

Implementation of DDARC: Software Architecture for Debugging Distributed Programs

Sushma Rai¹, D.Sampath² and Srivathsa.N.S³

Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India

Abstract

Debugging and testing is a larger part of the effort spent in a software development cycle. Debugging a program is time consuming and is a continuous cycle of code modification and testing. The task of debugging depends on the environment, the language, the operating system; the problem; and more interestingly even the individual program. In this paper, we present a detailed implementation of software architecture for debugging distributed programs. The debugger based on the proposed architecture supports debugging based on a mechanism for controlling multiple processes, presenting the process states, grouping of processes for ease in controlling the distributed programs developed on PVM based computing paradigm.

1. Introduction and Related Works

The technological advancement in computing and development in microprocessor architecture along with networking systems opens up a new trend in developing high performance reliable cluster computers and distributed systems. Debugging sequential program is itself a challenging task and distributed programs pose more challenge due to various issues in controlling, reporting and detecting errors in program execution. There are independent efforts at the research and industry laboratories towards developing models and tools for debugging distributed programs based on conventional and new approaches [7,8].

The approach adapted in the development of debuggers for distributed programs differs [1,3] mainly from the single-image abstraction supported at different

levels of distributed system's abstraction either at hardware, low level software layer (like OS kernel), middle layer software architectures (like PVM, MPI, CORBA, et al). In recent times, various research groups [4,5,10,11,12] have realized the importance of testing and debugging procedures for successful implementation of software systems in distributed computers.

1.1 Issues in Debugging Distributed Programs

There are number of reasons which make debugging a distributed program more difficult compared to a sequential program. They are:

- Distributed programs have many foci of control.
- Without a synchronized global clock, it would be difficult to determine the precise order of events occurring in distinct and concurrently executing process [2,6].
- Distributed asynchronous programs are inherently non-deterministic. Therefore it is difficult to reproduce errors and test possible but improbable situations during debugging.
- Monitoring a distributed system alters its behavior. This is called probe effect in monitoring distributed programs.
- In a distributed program the number of processes are large and hence a method has to be evolved by which the details can be presented in a appropriate manner.
- Interaction between the system and user of debugging tool is more complex.

Design of a debugger for distributed programs have to address various issues associated with problems mentioned above. The debugger discussed in this paper is implemented for PVM based distributed programs;

¹ Currently at DDORG, Bangalore

² Currently at DBS Bank, Singapore

³ Currently at Sun Microsystems, Bangalore

therefore, we briefly explain the PVM environment in the following section.

1.2 PVM: Parallel Programming Environment

The proposed debugger in this paper is implemented for PVM based parallel programs under UNIX operating system environment. PVM supports execution of software systems on different machines/nodes in a user-configurable pool, and presents a unified, general, and powerful computational environment of concurrent applications. User programs written in C, C++ or FORTRAN access PVM through library routines for functions such as process initiation, message transmission and reception, and synchronization via barriers or rendezvous. PVM operates on a collection of computers (UNIX is most popular) connected by one or more networks, which is called the virtual machine. It is comprised of two main components, namely daemon processes and libraries, which are explained in the following paragraphs.

A daemon process is a process, which reside always in memory and waiting for some events to happen in order to continue its execution. One daemon process runs on each machine configured under the parallel virtual machine, which in turn is configured by the user. Each daemon process maintains a table of configuration and process information relative to the parallel virtual machine. User processes communicate with each other through the daemon process. They first communicate to their local daemon process via the library interface routines. Local daemon process then sends/receives messages to/from remote host daemons. Each machine must have its own architecture dependent version of PVM daemon process built and installed.

Library Interface Routines are simple subroutine calls that the application programmer may imbed in concurrent or parallel application code. These library routine provide facilities to: (a) Initiate and terminate processes; (b) Pack, send, and receive messages; (c) Synchronize via barriers; and (d) Query and dynamically change configuration of the parallel virtual machine. Library routines do not communicate directly to other processes, instead they send commands to the local daemon process and receive status information. Data format conversion (XDR) performed automatically between hosts of different architectures.

The rest of paper is organized as follows: In Section 2, we present the overall architecture and design of the DDARC based distributed debugger. In Section 3, we discuss the detailed specification and implementation techniques for the GUI module. The implementation details of the debugger DDARC is presented in Section 4. Subsequently in Section 5, we present the third component of the DDARC i.e., the slave module, and

finally in Section 6, the conclusion and future works are presented.

2. Architecture and Design of DDARC

Distributed Debugger Architecture (DDARC) is designed for debugging distributed programs and the specific implementation is carried-out for under PVM environment. The overall system is designed in such a way that each of the process of the distributed program is controlled by a mini conventional debugger or commercially available sequential debuggers such as *dbx*. This strategy adapted in the design has helped us to focus on all aspects of controlling mechanism and implementing trace facility for Inter-Process Communication (IPC) rather than re-designing the sequential debugging features. The overall system design is based on multilevel client-server model. We refer the main controlling module as *master module* and the client program, which controls the sequential debugging features such as *dbx* as *slave module*. The main features of DDARC are:

- Control mechanism for multiple tasks of distributed programs.
- Graphical user interface to ease the user interaction with DDARC.
- Facilities for grouping a set of tasks.
- Event tracing facility to handle the Inter-Process Communication based errors.

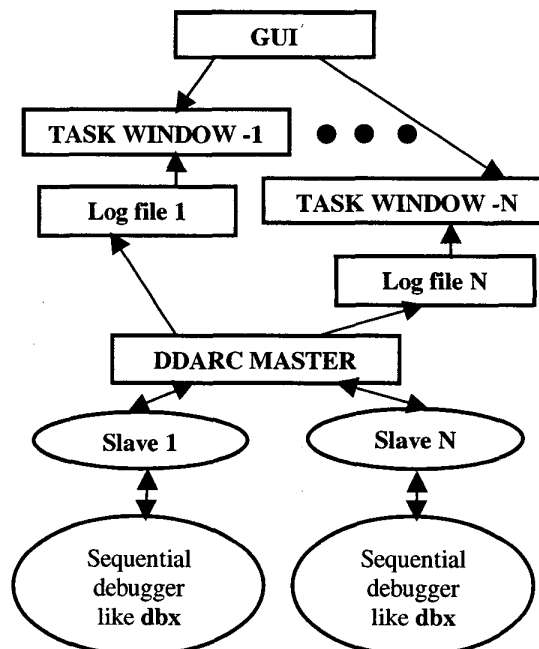


Figure 1. The multilevel software architecture of DDARC

The overall architecture of DDARC is given in Fig. 1. In order to support the above functions, the overall system is composed of the main modules as presented below.

Graphical User Interface (GUI): GUI receives the information such as task names, group names (where a set of tasks are grouped together for ease in controlling and monitoring), host names, log-file names and commands to the debugger from the user. It creates output windows for each task, displays the output of sequential debuggers on them. If a task or group is deleted it closes all the corresponding task output windows.

Master Module: Receives the information sent by GUI, process them, creates slave processes on the specified machines, and sends the information to slave processes through PVM interface mechanism.

Slave module: Receives the information from master through PVM, executes them with the available sequential debugger such as *dbx* and sends the output to master through PVM interface mechanism.

3. Design and Implementation of GUI Module

One of the most important parts of a software system to be successful in usage is the User Interface components. The GUI provides User Friendly Interface for task level debugging, grouping techniques, selection of commands through single window and displays the results of debugging on different windows for different tasks. Graphical User Interface (GUI) provides a single window through which user can select task names, group names, commands to be send to individual task debuggers, and host names on which user wishes to create the slaves and subsequently execute the tasks. In order to facilitate ease in issuing commands to the master debugger by the user, the GUI module provides simple and elegant interface design as described below.

Various Sub-Modules Used in GUI module for different functions.

1. *To select the task names and host names:*
It creates an entry window for selecting the task name and the corresponding host name on which it is to be created and executed.
2. *To select the group names:*
It displays a message for the user to select any of the groups, by clicking the mouse on group buttons provided in the GUI. On selecting a group it disables that button, so that the user can select any of the remaining groups for future grouping.
3. *To add task names and host names to string to send them to DDARC master:*
It adds the task names, group names and the host-names selected by the user to a string separating them with ":" to be sent to DDARC master module.
4. *To delete a task:*

It gets various task names to be deleted, and sends to DDARC master module. Also destroys the output windows for those tasks.

5. *To create a group:*
It gets the group name, and also the task names which are to be added to those groups, and sends them to DDARC master module in the form of a string with ":" separating them.

6. *To delete a group:*
It gets the group name to be deleted, and sends the group name and all the tasks belonging to that group to DDARC master. Also closes all the display windows for the tasks of the group, which are already terminated.

7. *To create a display window for each task and display the information received by the master module:*

It creates output display windows for each task added with task name as title, displays the contents of the respective log files in those windows.

8. *To receive and send commands:*
It displays a message for the user to select a command. As the user selects a command button provided, it sends those commands with a new line character appended at the end to DDARC master module.

3.1 Implementation Details of GUI Module

The Graphical User Interface (GUI) designed for the distributed debugger provides a single window interface between the user and the distributed debugger so that user can control multiple tasks on different machines using a single window. GUI has three main sections. The first section contains the selection menus. User can follow either single task level debugging or group level debugging according to the requirements. The TASK menu provides the options such as *Add Task* and *Delete Task*. After the selection of a menu button, GUI will pop up a data entry window, in which user can enter the task name and the host name on which he wants to execute the task. Similarly user can delete a task from the task list using *Delete Task* option.

The group level debugging allows the user to group multiple tasks. The *Group* menu facilitates the user for creation of different groups, addition of tasks to the existing groups, deleting of a group or task from a group. On selecting the *Create Group* option, GUI will display a message to select any of the groups from *G1* to *Gn*. On selecting the group, user can add any number of tasks to that group or delete tasks from the group. Similarly user can delete any of the group which is already created. After selecting the task/group and the hosts GUI will pop up output windows to each of the tasks. It polls the log file if the file exists, and displays its contents on the respective task output windows. GUI then displays a message to enter commands to the debugger. In order to debug at the task level, the user has to select the *Tasks*

section of the main GUI window, which will enable the system to send the respective command to the debugger. And, for the group level debugging the user can press any of the buttons in the Groups section of the main GUI window. In the case of group level debugging the GUI sends the same command to all tasks of a group. The commands are sent to the serial debuggers and the results of those, which are available in the corresponding log files, are displayed on the respective task output windows. The flow chart for GUI module is given in Fig 2.

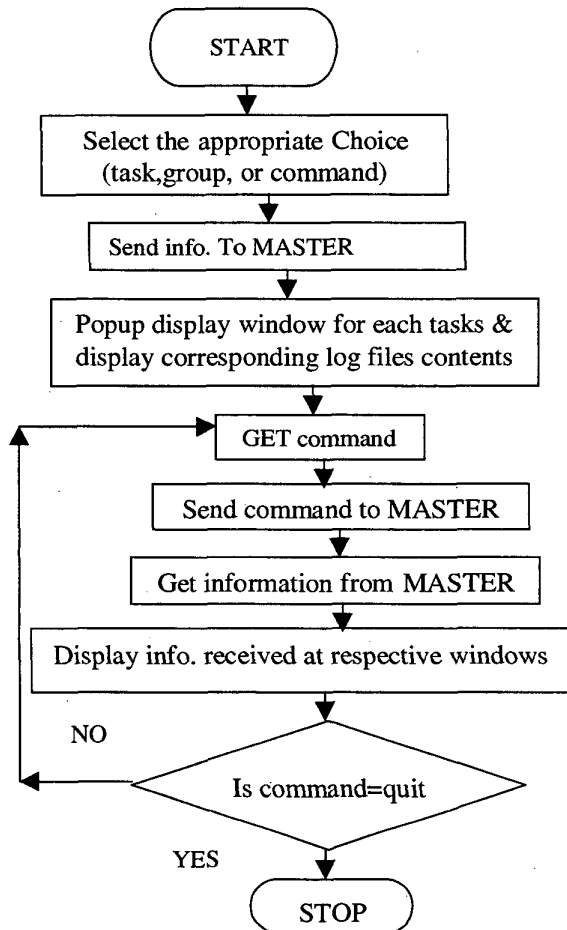


Figure 2. Flow Chart for GUI Module

4. Design and Implementation of Master Module

DDARC Master module is the central controlling module of the overall DDARC system. DDARC master module receives the task name/group name, machine name, log-file name, and the commands to either create a task/group or to delete a task/group from the GUI. The master module either creates a task/group of tasks or

terminates the tasks/group of tasks according to the command issued by the user. It also creates a log file for each of the created task. This module is also designed as an interface between the GUI and the actual tasks by receiving the command from the GUI and sending it to the particular task after preprocessing and parsing. Further, this module also communicates outputs from the slave processes back to the GUI by storing them in their respective log files for the GUI to display on the corresponding windows. The Master Module has the following functions and composed as sub-activities.

Sub-Modules of Master Module:

1. *To read the inputs from GUI and to store them in a Data Structure:*

It reads the task/group names, log-file names, host-names, which are sent by the GUI in the form of a string, separate by ":", parses the string and stores the details in either a task structure or group structure.

2. *To create DDARC slave programs on specified hosts:*
The creates the slave programs on hosts specified by the GUI, stores the task identifications (Ids) in an array.

3. *To send the information read from the GUI to the DDARC slave programs created:*

It starts the PVM Daemon process and sends the host name or command read from GUI to all the slaves using PVM calls.

4. *To read the information send by DDARC slave programs:*

It gets the information sent by the slaves through PVM.

5. *To write the information read by the slave programs into the log files:*

Creates the log files for each task, and writes the information read from the slaves into the files.

6. *To read the commands to the debugger from GUI and to send them to the respective slave programs:*

Gets the commands continuously and passes them to slave programs using PVM calls.

7. *To append the output of slave programs to the respective log files:*

Gets the outputs of slaves through PVM and appends the information read into the corresponding log files.

The overall design of the Master Modules of DDARC is given in Fig 3. The Master module communicates with the slave modules through any software system (in this work we have used PVM) that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resources.

4.1 Implementation Method for Master Module

The master module is the central part of the DDARC system, which supports complete controlling mechanism for debugging. The implementation detail of the system is explained below with the pseudo-code. Master sends the

information received from the GUI to the respective slave programs to execute under sequential debuggers, and receives the output of the debuggers sent through the slave programs, writes it into the respective log files.

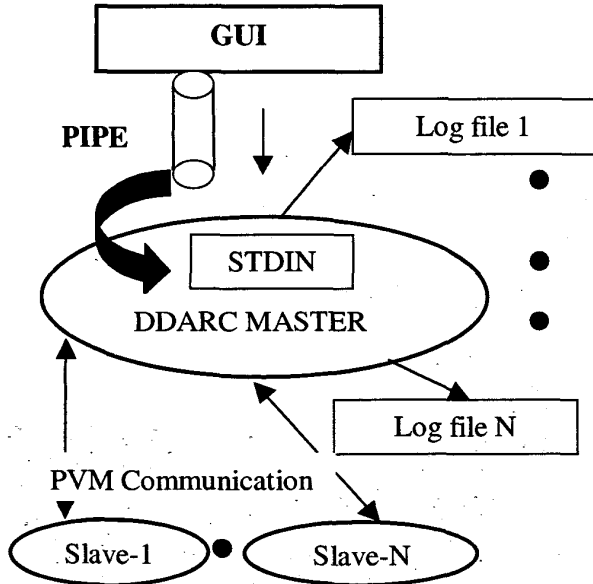


Figure 3. Design of Master Module

Pseudo Code for DDARC Master Module

```

/* The main function of the Master Module */
void main (void)
{
while (inputString!=over) {
    • Call      GetInputFromGUI(InputString,
      tasklist[Nslaves])
    • Initialize SlaveName
    if (Task Level debugging) {
      if (AddTask) {
        • pvm_spawn (taskname, hostname, taskid)
        • Update the structure tasklist
      }
      if (Delete Task) {
        • Find the TaskID of given Task
        • Kill that task
        • Update the structure tasklist
      }
    }
}
if (Group Level Debugging) {
  if (CreateGroup) {
    for (i=0; i<Nslaves; i++)
    {
      • Call pvm_spawn(SlaveName, Grouplist,
        hostname[i], TaskTds)
      • Update the structure grouplist
    }
  }
}
}

```

```

}
if (DeleteGroup) {
  for (i=0; i<Nslaves; i++) {
    • Kill all the spawned tasks
    • Update the structure grouplist
  }
}
}
}
}

```

There are many other functions, which are part of the master modules designed to work in a cohesive manner.

5. Design and Implementation of Slave Module

DDARC slave program is designed to control the actual debugging feature for the multiple processes of the distributed program through sequential debuggers. The Slave Module establishes the connection between the sequential debuggers and the DDARC master program. Each DDARC slave program creates a child process to execute a sequential debugger with the given process (task). They receive the names of programs to be executed from the master through any computational environmental which is used to create and execute parallel applications and communicates to the child process through a pipe to execute under serial debugger. The commands received from the master module is directed to the actual sequential debugger which controls the task and the resultant output from debugging process is passed back to the GUI interface through the master module. The outputs from the serial debuggers are sent to the DDARC master module.

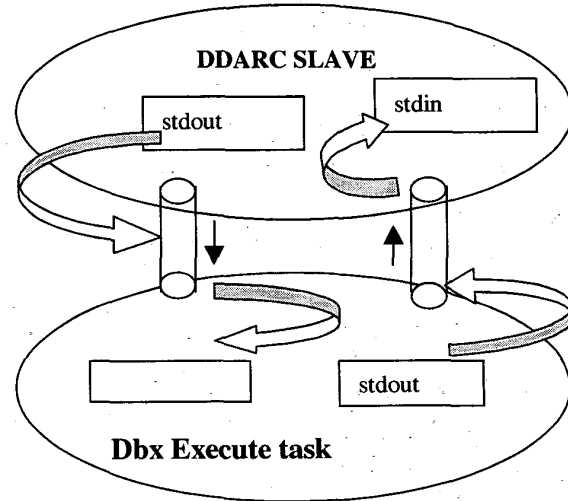


Figure 4. Design of Slave Module

The design of slave DDARC module is given in Fig. 4. The slave DDARC module consists of various sub-modules as give below.

The sub-modules used in Slave process:

1. To receive the information sent by the DDARC master program:

It receives the information sent by the master through PVM

2. To fork a child to execute the tasks specified by the DDARC master with the sequential debugger and to establish the connection between it and the child forked by it through pipes.

3. To read the output of the serial debugger.

Reads the output of the slaves though a pipe

4. To send the output of the serial debuggers to DDARC master module:

Sends the information read from the child to DDARC master though PVM.

5.1 Implementation Method for Slave Module

DDARC Slave program communicates through pipe mechanism with the child process. The name of the program to be executed, received by the DDARC slave program is passed to the debugger through the outgoing pipe, and the output of the sequential debugger, is read through the incoming pipe. The child closes the write descriptor of the pipe from the slave and read descriptor of the pipe to the slave. It also closes stdin, and duplicates the read descriptor of the pipe from the slave (duplicates the stdin). Also the child closes the stdout and stderr and duplicates the write descriptor of the pipe to the slave program. DDARC slave module (i.e., the parent) closes the read descriptor of the pipe from the child and write descriptor of the pipe to the child.

Pseudo code for DDARC Slave Module

void main (void) {

- Find the task id of DDARC_MASTER program using PVM library routine pvm_mytid and store in Ptid.
- Call RecvFromMaster (Ptid, PgmName)
- Create two pipes Pin and Pout
- Switch (fork ()) {
 - case -1:
 - Print the error message
 - Exit
 - case -0: /*Child Process*/
 - Close write file descriptor of Pin and read file descriptor of Pout at the child process.
 - Close the stdin, stdout and stderr at the child process.
 - Dup the read file descriptor of Pin and write file descriptor of POut.
 - Exec the task PgmName with dbx.

```

default: /*Parent Process*/
• Close read file descriptor of Pin and write
file descriptor of POut at the parent process.
• Call ReadFromDbx (POut[0], Message)
• Call SendToMaster (Ptid, Message)
do
{
• Call RecvFromMaster(Ptid, Command)
• Call CommandToDbx(Pin[1], Command)
• Call ReadFromDbx(POut[0], Message)
• Call SendToMaster(Ptid, Message)
}
while (command <> "QUIT") }
• exit PVM by using PVM library routine
pvm_exit()

```

/* RecvFromMaster */

```

RecvFromMaster (PTid, String) {
• Receive the string sent by DDARC_MASTER
program with TID PTid using the PVM routing pvm_
recv()
• Unpack the received character string and store it in
String using PVM routine pvm_upkstr()
}

```

/* SendToMaster */

```

SendToMaster(PTid, Message) {
• Clear the current buffer (if exists) or create a
new buffer using PVM library routine
pvm_initsend()
• Pack the output of dbx to be sent to
DDARC_MASTER using the PVM library
routine pvm_pkstr()
• Send the packed message to DDARC_MASTER
using the PVM library routine pvm_send()
}

```

/* CommandToDbx */

```

CommandToDbx(FileDescriptor, Char * Command) {
• Write the string Command to the pipe referenced by
the file descriptor File Descriptor.
}

```

/* ReadFromDbx */

```

ReadFromDbx (FileDescriptor, Message) {
• Read from the pipe referenced by the file descriptor
File Descriptor and store in a char array message
}

```

6. Conclusion and Future Work

In this paper we have discussed the issues of debugging distributed programs and presented some features which are essential for debugging distributed programs. Subsequently we have presented the architecture of the distributed debuggers designed for PVM based parallel programs. We have also presented the detailed design functions and some implementation

techniques for our distributed debugger DDARC. We have implemented this debugger on IBM SP-2 distributed system under AIX (UNIX variant) operating system. Though, the system has been implemented successfully and being used it has some limitations: (1) GUI module does not provide all the available commands to the user; and (2) DDARC Master Module controls all slave modules together, therefore it is an overloaded and the response time might degrade as the parallel tasks increases. As a future work we want to incorporate performance-debugging feature with DDARC as one of the main advantage of parallel programming is better performance and hence we believe that a debugger for such environment should support features for performance debugging.

References

- [1] Araki, K.; Furukawa, Z.; Cheng, J.: A General Framework for Debugging, *IEEE Software*, Volume: 8.3, May 1991, pp. 14-20.
- [2] Chandy, K.M, Lamport.L: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, Feb. 1985, pp.63-75.
- [3] Cheung, W.H.; Black, J.P.; Manning E.: A Framework for Distributed Debugging, *IEEE Software*, Volume: 7 1, Jan. 1990, pp. 106-115.
- [4] Chul-Eui Hong; Bum-Sik Lee; Gi-Won On; Dong-Hae Chi: Replay for Debugging MPI Parallel Programs, *Proceedings of MPI Developer's Conference*, 1996, pp. 156 -160.
- [5] Jianxin Xiong; Dingxing Wang; Weimin Zheng; Meiming Shen: BUSTER: An Integrated Debugger for PVM, Proc. of *IEEE Second International Conference Algorithms & Architectures for Parallel Processing (ICAPP' 96)*, 1996, pp.124-129.
- [6] Lamport, L.: Time, Clock and the Ordering of Events in a Distributed System, *Communication of the ACM*, 21(7), July 1978, pp. 558-565.
- [7] LeBlanc,T.J: Parallel Program Debugging, Proc. of 13th Annual International Conference on Computer Software and Applications Conference (COMPSAC 89), 1989, pp. 65-66.
- [8] Lourenco, J.; Cunha, J.C.; Krawczyk, H.; Kuzora, P.; Neyman, M, Wiszniewski, B.; An Integrated Testing and Debugging Environment for Parallel and Distributed Programs, *Proceedings of the 23rd EUROMICRO Conference on New Frontiers of Information Technology*, 1997, pp. 291-298.
- [9] MPI: A Message Passing Interface Standard, June 1985.
- [10] Paik, E.H.; Chung, Y.S.; Lee, B.S.; Chae-Woo Yoo: A Concurrent Program Debugging Environment using Real-time Replay, *Proc. of International Conference on Parallel and Distributed Systems*, 1997, pp. 460-465.
- [11] Scholten, H.; Posthuma, J.: A Debugging Tool for Distributed Systems, *IEEE Region 10 Conference on Computer, Communication, Control and Power Engineering (TENCON '93)*, Vol. 1, 1993, pp. 173 -176.
- [12] Tarafdar, A.; Garg, V.K.: Debugging in a Distributed Word: Observation and Control, *Proc. of IEEE Workshop on Application-Specific Software Engineering Technology*, 1998. (ASSET-98), 1998, pp. 151-156.