^s$ and $t_n < t_{l_1}^d = t_T^d$.*

Let $s_{t_i} = \{l_1, ..., l_m\} \in T$. We say l_m is the *head of* T at instant t_i. We also say that the threads $l_1, ..., l_m$ *participate in* T at instant t_i. Definition 2 has a number of interesting implications. First, for every local thread $l \in L_C$, we have a distributed thread T such that **(1)** T starts and dies with l, and **(2)** l is the base of T. Second, DT snapshots can be interpreted as follows. Let $s_{t_i} = \{l_1, ..., l_n\}$, **(1)** if $1 < i \leq n$, then l_i is handling a request that has been initiated by l_{i-1}, and **(2)** if $i < n$, then l_i is blocked in a remote call that is being handled by l_{i+1}.

We shall call a snapshot that contains a single local thread a *trivial snapshot*. All distributed threads begin with a trivial snapshot, and may contain several trivial snapshots (all identical) along its snapshot set. Fig. 1(a) shows a DT and part of its snapshot set (relevant state shifts) as it progresses through a three-node call chain. Lastly, our notion of a valid set of DTs shall be bound by a rule we call "the single participation rule". This rule, as we will see, shall constrain the class of middleware implementations that our technique applies to. In order to make the definition less complicated, we will define two auxiliary concepts, expressed in Def. 3 and Def. 4. What Def. 3 says is that the state of a distributed thread T, at an arbitrary instant x, is either empty (if x falls off the lifetime interval $[t_T^s, t_T^d]$ of T) or it corresponds to the last snapshot of T up to instant x.

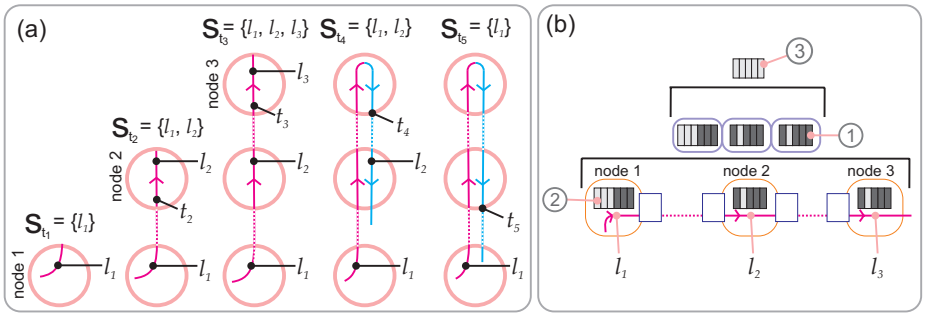

Fig. 1. (a) DT and its sequence of snapshots during part of a three-node call chain. (b) Assembly of the virtual stack from a snapshot.

Definition 3 (State of a Distributed Thread). *Let D_C be the set of all distributed threads that ever existed in C and let $T = \{s_{t_1}, ..., s_{t_n}\} \in D_C$ be one of these distributed threads. Also, let S_C be defined as before. The state of a distributed thread T at instant $x \in \mathbb{R}$ is given by the function $f : D_C \times \mathbb{R} \rightarrow S_C$ where:*

$$f(T,x) = \begin{cases} s_{t_i}, \text{ if } t_i \leq x < t_{i+1} \text{ and } 1 \leq i < n, \text{ or} \\ \quad t_i \leq x < t_T^d \text{ and } i = n \\ \emptyset, \quad \text{otherwise} \end{cases}$$

Definition 4. *Let s_1 and s_2 be two distributed thread snapshots. We say that s_1 is a subsnapshot of s_2, or that $s_1 \rightarrow s_2$, if and only if s_1 is a suffix of s_2.*

The "single participation rule" attempts to establish the situations in which it is valid for a local thread l to participate in the state of more than one distributed thread, simultaneously. Mainly, we would like to express that a local thread l cannot simultaneously participate in the state of more than one distributed thread, except under some very special circumstances. These circumstances will be characterized with the help of the following remark:

Let $s_{t_i} = \{l_1, l_2, ..., l_n\}$ be a nontrivial snapshot of a distributed thread T. Then s_{t_i} can be expressed as $\{l_1\} \cup f(T', t_i)$, where T' is the distributed thread whose base is l_2. That is $f(T', t_i) \rightarrow f(T, t_i)$.

To give a more concrete example of the meaning of this remark, let T be a distributed thread, and suppose $f(T,t) = \{l_1, l_2, l_3\}$ at some instant $t \in \mathbb{R}$. The remark shows that, if we take a local thread $l_2 \in f(T,t)$, then this local thread participates – simultaneously – in the states of all distributed threads that can be formed by removing prefixes of size less than 2 from $f(T,t)$. In our example, l_2 will participate in $f(T,t) = \{l_1, l_2, l_3\}$ and in $f(T',t) = \{l_2, l_3\}$, where T' is the distributed thread whose base is l_2 – hence l_2 participates in the state of more than one distributed thread simultaneously. Those will be the only situations where it will be allowable for a local thread to participate in the state of more than one distributed thread simultaneously. Therefore:

"Single Participation Rule": Let $\{s_1, ..., s_n\}$ be the set of snapshots in which a local thread l participates at an instant t. The single participation rule is obeyed if we can find a permutation π of $\{1, ..., n\}$ such that $s_{\pi(1)} \rightarrow ... \rightarrow s_{\pi(n)}$.

The work described in this paper applies, in general, to distributed computations whose set of DTs conform to the single participation rule. Though most middleware implementations do produce compliant executions, some real-time ORBs [8] may not – but then again, symbolic debuggers are usually regarded as being far too intrusive for real-time systems. DTs play an important role in the achievement of our goal, because they make the necessary link between RMI-based distributed object systems and symbolic debuggers. The whole idea behind a symbolic debugger that leverages DTs is that it may present the execution as a collection of states on DTs, instead of on loosely-coupled local threads.

A more practical view of a distributed thread – as it is displayed by our debugger – is shown in Fig. 1(b). The depicted snapshot **(1)** is composed of three local threads (l_1, l_2, and l_3), which are represented along with their call stacks **(2)**. The darker stack frames represent calls into middleware code, whilst the lighter frames represent calls into user code. The debugger strips out the darker frames, assembling and presenting to the user a virtual stack **(3)**, which is composed of user code only. The debugger then allows the user to interact with this "virtual" thread as he does with regular threads – stepping, resuming, suspending, inspecting, etc. – in debuggers such as GDB [17].

3 A Distributed-Thread-based debugger

Now that we have presented our motivation and characterized in precise terms how the distributed threads we intend to deal with should behave, we will present the actual tool we have built, that leverages them for debugging.

3.1 Representing and Tracking Distributed Threads

The first step to presenting the distributed system as a collection of distributed threads is being able to track them. There are a multitude of possible approaches to the problem, but those mostly vary between how much of the distributed thread representation will stay at the debugger side (meta-level) and how much will stay at the debuggee side (base-level). Our approach begins by constructing a representation of the distributed threads that is accessible at the base-level.

This representation is inspired on the way distributed transactions are typically identified, and also by the work of Li [9]. In our approach, each local thread that participates in a distributed computation is uniquely identified by a two-part, 48 bit id. The first 16-bits uniquely identify the node to which the local thread belongs to. The remaining 32-bits are drawn from a local sequential counter. 48 bit ids allow us to identify enough nodes/local threads while keeping overhead small, but the actual id length can be tuned. Whenever a local thread that is part of a single trivial snapshot initiates a chain of remote calls, its id is propagated, "tainting" all subsequent threads in this call chain, as shown in Fig.

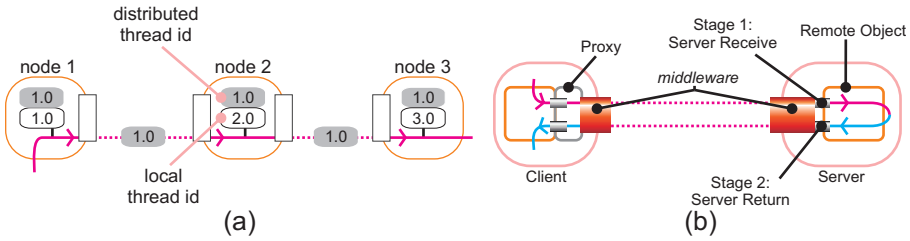

Fig. 2. (a) Propagation of the id of a local thread, tainting subsequent threads and characterizing a distributed thread. (b) Two-stage tracking protocol.

2(a). As for the actual tracking of distributed threads, we just have to remember that for each snapshot, there is always a single head. The remaining threads are all blocked, and, if we assume a failure-free scenario (we will lift this restriction in a moment) we can expect that these blocked threads will not produce any "interesting" changes until they become heads again – that is, until the current head finishes handling the remote request that has been initiated by the local thread that immediately precedes it in the current snapshot. More precisely speaking, local thread "starts handling a remote request" when the middleware calls into the remote object code for the first time during the handling of this remote request, causing a frame, which we will call the *entry frame*, to be pushed into the call stack of the server-side, local thread. Based on this information, it is not hard to come up with a simple tracking protocol that can be used to reconstruct the trajectory of a distributed thread. Our protocol, shown in Fig. 2(b), is composed of two stages: **(1) Server Receive** and **(2) Server Return**, which are triggered when an *entry frame* is pushed, and popped, respectively. Both stages capture the id of the distributed thread, and the id of the local thread.

3.2 Interactivity and Synchronicity

Our intention is to extend a symbolic debugger so that it may handle distributed threads as naturally as it handles local threads. Symbolic debuggers are on-line [7], interactive debuggers by nature, and their ability to operate (e.g., view, suspend, step, resume, inspect) on threads is amongst the most fundamental ones. The problem with interactive operations is that they operate on "live" entities. Debuggers, however, act as observers of computations – they will merely reconstruct an approximation of the state of the application based on information that is sent from instrumentation probes (*local agents* in Fig. 3(a)) that are deployed with the application. Therefore, there is always the chance that the state of the execution as observed by the debugger will not correspond to the actual state of the application, either due to network latency, or because the state change cannot be immediately detected/reported (as when the entire machine that contains the debuggee crashes). This kind of situation is, for obvious reasons, much more common in scenarios where debugger and debuggee are separated by

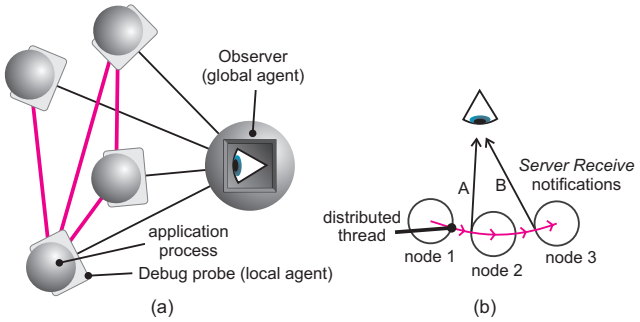

Fig. 3. (a) High-level architectural view of our distributed debugger. (b) Causally related events in a single call chain.

a network. From our experience, debugger implementations may handle these situations in one of two ways: **(1)** allow state drift to go unbounded, knowing that interactive operations might fail because the "live" entity at which the operation is directed may no longer exist, or may transition to a state where the operation is no longer allowed; or, **(2)** bound the drift by synchronizing debugger and debuggee at certain key state transitions, therefore eliminating operation failures that arise from the sort of race condition described in **(1)**.

Alternative **(2)** is accomplished by having the debug probes halt the execution of the local processes at some key state transitions to make sure that the debugger (global agent in Fig. 3(a)) has registered these transitions before proceeding. This does not help, however, in cases where the operation fails because the remote node died unexpectedly – in fact, node death due to crashes is one of the only kinds of state changes that cannot, ever, be reported in a synchronous fashion. Basic guidelines for scalable distributed systems dictate that we should take the first option whenever possible, reducing the amount of synchronous reports to a minimum. The particular issue we faced was how to support user controlled *suspend* operations on distributed threads reliably. A precondition to being able to suspend a distributed thread at an arbitrary instant is knowing the exact position of its head, also at an arbitrary instant – therefore this relates directly to *how* we may consume the information produced by the protocol described in Sec. 3.1. The problem lies in the fact that, if we track the head asynchronously, then we have no way of knowing if the knowledge of the global agent with respect to the current head of a given distributed thread is stale or not. This could lead to a situation where the debugger is constantly behind the actual state for a given distributed thread, making support for suspend inefficient and unreliable. Therefore, in order to provide adequate support for suspend, we opted for synchronously tracking the head; that is, the reporting of *Server Receive* and *Server Return* events will cause the ongoing request to halt until the global agent is known to have updated its knowledge about the new head.

One useful consequence of capturing these state changes in a synchronous fashion is that if two events are causally related (like events A and B in Fig.

3(b)) then the global agent will never observe an inconsistent ordering, simply because B can not happen while A is not allowed to complete. Therefore, we can get away without timestamping, and still trust that the results will be correct.

3.3 Node and Link Failures

So far we have been discussing how to track distributed threads in the absence of failures. When failures are introduced (link failures and node crashes) it is possible that threads which were known to be in the middle of a snapshot begin unblocking and producing snapshot-changing events, thus violating the expected behavior (as by Def. 2) for distributed threads. To deal with such situations, we will introduce an extension to Def. 2 which allows distributed threads to be "split" under certain circumstances. Before that, however, we must define what we consider to be a "normal" unblocking for a local thread.

Definition 5 (Normal Unblocking of a Local Thread). *A local thread l_i that is blocked in a remote call is said to have unblocked normally if its unblocking results from the processing of a reply message, that has been sent by the server which contained the thread that handled the request initiated by l_i, informing that this remote request has completed (either successfully or unsuccessfully).*

Therefore, a blocked local thread unblocks non-normally whenever its unblocking can not be causally traced to a reply message from the server. Now let $f(T,t) = \{l_1, ..., l_n\}$ be the state of a distributed thread T at instant t. Also, let T' be the distributed thread whose base is l_{i+1}, where $l_{i+1} \in f(T,t)$, and let $\epsilon \in \mathbb{R}, \epsilon > 0$. We shall establish that:

- If thread l_i unblocks non-normally at instant $t + \epsilon$, then $f(T, t + \epsilon) = \{l_1, ..., l_i\}$, and $f(T', t + \epsilon) = \{l_{i+1}, ..., l_n\}$.
- If thread l_i is known to be dead at instant $t+\epsilon$, then $f(T, t+\epsilon) = \{l_1, ..., l_{i-1}\}$ and $f(T', t + \epsilon) = \{l_{i+1}, ..., l_n\}$.

In both cases, we say that T has been "split". Thread splits are enough for us to reorganize information whenever a node or link fails for whatever reason, leaving broken distributed threads behind. The user will be notified whenever a split occurs (as these are always errors), and the debugger will do its best to relate the split to its cause (mainly by checking whether the node that is adjacent to the split is still alive).

3.4 Debugger Architecture and Implementation

So far we have discussed our debugger at a fairly high and conceptual level. In this section we discuss its architecture, and some key aspects of its implementation. A general layout of the architecture has already been given in Fig. 3(a). The debugger can be roughly decomposed into two types of participants – the **global agent**, which is the module responsible for assembling execution information, presenting them to the user, and accepting interactive commands, and

the **local agents**, or debug probes, which are responsible for collecting runtime information and interacting with the distributed system processes on behalf of the global agent (or any other client). This centralized architecture is a natural result of the fact that at some point there must be one single observer, who has a global view of the distributed computation.

The Local Agents are composed of a combination of standard symbolic debuggers and custom instrumentation code that is injected by our extensions. This custom instrumentation code implements the thread id propagation scheme described in Sec. 3.1, the tracking protocol, and other assorted functionalities required by the debugger. As shown in Fig. 4(a), local agents use two distinct wire protocols – one that is specific to the symbolic debugger in use (which will be used for setting breakpoints, controlling local threads, getting local state information, etc.), and another one, which is language-independent, that will be used to convey the information required by the thread tracking protocol of Sec. 3.1 (we call this protocol DDWP, or Distributed Debugging Wire Protocol). A simplified schematic view of the actual tracking mechanism, in its Java/CORBA version, is presented in Fig. 4(b). The Java tracking mechanism of Fig. 4 **(b)** is

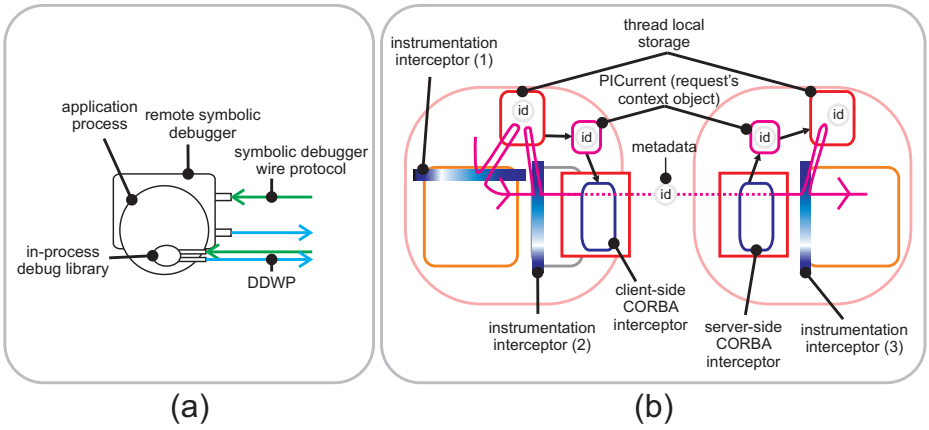

Fig. 4. (a) Anatomy of the local agent. (b) Tracking mechanism for Java/CORBA.

implemented through a combination of thread-local storage, CORBA portable interceptors, and custom instrumentation interceptors, which are inserted at runtime with the help of a Java transformation agent, and the Bytecode Engineering Library [1]. The first custom interceptor is inserted at the beginning of each *Runnable#run()* method, and is responsible for assigning the unique two-part id described in Sec. 3.1 to each local thread as soon as it is started, as well as for enrolling the local threads in a registration protocol which is required for the mapping of numeric ids to *ThreadReference* mirrors provided by the Java Debug Interface (JDI) [18]. The second (inserted at each CORBA stub) and third (inserted at each CORBA servant) instrumentation interceptors will bridge the

thread-local storage and the CORBA portable interceptors, allowing us to pass on the required ids which each request.

Apart from its use in the tracking mechanism, the CORBA and Instrumentation interceptors are also used for the detection of non-mediated recursive calls, and abnormal unblockings of blocked local threads (Sec. 3.3). For the detection of recursive calls, we simply insert a token in the *PICurrent* object whenever a request passes through a CORBA interceptor. The instrumentation interceptors inserted at the remote objects **(3)** will always test for the presence of this token. If it is found, the interceptor will remove it, generate a *Server Receive* event, and initialize a thread-local counter to zero. Subsequent calls to remote object implementations that are not mediated by the ORB will trigger the instrumentation interceptors which, failing to see the token, will just increment the thread-local counter, without generating further *Server Receive* events. Whenever one of these instrumented methods return (either due to a normal return, or due to an exception), the instrumentation interceptor **(3)** will be activated again, and the thread-local counter will be decremented. When the counter reaches zero, the interceptor will know that the current stack frame is an entry frame, and will generate a *Server Return* event to signal that the current head has changed.

The mechanism for testing for abnormal unblockings is also token-based. Recall from Def. 5 that unblockings occur whenever a given thread l_i unblocks due to a reason other than the client-side middleware getting a regular reply message. The two most common causes behind abnormal unblockings are link and node failures, which are not distinguishable from the point of view of the client. We detect abnormal unblockings by sending a single-bit token with each reply. This single-bit token is inserted into the request service context by the server-side CORBA interceptor shortly before the reply is sent, and loaded into the *PICurrent* object by the client-side CORBA interceptor when the reply is received. If there is no reply message, however, the token will never get to the instrumentation interceptor **(1)**, indicating that an abnormal unblocking has occurred. Whenever that happens, the instrumentation interceptor **(1)** will (synchronously) notify the global agent, which will perform the appropriate distributed thread split and notify the user of the erroneous behavior.

The Global Agent: has many responsibilities. It has to control and manage the distributed processes, it has to combine the partial state information provided by each of the local agents, and it has to handle user interaction. According to the notions on computational reflection laid down by Maes [11], symbolic debuggers can be seen as meta-systems whose domain are the execution environments of the debugged processes. Among other things, this means that debuggers should have access to structures that represent (aspects of) the execution environments of such debugged processes. In order to keep things simple, we decided to take the JDI [18] approach and reify distributed threads at the symbolic debugger level. There were, however, many other issues we needed to resolve.

The elements that compose the execution environment of a distributed application may come from many environments. Reifying the environment of the

distributed object system means coming up with an object model capable of accommodating this heterogeneity. Also, since we are worried with extensibility, we would like to have a model that is capable of accommodating new environments with relative ease. Coming up with such a model from the ground up, however, requires time and experience. Fortunately there is already one mature, open source and widely developed debug model which has proven to accommodate heterogeneity, and which would be a perfect fit for our own debugger: the Eclipse debug model [19]. Eclipse is a well-known, extensible environment for building Rich Client Applications. Its debug model has successfully accommodated reified versions of the main elements of Java, C++, Python, Ruby and many other execution environments. Based on that observation, we decided to implement our distributed-thread-based debugger as a set of extensions to the Eclipse debug framework. Our extensions are depicted as the grey areas in Fig. 5. Our main

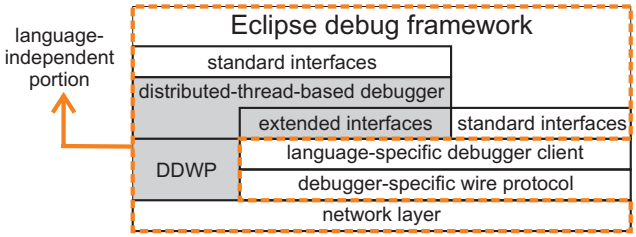

Fig. 5. Layered architecture of the global agent.

contribution to the Eclipse debug framework has been a language-independent, distributed thread debugger, and set of extensions to the regular Eclipse *IThread* interface. The contribution of the Eclipse platform to our project, however, has been also very rich – a collection of ready-to-use debugging solutions that could be realistically adapted to work with our debugger. Adapting an existing Eclipse debugging client amounts to implementing our extended interfaces.

Applicability – middleware, language, and debugger requirements: Now that we have discussed some of the key aspects of our implementation, we are in position to make an assessment on some of the requirements imposed by the technique. This should point us toward some answers to the question that matter the most: how difficult it is to actually port our debugging machinery to other languages/runtime environments, as well as other middleware systems:

1. Application must be based on distributed objects. To take advantage of the distributed thread abstraction, it should also use synchronous calls. Asynchronous calls are supported, but benefits are less clear.
2. The target middleware should allow context information (metadata) to be passed with each request. This is the only requirement imposed on the middleware apart from the use of distributed objects.
3. There must be a "standard" symbolic debugger available for the language, and we should be able to operate it remotely.

4. Object proxies (stubs) and remote objects (servants) must be instrumentable.
5. There must be a way to assign identifiers to each of the local threads that will take part in distributed threads.
6. For distributed deadlock detection, there must be a way for the global agent to know which locks are being held by which local threads.

Requirement 1 is actually a restatement of the type of systems that are the target of this work. Requirement 2 is fulfilled by almost every middleware implementation in use today. Also, it is not a hard requirement – we could get away with 2 by modifying the stub/skeleton generator to include an extra parameter, as in [9]. This would be, however, much more cumbersome. Requirement 3 is also rather reasonable, at least with mainstream languages. A "standard" symbolic debugger is, roughly speaking, a debugger that supports at least line breakpoints, step into, step over, step return, and source mapping capabilities – a feature set that is common to all symbolic debuggers we know of. Also, there must be a way to operate the symbolic debugger remotely, as with GDB [17], or the JPDA [18] debugger, for example. Requirements 4 through 6 are highly language-dependent. In Java – which could hardly be described as a language with strong reflective capabilities – we were able to implement the instrumentation mechanisms rather easily, thanks to Apache BCEL [1] and the Java instrumentation agents. In languages with more sophisticated reflective capabilities such as Python, Ruby, or Smalltalk, this should be even more straightforward. In a language like C++, the task would be made easier with software like OpenC++ [3].

3.5 More on Scalability and Correctness

There are two main sources of concern when the word "scalability" is applied to a distributed debugger: performance scalability (how many nodes can be debugged simultaneously before performance becomes an issue?), and visualization (how many nodes can be debugged simultaneously, before the maze effect takes over?). Our information visualization and navigation mechanism – based on the distributed thread abstraction – scales better than plain local threads. It allows the user to selectively focus his/her attention into what matters the most – the flow of control of his own application – while complementing this stripped down information with other kinds of error information, like detection of distributed deadlocks, node, and link failure. Therefore, from this point of view, our debugger has been built to scale better than conventional symbolic debuggers.

Performance scalability, on the other hand, has been one of our main sources of preoccupation from the start, due to our centralized architecture. This turned out to be less of an issue than initially thought, however, due to the dynamics of the updates produced by the local agents. The global agent keeps an internal table, where each entry contains state information for a distinct distributed thread. As we mentioned before, notifications are always synchronous. This means that, unless there is a thread split going on, updates to a single distributed thread (table entry) will be performed one at a time. A hypothetical update scenario is shown in Fig. 6(a). Since these entries are disjoint, the updates can be per-

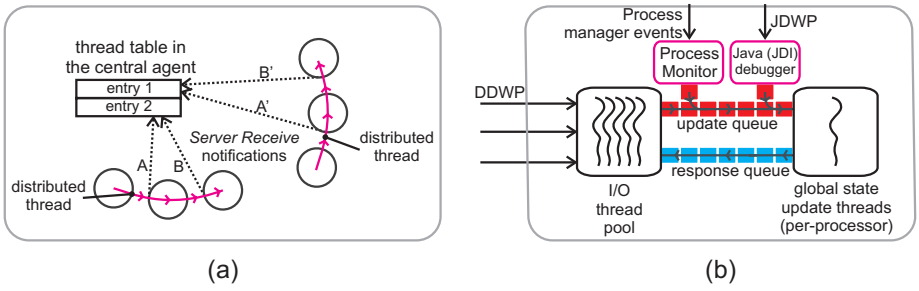

Fig. 6. (a) Dynamics of "regular" updates. (b) Processing in the global agent.

formed concurrently, as long as we keep one updater thread per processor/core (Fig. 6(b)). Also, since updates are performed one at a time, contention on a single entry is non-existent in the absence of failure. Fig. 6(b) shows that the DDWP server is not the only source of state update events – the Java debugger (and other language debuggers), and the process monitor may also contribute with information. This information is related to thread and process lifecycle, and might be used by the updater threads to anticipate the occurrence of thread splits. Although we have not performed any conclusive tests on server scalability, we expect that its capacity to handle more nodes will increase as more processors are added, due to its simple design and due to the small amount of thread-shared state it contains. So far, the DDWP server has been capable of handling more than two dozen nodes without any noticeable performance degradation.

3.6 One-click launch, and Debugger for Testing

Most symbolic debuggers are capable of either instantiating or attaching to running processes with almost no burden on the user. In fact, with "modern" GUI-based debuggers, instantiating or attaching to running processes (either remote or local) can be, in most cases, a simple one-click operation (after previous configuration, of course). This is yet another point where centralized and distributed systems differ fundamentally: while centralized processes can be in one of two states – running or not running – distributed systems can be in many, partially running states, not all of which may work equally well. Take as example the instantiation of a simple client-server application, where the client makes a single request to the server. If the client is started before the server, then it is very likely that it will attempt to perform its request before the server has had the opportunity to properly initialize, resulting in failure. Simply ensuring that the server is started first, however, is not enough – the startup time for the client is probably much smaller than for the server, and the end results will be similar.

Keeping in line with our philosophy of making distributed debugging easier, we have developed a one-click instantiation facility that works with distributed systems as well. Taking our example and generalizing it a little, ensuring successful launching equals ensuring that certain processes are not launched until we are

sure that all other processes it depends on are in a certain state (ready to take incoming communication). Defining proper state without getting application-specific could be a difficult matter. Fortunately, however, we have some powerful instrumentation and state inspection machinery at our disposal – the symbolic debugger itself. With that in mind, we developed a simple launch constraint language, which alleviates this issue by leveraging the knowledge obtainable by the symbolic debuggers. Mainly, this language allows the user to enter declarative statements like:

```
when <Name Srv> reaches org.jacorb.ORB:1278 launch <Srv1>,<Srv2>
when <SomeServer> reaches module1.Type2.line=‘‘ready_to_go’’ &
(module1.Type1.state=1 | module1.Type1.state=2) launch <Client>
```

These dependencies are mapped at runtime into edges in a DAG, which has all of the distributed system processes represented as vertices. The single-click run operation causes all processes with fan-in zero to be started. The remaining processes are launched as local predicates are satisfied. Something we quickly noticed is that this mechanism is very useful for writing automated distributed tests. It is a small part of the puzzle, of course, as it ensures only a deterministic launch sequence. That did not prevent it from being very useful, however, as we were writing distributed, automated integration tests for the debugger itself.

4 Related Work

There is a very vast body of literature on the subject of debugging (and testing) of concurrent programs, but most of this research has been directed at parallel systems. We have therefore selected three works which we consider to be most representative as far as the topics of debugging of distributed object applications and portable debugging are concerned.

OCI's OVATION: The Object Viewing & Analysis Tool for Integrated Object Networks [14] is an extensible debugging tool for CORBA-based distributed object systems. It is comprised of an extensible analysis and visualization engine, and by a collection of probes, which are accompanied by a probe framework. Among other features, it is capable of replaying execution traces off-line. It provides a set of probes for monitoring common CORBA and distributed object application events, like client and servant pre-invoke and post-invoke, exchanged messages, request processing time, and others. Instrumentation may be automatic (as with the message exchange probes), or manual (for user-defined probes, as well as for some OVATION-provided probes). Unlike our tool, OVATION is a monitoring and analysis system. This means that it is not possible to use it to interact with the running distributed system, at least not with the richness attainable with a symbolic debugging tool. Also, as with all monitoring and analysis tools, instrumentation must be thought up-front. And finally, its probe framework (including instrumentation macros) is written in C++, which leads us to believe that, at the time of this writing, no other languages are supported for applications.

IBM's Object Level Trace: Object Level Trace (OLT)[6] is an extension to the IBM distributed debugger. Unlike other debugging tools targeted at distributed object systems, IBM's Object Level Trace incorporates a symbolic debugging service and, like our tool, it allows the user to follow the flow of control of his application from client to server, abstracting middleware details away. The concepts are similar, but OLT does not try to be a symbolic debugger – there are no explicit distributed threads, only message tracking. We are also not quite sure about how extensible OLT is, as it is a closed-source implementation. As far as the authors knowledge go, OLT is restricted to IBM's own technology, like WebSphere and the Component Broker. Also, neither OLT nor IBM's distributed debugger seem to be concerned with application instantiation, at least not beyond providing a simple facility for firing remote processes.

P2D2: The Parallel and Distributed Program Debugger[2] (P2D2) aimed at being a portable debugger for parallel and distributed (cluster) applications based on middleware such as PVM and MPI, as well as runtime environments like HPF. Our approach is based on many of the principles of P2D2, such as a decoupled client-server architecture, the use of a standard, language-independent wire protocol, and the leveraging of existing symbolic debuggers. The difference lays in the fact that we are counting on being able to adapt existing Eclipse-based symbolic debuggers so they can be integrated into our implementation, whereas P2D2 attempted to provide a standardized foundation all by itself. This means our implementation is much simpler. Also, P2D2 attempted to provide a debugger-neutral layer on top of existing symbolic debuggers at the server-side, meaning that all of its wire protocol is standardized. We adopt a different approach with our two-protocol local agent, again trying to facilitate reuse of existing Eclipse-based debugger clients. On the other hand, we require remote-debugging-enabled symbolic debugger clients. Regarding process management, P2D2 delegates the responsibility for process creation to the underlying infrastructure, whereas our implementation takes this responsibility upon itself. While this means we had to develop our own infrastructure, it also meant we did not have to think about interfacing with existing infrastructures.

5 Conclusions and Future Work

This paper presented a simple technique and an extensible Eclipse-based tool for symbolic debugging of distributed object applications. Our rationale for the development of this work has followed two principles: portability and usefulness. Our tool is portable because the tracking technique is simple, and based on elements that are common to synchronous-call middleware platforms. It is also portable because instrumentation requirements are not demanding, and because we can leverage existing, open source debugging clients. The conclusion that it is a suitable candidate for surviving technology evolution draws from these characteristics. Our tool is useful because it helps the user fight the maze effect by bringing debugger abstractions on par with middleware abstractions, because it helps detecting failures and distributed deadlocks, and also because it streamlines the

workflow with its process management and instantiation infrastructure. There are many issues we did not attempt to address in this work, and which could be of value. Integration of more scalable visualization mechanisms (like event and call graphs), and automatic analysis tools [7] would be two examples. Addressing perturbations to the underlying execution with a replay facility would be another avenue. We are currently not, however, any close to having portable execution replay in multithreaded environments. A demonstration screencast, and the source code for our tool, can be obtained at http://god.incubadora.fapesp.br.

References

1. Apache BCEL website. http://jakarta.apache.org/BCEL.
2. D. Cheng and R. Hood. A portable debugger for parallel and distributed programs. In *Proc. of the 1994 ACM/IEEE conf. on Supercomputing*, pages 723–732, 1994.
3. S. Chiba. A Metaobject Protocol for C++. In *Proc. of the ACM OOPSLA'95*, pages 285–299, Oct 1995.
4. S. K. Damodaran-Kamal. *Testing and Debugging Nondeterministic Message Passing Programs*. PhD thesis, Univ. of Southwestern Louisiana, 1994.
5. J. Gait. The Probe Effect in Concurrent Programs. *Soft.: P & E*, 16(3):225–233, Aug 1986.
6. IBM. Object Level Trace. http://publib.boulder.ibm.com/infocenter/wasinfo/v4r0 /topic/com.ibm.websphere.v4.doc/olt_content/olt/index.htm.
7. D. Kranzlmueller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johannes Kepler University, Linz, Austria, Sept 2000.
8. Y. Krishnamurthy et. al. The Design and Implementation of Real-Time CORBA 2.0: Dynamic Scheduling in TAO. In *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 121–129, 2004.
9. J. Li. Monitoring and Characterization of Component-Based Systems with Global Causality Capture. In *Proc. of the 23rd ICDCS*, pages 422–433, 2003.
10. A. Lima et. al. A case for event-driven distributed objects. In *Proc. of DOA'06*, LNCS, pages 1705–1721. Springer-Verlag, 2006.
11. P. Maes. Concepts and experiments in computational reflection. In *Proc. of the OOPSLA'87*, pages 147–155, 1987.
12. N. Mittal and V. K. Garg. Debugging Distributed Programs Using Controlled Re-execution. In *Proc. of the 2000 ACM PODC*, pages 239–248, 2000.
13. B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carneggie Mellon University, Pittsburg, PA, 1981.
14. OCI. OVATION Website. http://www.ociweb.com/products/ovation.
15. C. Pancake. Establishing Standards for HPC System Software Tools. http://nhse.cs.rice.edu/NHSEreview/97-1.html.
16. R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.
17. R. M. Stallman. *GDB Manual: The GNU Source Level Debugger*. FSF, Cambridge, Massachusetts, January 1987.
18. Sun Microsystems. The Java Platform Debug Architecture. http://java.sun.com/prhttp://java.sun.com/products/jpda/index.jsp, 2007.
19. D. Wright and B. Freeman-Benson. How To Write an Eclipse Debugger. http://www.eclipse.org/articles/Article-Debugger/how-to.html.