Debugging distributed applications with replay capabilities

Daniel NERI
Royal Institute of Technology
Stockholm, Sweden
d92-dne@nada.kth.se

Laurent PAUTET
ENST
Paris, France
pautet@inf.enst.fr

Samuel TARDIEU
ENST
Paris, France
sam@inf.enst.fr

Abstract

This paper focuses on the latest developments made by the ENST research team to GLADE, the implementation of the Distributed Systems Annex for the GNAT Ada95 compiler; we have extended GLADE's communication subsystem and added recording facilities and replay capabilities. This makes debugging distributed applications much easier because of the possibility to replay separately each partition by simulating external events at consistent dates without loosing the possible determinism of the original program, and it also eases the debugging in cases where it is not practical to re-run the whole program or when it is impossible to get exactly the same behaviour from one of the part (for example when one or several parts of the application run on embedded targets and send messages depending on sensor inputs while the other parts run on fixed workstations).

1 Ada95 RM Annex E and GLADE

The Ada Distributed Systems Annex, Annex E, provides a solution to programming distributed systems. An Ada application may be partitioned for execution across a network of computers so that typed objects may be referenced through remotely-called subprograms. The remotely-called subprograms declared in a library unit categorized as remote call interface (RCI) may be either statically or dynamically bound, thereby allowing applications to use one of these three classical paradigms:

Remote procedure call: a remote procedure call[1] is similar for the end-user to a regular procedure call. Arguments are pushed into a stream along with some data pointing out which remote procedure is to be used and the stream is transmitted (maybe after some filtering phase, see [14]) over the network to the other partition. This partition decodes the stream, does the regular subprogram call, then put the output parameters into another stream along with the exception (if any) raised by the subprogram and sends this stream back to the caller. The caller decodes the stream and raises the exception if needed. Also, a pointer on a remote procedure call can be dereferenced to make the remote call occur.

Asynchronous procedure call: an asynchronous procedure call is similar to a remote procedure call, but it doesn't wait for the completion of the remote call and lets the caller continue its execution path. Of course, this model is applicable to procedure with only *in* parameters and any exception raised during the execution of the remote procedure is lost.

Distributed objects: this model, similar to the one adopted in CORBA, defines the notion of remote-pointer, a pointer which designates a remote object (the pointer type is said to be RACW, that is remote access to class-wide type). When a primitive dispatching operation is invoked on an object pointed by a RACW, then the corresponding remote call is performed transparently (if the object was created locally). This very powerful mechanism provides the user with a mean of registering objects into an object server and calling methods of these objects without knowing where they were originally created.

An Ada95 distributed application include a number of partitions which can be executed concurrently on the same ma-

[1] In fact a remote procedure call can well be a remote function call, so the appropriate term (not chosen for the Reference Manual) should have been "Remote Subprogram Call".

chine or can be distributed on a network of machines. A partition is a set of compilation units which are linked together to produce an executable binary. Each partition is identified by a unique `Partition_ID` which can be obtained using the `Partition_ID` attribute on any of the packages present in the partition.

The Distributed Systems Annex does not describe how a distributed application should be configured. It is up to the user (using a partitioning tool whose specification is outside the scope of the annex) to define what the partitions in her program are and on which machines they should be executed.

GLADE[2], the implementation of Annex E of the Ada Reference Manual for the GNAT compiler, provides a Configuration Tool and a Partition Communication Subsystem to build a distributed application.

The GNATDIST tool and its configuration language have been purposely designed to let the user partition her program and specify the machines where the individual partitions will be executing.

The Generic Ada Reusable Library for Interpartition Communication (GARLIC in what follows) is a high level communication library (described in [7]) that implements with object-oriented techniques the interface between the Partition Communication Subsystem defined in the RM and the network communication layer.

Therefore, GLADE provides the user with a full environment to develop a distributed application, but debugging remains an unresolved issue. The debugging cycle is the most common methodology for finding and correcting errors in sequential programs. Cyclic debugging is effective because sequential programs are usually deterministic. Debugging parallel (distributed) programs is considerably more difficult because successive executions of the same program often may not produce the same results or may produce the right results by using different execution orders, which may result into severe headaches.

Section 2 of this paper briefly describes a basic way of debugging a distributed application, before we give a more accurate solution with the help of GLADE. In section 3, we describe another class of debugging models based on instant replay facilities. We give an example where the non-repeatability becomes a serious problem to track down bugs in a distributed application. Section 4 describes the design we chose to implement these facilities as well as the user interface. Section 5 gives implementation details and the protocol format of this solution based on the object-oriented facilities offered by GLADE. Section 6 describes some pitfalls in our approach (which may be due to external causes). Section 7 finally concludes the paper and points out some current and future directions of work.

[2]The current release of GLADE has been developed by the ENST team and is maintained by ACT Europe. See `http://www.act-europe.fr/`.

2 Debugging concurrent programs

2.1 Why is this difficult?

Tracking bugs in a sequential program constitutes a very crucial part in software development cycle. At any phase of coding, the possibility of following the flow of a program and inspect and maybe modify variables helps the programmer in enhancing the software and making it more robust.

When working on sequential programs[3], the programmer can assume that, for similar inputs from the environment[4], two successive executions of the same program will lead into identical behaviors.

When working on parallel programs[5], the programmer can not always make the same assumptions since task synchronization can occur at different times, making it much more difficult to ensure that a run can be duplicated identically. Especially in the case of distributed programs, the network load may modify the behavior of the different partitions because events will not occur in the same order.

This problem becomes even more critical when using GDB (the GNU debugger which is Ada-aware) on several partitions at the same time. Given that the various debuggers do not work cooperatively, all the partitions except the ones being stopped by the debugger will keep running thus causing a possibly completely different execution.

2.2 Debugging with GLADE

During the development of GLADE, the use of a debugger for each partition appeared to be very insufficient to verify that the observed behavior was matching exactly the expected one. Even the scripting capabilities of GDB were too intrusive because it forced the whole partition to be stopped while GDB was executing scripts. Moreover it would have been very impractical to beg GLADE users to use GDB themselves to watch what was going on in the distributed program and it would have been very impractical for them to send meaningful bug reports.

For these reasons, we have added selective traces in the whole run-time library. When configured without the "`--enable-debug`" option, the run-time library omits all the

[3]A sequential program is a program which has only one thread of control.

[4]An input from the environment may consist into data read from a file, user interaction, numbers coming from a pseudo-random generator and even the time of the day since the program may behave differently depending on this time.

[5]A parallel program is a program which has several threads of control. A distributed program is thus a particular form of parallel program.

traces hence resulting in optimal execution speed. On the contrary, "`--enable-debug`" at configuration time enables the traces that can be individually selected at run-time by the mean of environment variables; this of course causes the execution of the distributed program to be slightly slower.

For example, setting the environment variable "`TCP`" to "`C:T:W`" enables the output of tracing information from the TCP/IP protocol package concerning peer-to-peer communication (`C`), internal tables management (`T`) and warnings issued by the run-time when a suspicious behavior is detected (`W`).

Using this method, it happened to be very easy for us to handle bug reports from end users (we ask them to turn on all the debugging informations and then to send logs) and to understand and deal with them in an efficient way. Unfortunately, this method is well suited to debug GLADE runtime libraries but cannot be used to debug an end-user's distributed application, since the traces do not have any level of abstraction and cannot be intepreted meaningfully by the user.

3 Trace/Replay Facilities

3.1 Determinism vs. Non-Determinism

There exist many causes of non-determinism. One of them is the fact that parallel program can use asynchronous message queues between different partitions of the program while a sequential program will wait for the completion of a subprogram before continuing its execution path.

In the presence of asynchronous communication, message latencies can cause two executions of the same program (on the same input) to produce different results because of differences in ressources availability between the two run (network buffers for example). This non-repeatability makes it difficult to use traditional sequential debugging techniques that are most usable when the very same execution can be replayed.

To illustrate this problem in the context of the Ada95 Distributed Systems Annex, we will show a simple example based on the use of asynchronous subprograms.

Considering the code shown from sample 1 to sample 3, we have the following configuration: Main is located on partition P0, RCI_1 is located on partition P1 and RCI_2 on partition P2. The expected behaviour of the Main subprogram in the non-distributed case (or in the absence of `pragma Asynchronous`) is to execute RCI_1.P(1) (which itself invokes RCI_2.P(1) and sets the variable RCI_2.Y to 1) then to execute RCI_2.P(2) which will at the end let the variable RCI_2.Y with the value 2.

But in the distributed asynchronous case, if the (P0, P1) communication channel is slower than the (P0, P2) and (P1, P2)

```
package RCI_1 is
   pragma Remote_Call_Interface;
   procedure P (X : in Integer);
   pragma Asynchronous (P);
end RCI_1;

with RCI_2;
package body RCI_1 is
   procedure P (X : in Integer) is
   begin
      RCI_2.P (X);
   end P;
end RCI_1;

package RCI_2 is
   pragma Remote_Call_Interface;
   procedure P (X : in Integer);
end RCI_2;

package body RCI_2 is
   Y : Integer := 0;
   procedure P (X : in Integer) is
   begin
      Y := X;
   end P;
end RCI_2;
```

Sample 1: The RCI packages

```
with RCI_1, RCI_2;
procedure Main is
begin
   RCI_1.P (1); – Step 1
   RCI_2.P (2); – Step 2
   - - What's the value of Y now? 1 or 2?
end Main;
```

Sample 2: The `Main` procedure

```
configuration Example is
   P0 : Partition;
   P1 : Partition := (RCI_1);
   P2 : Partition := (RCI_2);
   procedure Main is in P0;
end Example;
```

Sample 3: The GLADE configuration file

communication channels, the assignment of Y to 2 can occur before the assignment to 1. In this case, variable Y in RCI_2 is undetermined (either 1 or 2). The determinism of this sequential program in a non-distributed environment is no longer preserved in a distributed execution.

A particular case we are interested in is the one of a deterministic non-distributed program which is then splitted into several partitions. We want to be able to reproduce the same scenario several times even if the non-distributed program becomes nondeterministic because of asynchronous messages. We also want to make it practical to replay a particular partition to find bugs in situations where it is not easy to re-run the whole program; this is the case for example with a distributed program running at the same time on an embedded target and on a workstation. It must be possible to replay the workstation partition without re-loading and re-running the embedded system partition, especially if the behaviour of the embedded system depends heavily on the result of sensors, which may be very difficult to duplicate with enough precision, especially if sensors measure things like heath, very precise distance from an object, very precise angle, or any uncontrollable input.

3.2 Introduction to Trace/Replay

To be able to use the normal iterative debugging method used when debugging traditional sequential systems, we must ensure that re-execution is deterministic. We want the possibility to repeatedly replay an erroneous execution to analyze it more carefully and gain more information about bugs. A general solution is to use a trace/replay scheme.

There are two general approaches to replay — *control driven replay* and *data driven replay*. The latter is, perhaps, the most straight-forward, in which both event order and message content are recorded during tracing. This method is generally suitable for loosely coupled parallel systems.

This approach eases considerably the debugging of the communication layer itself and can help to detect an inconsistency in data representation. Such an inconsistency may occur either because of a bug in a user-defined filter[6] or because of a bug in GLADE.

This solution also allows the user to debug, in isolation, a single partition in a distributed program. Since every message exchanged between this partition and the rest of the distributed program has been saved, the replay module is able to simulate the presence of the other partitions.

In contrast, with control driven replay[10], only the event order is recorded. The replay is rendered deterministic by re-execution within the same external environment as during tracing, and forcing the event order to be the same as during the traced execution. Control driven replay is suited for tightly coupled systems, where otherwise the amount of trace data may turn out to be a problem. By tracing the original message deliveries and then forcing them to occur during replay, the computation and all its messages will be exactly reproduced.

The solution we chose to implement is the data-driven replay, because we consider it as much more useful since the embedded systems we are working on cannot always guarantee to get the same input from their sensors. Therefore we must replay the whole scenario with exactly the same input data, thus using data-driven replay.

4 Our approach

At first, we chose to implement data driven replay, which is straight-forward, and in this case, the most useful solution. If "trace mode" is activated, GLADE will record all messages received by a partition into a trace file. The trace file can then be used to replay the execution of the partition, in isolation.

4.1 GARLIC Architecture

GARLIC has a modular, layered and object-oriented architecture[7], which made relatively easy to integrate a trace/replay facility. The important modules in this context are the core of GARLIC, called *Heart*, and the protocols. The protocol layer makes communication on the layers above it independent of the actual protocol used to transmit messages.

All protocols inherit from a common abstract protocol class. The functionalities are divided in two parts: a passive one and an active one. The passive part gets fed through a *Send* subprogram which directs the protocol object to send some data to another partition. The active part is responsible for catching any incoming data from other partitions and for sending it to the *Heart* module which will handle the incoming packet.

In our case, the tracing utility acts as a hook which records any transaction in the log file and will eventually replay them later. The name of the log file is made from the name of the partition (as chosen by the user in the configuration file) with a standard extension added.

4.2 Recording a session

If a partition is launched with the `--trace` command line argument, trace mode is activated.

[6]A GLADE user can define new filters to compress or encrypt data before sending it on the network, see [14].

[7]See [7] for details about GARLIC implementation.

In trace mode, a trace of each message arrival is stored in the partition's trace file.

4.3 Replaying one or more partitions

When a partition is launched with the `--replay` command line argument, "replay mode" is activated. The right log file is automatically selected from the name of the partition (see section 4.1).

A partition is replayed using a special "replay protocol" instead of the usual communication protocol[8]. This particular protocol is transparently used as a substitute for the one which was chosen during the session recording.

4.4 Analyzing the log files

The standard package Ada.Streams.Stream_IO is used to read and write the trace entries to the trace file. Each entry is preceded by its length, since messages (and thus trace entries) are of variable length. The format of these entries is simple and well described and will be used by the external analysis tools we are going to implement (see section 7).

5 Implementation Details

5.1 Tracing Hooks

In trace mode, when a message is received from another partition and delivered to *Heart* by a protocol, a trace entry is stored in the trace file. The trace entry consists in the message contents along with the sender `Partition_ID` and the time elapsed since the arrival of the previous message. Thus the modification we had to make to GARLIC core was to insert a hook that records traces of all incoming messages whenever the trace mode (controlled by the `--trace` command line argument) is activated. On figure 1, we can see a scenario where partition 1 is communicating with partition 2. Partition 1 is being traced and all the input it receives is stored permanently in a log file.

In GARLIC, the computation of `Partition_IDs` is nondeterministic because they are fully dynamic and depend on the order in which the partitions first connect to the boot server, which is the part of the main partition which is in charge of allocating unique `Partition_IDs` to ID-less partitions. During replay it is crucial that the partition obtains the same ID as during tracing. Consequently, there is also a hook in the `Partition_ID` assignment code. This hook stores the ID assigned to the partition in the trace file. Then, in replay mode, the stored ID is used instead of one allocated by the boot server.

[8]"TCP" for example is a protocol which uses BSD sockets[19].

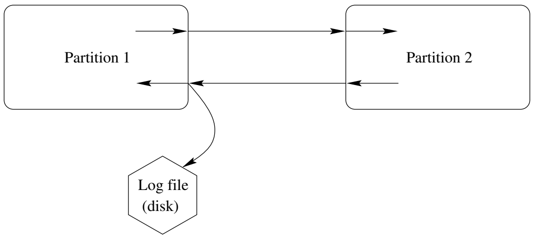

Figure 1: Tracing hook

5.2 The trace file format

A trace file format is made of:

1. The `Partition_ID` of the partition. Even if some exchanges took place to obtain the `Partition_ID` (this is usually the case for any partitions but the main one), their content is not stored in this file since the negotiation happens under the protocol layer. Each protocol is in charge of providing the system with a way of obtaining a `Partition_ID` and thus this is out of the scope of the trace and replay module.

2. A succession of `Trace_Entry_Type` records (as described in sample 4), each one corresponding to an incoming message. No attempt is made to interpret the content of this array of bytes, since it may have been encrypted, compressed or otherwise filtered using GLADE's filtering scheme ([14]).

```
type Trace_Entry_Type
     (Length : Ada.Streams.Stream_Element_Count)
is record
     Arrival_Delay : Ada.Real_Time.Time_Span;
     Data          : Ada.Streams.Stream_Element_Array
                        (1 .. Length);
     Partition     : System.RPC.Partition_ID;
end record;
```

Sample 4: Trace file format

It is interesting to note that since GLADE uses the XDR format ([11]) to build streams in order to provide heterogeneous distribution, the trace file format is also subject to this encoding. It means that a trace file generated on a little-endian machine will be read immediately on a big-endian machine even with a different word-size. This implies that a session recorded on an embedded system will be handled successfully later by analysis tools (see section 7) running on a workstation with a windowing system such as X-Window; it will then be more practical to examine the content the various packets.

5.3 Adding a Protocol to GARLIC

A new protocol is added by subclassing the abstract *Protocol* class. In addition, this new class must be registered during the startup of GARLIC, so that it can be employed by the user. The registration associates the protocol with a name which is stored in the table of protocols.

5.4 Details of the "replay" protocol

The replay protocol — like all GARLIC protocols — inherits from the class *Protocol*.

The actual active part of the replay protocol consists in a task which reads an entry from the trace file, sleeps[9] for the recorded amount of time, and then delivers the message recorded in the current trace entry to *Heart* until all trace entries are consumed.

The *Send* method (the passive part) of the replay protocol simply discards the message. Figure 2 shows the replay of partition 1; fake input is received from partition 2 (which is not existing anymore at this time) and output for partition 2 is silently discarded.

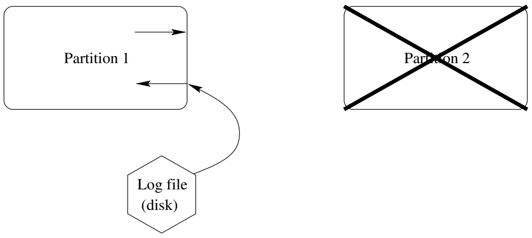

Figure 2: A replay session

In this way, the replay protocol can be said to simulate the other partitions using a trace file. Thus, in replay mode no communication is going on, all messages *to* the partition come from the trace file. The messages *from* the partition are simply discarded.

[9]This delay is provided to simulate delays previously caused by remote computations and message passing times; re-execution would not be deterministic without it.

6 Pitfalls

As indicated in section 5, the whole stream is recorded into the log file. This means that this stream contains a value of type `System.RPC.RPC_Receiver`, which is an access to subprogram type. This value had been transmitted to other partitions during the original run and had been used in any request to indicate the receiver what subprogram had to be called to handle the incoming request.

If the executable gets re-compiled or re-linked with different options, different subprograms may end up at different addresses, thus invalidating all the previously recorded access to subprogram values. To prevent this, we could implement a timestamp or checksum-based check on the executable to make sure this constraint is not violated.

A similar problem can occur if the operating system can relocate the executable at different places for different executions. Although this is not common, this is more serious since there is no way to keep valid pointers across two separate executions. A solution could be to modify the stream at an upper-level to map `System.RPC.RPC_Receiver` objects forth and back to an index in an array. This way, it would be possible to replay the partition code anyway.

A different class of problems is that it is not always possible to store every message since the beginning of the execution, especially for server partitions whose lifetime can be very long. A checkpointing mechanism is being considered to decrease the size of the logfile in certain cases (see section 7).

7 Future work

In the near future, we will be implementing several debugging tools built on top of a library designed to interpret the trace file after the execution has been completed (this technique is known as "*post-mortem* analysis"). A text-only tool will be developed whose output will be easily incorporated in a regression testing tool, and a graphical tool will let the user examine step by step what happens between the different partitions.

The next step will be a graphical tool which will let the user replay the distributed program using the recorded information, with the capability to modify the dates of arrival of the various messages, to easily simulate a network congestion or a fault in a partition and check that the distributed program still behaves as expected.

The third step will be the extension of this debugger to the GNAT runtime library to execute and debug tasks the same way messages between partitions can be examined.

Finally, the current mechanism may be used as a starting point for cold restart in our future implementation of fault-tolerant distributed Ada programs. It may help keeping the messages received by a partition which needs re-launching after the latest checkpoint. Moreover, this mechanism will dramatically reduce the size of the log file since the checkpointing file has a bounded size ([12]) and checkpoints may be performed each time the log file becomes larger than a fixed limit.

References

[1] J S Briggs, S D Jamieson, G W Randall, and I C Wand. Debugging distributed Ada programs. Final report on the PAPA project, University of York, June 1994.

[2] U.S. Army Missile Command. *Real-Time Executive for Multiprocessor Systems*. Redstone Arsenal, Alabama.

[3] IEEE Standards Committee. *IEEE Standard 1003.1c*. 1995.

[4] Mariano P Consens, Masum Z Hasan, and Alberto O Mendelzon. Debugging distributed programs by visualizing and querying event traces. Technical report, University of Toronto.

[5] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, second edition, 1994.

[6] R Curtis and L Wittie. BugNet: A debugging system for parallel programming environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 394–399, October 1982.

[7] Yvon Kermarrec, Laurent Pautet, and Samuel Tardieu. GARLIC: Generic Ada Reusable Library for Interpartition Communication. In *Proceedings of the Tri Ada conference*, Anaheim, California, 1995. ACM.

[8] Thomas Kunz. Abstract debugging of distributed applications. Technical Report TI-10/93, Technische Hochschule Darmstadt, December 1993.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] Thomas J LeBlanc and John M Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[11] Sun Microsystems. *xdr – library routines for external data representation*.

[12] Sape Mullender. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[13] Robert H B Netzer and Barton P Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Minneapolis, MN, November 1992.

[14] Laurent Pautet and Thomas Wolf. Transparent filtering of streams in GLADE. In *Proceedings of Tri-Ada'97*, Saint-Louis, Louisiane, 1997.

[15] Michel Raynal. *Gestion des données réparties : problèmes et protocoles*. Collection de la Direction des Études et Recherches d'Électricité de France. Eyrolles, 1992.

[16] Edmond Schonberg and Bernard Banner. The GNAT project: A GNU-Ada 9X compiler. In *Proceedings of Tri-Ada'94*, Baltimore, Maryland, 1994.

[17] Richard M Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1994.

[18] Richard M Stallman and Cygnus Support. *Debugging with GDB, The GNU Source-Level Debugger*, 4.12 (for GDB version 4.16) edition, January 1994.

[19] Richard W Stevens. UNIX *Network Programming*. Prentice Hall, 1990.

[20] Tucker Taft. *Ada 95 Reference Manual: Language and Standard Libraries*. February 1994. ISO/IEC/ANSI 8652:1995.