Net-dbx: A Java Powered Tool for Interactive Debugging of MPI Programs Across the Internet

Neophytos Neophytou and Paraskevas Evripidou

Department of Computer Science
University of Cyprus
P.O. Box 537
CY-1678 Nicosia, Cyprus
Tel:+357-2-338705, Fax: 339062
skevos@turing.cs.ucy.ac.cy

Abstract. This paper describes Net-dbx, a tool that utilizes Java and other WWW tools for the debugging of MPI programs from anywhere in the Internet. Net-dbx is a source level interactive debugger with the full power of gdb augmented with the debug functionality of LAM-MPI. The main effort was on a low overhead but yet powerful graphical interface that would be supported by low bandwidth connections. The portability of the tool is of great importance as well because it enables us to use it on heterogeneous nodes that participate in an MPI multicomputer. Both needs are satisfied a great deal by the use of Internet Browsing tools and the Java programming language. The user of our system simply points his browser to the URL of the Net-dbx page, logs in to the destination system, and starts debugging by interacting with the tool just like any GUI environment. The user has the ability to dynamically select which MPI-processes to view/debug. A working prototype has already been developed and tested successfully.

1 Introduction

This paper presents Net-dbx, a tool that uses WWW capabilities in general and Java applets in particular for portable parallel and distributed debugging across the Internet. Net-dbx is now a working prototype of a full fledged development and debugging tool. It has been tested for the debugging of both Fortran and C MPI programs.

Over the last 3-4 years we have seen an immense growth of the Internet and a very rapid development of the tools used for browsing it. The most common of form of data traveling in gigabytes all over the net is WWW pages. In addition to information formatted with text, graphics, sound and various gadgets, WWW enables and enhances a new way of accomplishing tasks: Teleworking.

Although, teleworking was introduced earlier, it has been well accepted and enhanced through the use of the web. A lot of tasks, mostly transactions, are done in the browser's screen, enabling us to order and buy things from thousands

D. Pritchard, J. Reeve (Eds.): Euro-Par'98, LNCS 1470, pp. 181–189, 1998.
© Springer-Verlag Berlin Heidelberg 1998

of miles away, edit a paper, move files along to machines in different parts of the world. Until recently, all these tasks were performed using CGI scripts [1] .

The CGI scripts are constrained in exchanging information only through text-boxes and radio buttons in a standard Web Form . The Java language [1][2] is now giving teleworking a new enhanced form. Java applets allow the user to manipulate graphics with his mouse and keyboard and send information, in real time, anywhere in the Internet. Graphical web applications are now able to use low bandwidth connections, since the interaction with the server does not demand huge amounts of X-Protocol [3] data. This even makes it possible to perform the task of remote program development and debugging from anywhere in the net.

Developing and debugging parallel programs has proven to be one of the most hazardous tasks of program development. When running a parallel program, many things can go wrong in many places (processing nodes). It can lead to deadlock situations when there are bugs in the message distribution (in distributed machines), or when access to the common memory is not well controlled. Some processing nodes may crash under some circumstances in which the programmer may or may not be responsible. To effectively debug parallel programs we should know what went wrong in each of these cases [4].

In order to effectively monitor the programs execution, there are two approaches: Post mortem and runtime debugging. In post mortem debugging, during execution, the compiled program, or the environment, keeps track of everything that happens and writes every event in special files called trace files. These trace files are then parsed using special tools to help the user guess when and what went wrong during execution of a buggy program. Runtime tools, on the other side, have access to the programs memory, on each machine, and using operating system calls they can stop or resume program execution on any execution node. These tools can change variables during the execution and break the program flow under certain conditions.

The prototype described in this paper is a runtime source-level debugging tool. It enhances the capabilities of the gdb debugger [5],[6] , and the monitoring facilities of LAM [7][8] into a graphical environment. Using Java, we developed a tool that integrates these two programs in a collection of applets in a single Web Page. Our tool acts as an interface to the two existing tools, which provides the user with a graphical interaction environment, and in addition, it optimizes interaction between LAM and gdb.

2 Architecture of Net-dbx

Net-dbx is a client-server package. The server side is the actual MPI-Network, that is the group of workstations of processors participating in an MPI-Multicomputer. It consists of tools that are to be installed locally on each Node and

[1] CGI stands for Common Gateway Interface. It consists of a server program, residing on the HTTP server, which responds to a certain form of input (usually entered in web forms) and produces an output Web page to present results to the user.

can be used for individually debugging the processes running on that Node. These tools include the MPI runtime environment (we currently use the LAM implementation) and a source level runtime debugger (we currently use gdb). On the client side there should be a tool that integrates debugging of each Node and provides the user with a wider view of the MPI program which runs on the whole Multicomputer. The program residing in the client side is an applet written in Java. It is used to integrate the capabilities of the tools which rely on the server side. We can see Net-dbx as an environment consisting of two layers: the lower layer which is the MPI Network and the higher layer as the applet on the Web browser running on the user's workstation.

To achieve effective debugging of an MPI Parallel Program, one must follow the execution flow on every node of the multicomputer network. The status of messages and status of the MPI processes have to be monitored. Net-dbx is designed to relieve the user from the tedious procedure of manual parallel debugging. This procedure would involve him capturing PIDs as a parallel program is started, connecting to each node he wants to debug via telnet, and attaching the running processes to a debugger. There are also several synchronization issues that make this task even more difficult to achieve. In addition, Net-dbx offers a graphical environment as an integrated parallel debugging tool, and it offers means of controlling/monitoring processes individually or in groups.

2.1 Initialization Scheme

As mentioned above, to attach a process for debugging you need its PID. But that is not known until the process starts to run. In order to stall the processes, until they are all captured by the debugger, a small piece of code has to be added in the source code of the program. An endless loop, depending on a variable and two MPI_Barriers that can hold the processes waiting until they are captured. In a Fortran program the synchronization code looks like the following:

```
if ( myid .eq. 0 ) then
         dummydebug=1
         dowhile ( dummydebug .eq. 1 )
         enddo
     endif
call MPI_BARRIER( MPI_COMM_WORLD, ierr )
call MPI_BARRIER( MPI_COMM_WORLD, ierr )
```

The variable myid represents the process' Rank. As soon as all the processes are attached to the debugger, we can proceed to setting a breakpoint on the second MPI_BARRIER line and then setting the dummydebug variable on the root process to 0 so that it will get out of the loop and allow the MPI_Barriers to unlock the rest of the processes. After that the processes are ready to be debugged. The dummy code is to be added by the tool using standard searching techniques to find the main file and then compile the code transparently to the user.

2.2 Telnet Sessions to Processing Nodes

Telnet Sessions are the basic means of communication between the client and the server of the system. For every process to be debugged, the Java applet initiates a telnet connection to the corresponding node. It also has to initiate some extra telnet connections for message and process task monitoring. The I/O communication with this connection is done by exchanging strings, just like a terminal where the user sends input from the keyboard and expects output on the screen. In our case, the Telnet Session must be smart enough to recognize and extract information from the connections response. The fact that standard underlying tools are used guarantees same behavior for all the target platforms.

Standard functionality that is expected from the implementation of the Telnet Sessions role is to provide abstractions to:

- setting/unsetting breakpoints,
- starting/stopping/continuing execution of the program,
- setting up and reporting on variable watches,
- evaluating/changing value of expressions and
- providing the source code (to the graphical interface).
- Capturing the Unix PIDs of every active process on the network (for the telnet session used to start the program to be debugged).

In addition, the telnet Session needs to implement an abstraction for the synchronization procedure in the beginning. That is, if it is process 0 the session should wait for every process to attach and then proceed to release (get out of the loop) the stalled program. If it is a process ranked 1..n, then it should just attach to the debugger, set the breakpoints and wait. This can be achieved using a semaphore that will be kept by the object which will "own" all the telnet Sessions. In programming telnet session's role, standard Java procedures were imported and used, as well as third-party downloaded code, incorporated in the telnet part of our program. To the existing functionality we added interaction procedures with Unix shell, GNU Debugger (gdb), and with LAM-MPI so that the above abstractions were implemented.

One of the major security constraints posed in Java is the rule that applets can have an Internet connection (of any kind-telnet, FTP, HTTP, etc) only with their HTTP server host [9], [10]. This was overcome by having all telnet connections to the server and then rsh to the corresponding nodes. This approach assumes that the server can handle a large number of telnet connections and that the user will be able to login to the server and then rsh to the MPI network nodes.

Telnet Sessions need to run in their own thread of execution and thus need to be attached to another object which will create and provide them with that thread. A Telnet Session can be attached either to a graphical DebugWindow, or to a non-graphical global control wrapper. Both of these wrappers acquire control to the process under debugging using the abstractions offered by the telnet session.

2.3 Integration/Coordination

As mentioned above, several telnet sessions are needed in order to debug an MPI program in the framework that we use. All these sessions need a means of coordination. The coordinator should collect the starting data (which process runs on which processing node) from the initial connection and redistribute this information to the other sessions so that they can telnet to the right node and attach the right process to the debugger.

A large array holds all the telnet sessions. The size is determined by the number of processes that the program will run. Several pointers of the array will be null, as the user might not want to debug all the running processes. The other indices are pointing to the wrappers of the according telnet sessions that will be debugged. The wrapper of a telnet session can be either a graphical *DebugWindow*, described in the next subsection, or a non-graphical *TelnetSessionWrapper*, which only provides a thread of execution to the telnet session. Additionally it provides the methods required for the em coordinator to control the process together with a group of other processes. The capability of dynamically initiating a telnet session to a process not chosen from the beginning is a feature under development.

As the user chooses, using the user interface, which processes to visualize, and which are in the group controlled by the *coordinator*, the wrapper of the affected process will be changed from a *TelnetSessionWrapper* to a *DebugWindow* and vice-versa.

When LAM MPI starts the execution of a parallel program it reports which machines are the processing nodes, where each MPI process is executed and what is its Unix PID. The interpreted data is placed in an *AllConnectionData* object, which holds all the necessary startup information that every telnet session needs in order to be initialized. After acquiring all the data needed, the user can decide with the help of the graphical environment which of the processes need to be initiated for debugging.

After that, the coordinator creates *ConnectionData* objects for each of the processes to be started and triggers the start of the initialization process. It acts as a *semaphore holder* for the purposes of the initial synchronization.

Another duty of the coordinator is to keep a copy of each source file downloaded so that every source file is downloaded only once. All the source files have to be acquired using the list command on the debugger residing on the server side. These are kept as string arrays on the client side, since access to the user's local files is not allowed to a Java applet. A future optimization will be the use of FTP connections to acquire the needed files.

3 Graphical Environment

As mentioned in the introduction the user Interface is provided as an applet housed in a Web Page. Third-party packages [2] are specially formed to build a

[2] We modified and used the *TextView* component which was uploaded to the *gamelan Java directory* [11] by Ted Phelps at DTSC Australia. The release version will use

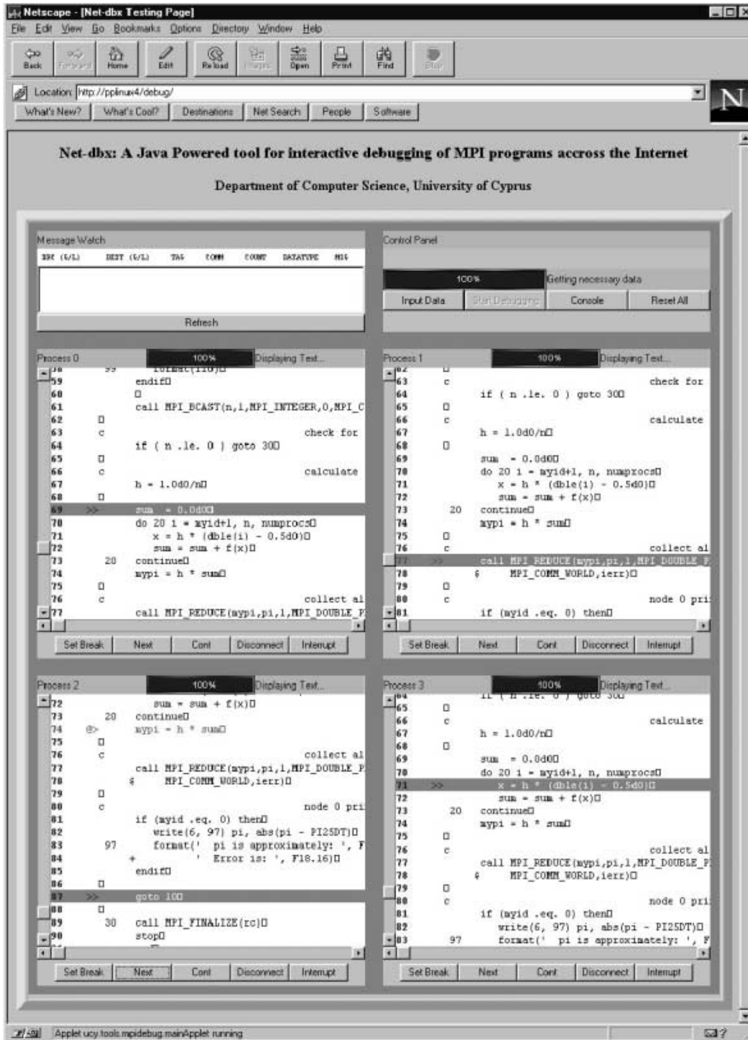

Fig. 1. A snapshot of the Net-dbx debugger in action.

fast and robust user interface. The graphical environment that is shown in the main browser screen (see figure 1) is consisted of:

- The *Process coordinator* window (control panel) which is the global control panel of the system.
- The *debugging windows*, where all the debugging work is done
- The *interaction console*, where the user can interact with the main console of the program to be debugged
- The *message window*, where all the pending messages are displayed.

The *Process coordinator window* provides a user Interface to the *coordinator* which is responsible for getting the login data from the user, starting the debugging procedure and coordinating the several telnet sessions. It provides means for entering all the necessary data to start a debugging session such as UserID, Password, Program Path which are given by the user. The user is also enabled to select which processes are going to be active, visualized, or controlled within a global process group.

The most important visual object used in this environment is the *Debug Window* in which the code of the program is shown. In addition to the code, as mentioned before, this object has to implement behaviors such as showing the current line of execution and by indicating which lines are set as breakpoints. These behaviors are shown using painted text (red color) for the breakpoints, painted text for the line numbers (blue color), and painted and highlighted text for the current line. The appropriate buttons to *set/unset breakpoints*, *next* and *continue* are also provided.

The *interaction console* is actually a typical telnet application. It is the user interface to a special telnet session used by the coordinator to initialize the debugging procedure. After initialization and capturing of all the needed data, the interaction console is left as a normal telnet window within which the user I/O takes place.

The *message view* is the applet responsible for showing the message status. It displays all the information that is given by the LAM environment when an mpimsg command is issued. All the pending messages are shown in a scrolling List object.

4 Related Work

A survey on all the existing graphical debugging tools would be beyond the scope of this paper. However we present the tools that appear to be most similar to Net-dbx. For an overview of most of the existing parallel tools, the interested reader can visit the Parallel Tools Consortium home page [12].

the *IFC Java widget set* as it appears to be one of the fastest and most robust *100% Compatibility certified* Java toolkits.

4.1 XMPI

XMPI [7] is a graphical environment for visualization and debugging, created by the makers of the LAM- MPI environment at the Ohio Supercomputer Center. It basically features graphical representation of the program's execution and message traffic with respect to what each process is doing. XMPI uses a special trace produced by the LAM-MPI environment during the execution. Although XMPI can be used at runtime it is actually a post-mortem tool and uses the LamTraces for every function that it provides. XMPI seems to be more of a visualization tool for displaying the program execution, but not controlling it. It is supported on DEC, HP, SGI, IBM and Sun platforms.

4.2 TotalView

This package is a commercial product of Dolphin Interconnect Solutions [13]. It is a runtime tool and supports source level debugging on MPI programs, targeted to the MPICH implementation by Argonne National Labs. Its capabilities are similar to those of dbx or gdb, which include changing variables during execution and setting conditional breakpoints. It is a multiprocess graphical environment, offering an individual display for each process, showing source code, execution stack etc. TotalView is so far implemented on Sun4 with SunOS 4.1.x platforms, Digital Alpha with Digital Unix system and IBM RS/6000.

4.3 p2d2

Another ongoing project is the p2d2 Debugger [14] being developed at the NASA Ames Research Center. It is a parallel debugger based on the client/server paradigm. It is based on an Object Oriented framework that uses debugging abstractions for use in graphical environments. It relies on the existence of a debugging server on each of the processing nodes. The debugging server provides abstractions, according to a predefined framework and can be attached to the graphical environment at runtime. Following this architecture the tool achieves portability and heterogeneity on the client side, whereas it depends on the implementation of the server on the server side.

5 Concluding Remarks and Future Work

In this paper we have presented Net-dbx, a tool that utilizes standard WWW tools and Java for fully interactive parallel and distributed debugging. The working prototype we developed provides proof of concept that we will soon be able to apply teleworking for the development debugging and testing of parallel programs for very large machines. A task that, up to now, is mostly confined to the very few supercomputer centers worldwide.

Net-dbx provides runtime source level debugging on multiple processes and message monitoring capabilities. The prototype provides a graphical user interface at minimal overhead. It can be run from anywhere in the Internet even using

a low bandwidth dialup connection (33KBps). Most importantly, Net-dbx is superior over similar products in compatibility and heterogeneity issues. Being a Java applet, it can run with no modifications on virtually any console, requiring only the presence of a Java enabled WWW browser on the client side.

The tool at it's present implementation is being utilized in the Parallel Processing class at the University of Cyprus. After this alpha testing, it will be ready for beta testing on other sites as well. We are currently working on extending the prototype presented, to an integrated MPI-aware environment for program development, debugging and execution. For more information of the Net-dbx debugger's current implementation state, one can visit the Net-dbx home page [15].

References

1. Laura Lemay and Charlse L. Perkins, Teach yourself Java in 21 days, Sams.net Publishing, 1996 182
2. The JavaSoft home page, `http://java.sun.com`, The main java page at Sun Corporation. This address is widely known as the JAVA HOME 182
3. X Window System, The Open Group, `http://www.opengroup.org/tech/desktop/x/`, The Open Group's information page on the X-Window System. 182
4. Charles E. McDowell, David P. Helmbold, Debugging concurrent programs, ACM Computing Surveys Vol. 21, No. 4 (Dec. 1989), Pages 593-622 182
5. Richard M. Stallman and Roland H. Pesch, Debugging with GDB, The GNU Source-Level Debugger, Edition 4.09, for GDB version 4.9 182
6. GNU Debugger information, http://www.physik.fu-berlin.de/edv_docu/documentation/gdb/index.html, 182
7. LAM / MPI Parallel Computing, Ohio Supercomputer Center, `http://www.osc.edu/lam.html`, The info and documentation page for LAM. It also includes all the documentation on XMPI 182, 188
8. Ohio Supercomputer Center, The Ohio State University, MPI Primer Development With LAM 182
9. J. Steven Fritzinger, Marianne Mueller, Java Security white Paper, Sun Microsystems Inc., 1996 184
10. Joseph A. Bank, Java Security, `http://www-swiss.ai.mit.edu/~jbank/javapapaer/javapaper.html` 184
11. The GameLan home page, `http://www.gamelan.com`. Here resides the most complete and well known java code collection in the Internet 185
12. The Parallel Tools Consortium home page, `http://www.ptools.org/` 187
13. Dolphin Toolworks, Introduction to the TotalView Debugger, `http://www.dolphinics.com/TINDEX.html` 188
14. Doreen Cheng and Robert Hood, A Portable Debugger for Parallel and Distributed Programs, Supercomputing '94 188
15. The Net-dbx home page, `http://www.cs.ucy.ac.cy/~net-dbx/` 189