A Portable Debugger for Parallel and Distributed Programs[*]

Doreen Cheng[†]
dcheng@prpa.philips.com

Robert Hood
rhood@nas.nasa.gov

Computer Sciences Corporation
NASA Ames Research Center, Mail Stop 258-6
Moffett Field, CA 94035-1000

Abstract

We describe the design and implementation of a portable debugger for parallel and distributed programs. The design incorporates a client-server model in order to isolate non-portable debugger code from the user interface. The precise definition of a protocol for client-server interaction facilitates a high degree of client portability. Replication of server components permits the implementation of a debugger for distributed computations.

Portability across message passing implementations is achieved with a protocol that specifies the interaction between a message-passing library and the debugger. This permits the same debugger to be used both on PVM and MPI programs.

The process abstractions used for debugging message-passing programs can be adapted to debug HPF programs at the source level. This permits the meaningful display of information obscured in tool-generated code.

1 A need for uniformity

A large gap exists between user needs and existing support for debugging parallel and distributed programs. In an environment with diversified computational facilities such as the Numerical Aerodynamic Simulation (NAS) Division at NASA Ames Research Center, the problems are compounded by the following issues.

- Each system has a distinct debugger, often with sophisticated user interface. This variation and the amount of time that must be invested to learn a new user interface, contribute to the user's reluctance to invoke a debugger.
- Some computational platforms, such as computations distributed across a network, have inadequate debugging tools.
- New programming interfaces, such as *PVM* [13], *MPI* [10], and *High Performance Fortran (HPF)* [6], hide mapping information needed by debuggers.
- Existing debuggers often provide information at a low level of abstraction rather than at the level of the program source [12].

In defense of vendors, the market has not demanded better debugging tools. That is, vendors are not losing sales of new systems over inadequacies in their debugger. Thus, they allocate fewer resources to debuggers than would be necessary to address these issues. Rapid evolution of hardware and software for parallel systems has kept debugger developers busy just porting existing code to new systems. Dynamic evolution of parallel and distributed programming models has made this situation even worse. This situation is likely to last for a while; thus, it seems that outside influence must be used to improve debuggers for parallel and distributed programs.

Our strategy for solving these problems is to employ a client-server model in the debugger implementation. The precise specification of the interaction protocol between the client and server will permit client user interfaces to be paired with a variety of server implementations, thus easing the task of porting a consistent user interface between platforms. Inclusion of appropriate mechanisms in the interaction protocol can also foster the implementation of debuggers that provide information and operations at a level meaningful to the user.

Our approach differs significantly from previous parallel and distributed debuggers. Most of the prior work falls into two main categories:

- trace-based (*i.e.*, post-mortem) debugging systems, some of which have addressed portability, such as Paragraph [3] and AIMS [15], and
- the parallel and distributed analogs of conventional state-based debuggers, such as *ipd* [5] or *Prism* [14] which have not tried to be portable.

The only exceptions to this categorization that we could find are distributed debuggers that layer the user interface on top of complete vendor debuggers [2][7][8]. While this

[*]This work was supported through NASA contract NAS 2-12961 and NASA's High Performance Computers and Communications Program.

[†]Doreen Cheng's current address is Philips Research Palo Alto, Philips Electronics North American Corp., 4005 Miranda Avenue Suite 175, Palo Alto, CA 94304.

is a step toward portability, the distributed debugger implementation is complicated by the need to accommodate variation in the user operation semantics of the vendor debuggers.

Using our client-server approach, vendors who choose to do so can focus their development effort on implementing servers and need not invest in costly user interface work. A user site would be able to build a client user interface that is tailored to its user community and the programming model it uses.

To demonstrate the value and feasibility of our approach and to provide NAS users a useful debugger, we are developing a prototype: The Portable Parallel/Distributed Debugger, *p2d2*. It has the following properties:

- it supports debugging parallel and distributed computations, as well as traditional sequential programs,
- it provides the same user interface on all platforms,
- it can accommodate programming models not supported by a vendor, and
- its user interface is tailored to the specific needs of users debugging computational fluid dynamics (CFD) codes.

In this paper, we describe the design and implementation of *p2d2*. In the process, we provide an overview of the client-server interaction protocols. Related papers provide more detailed descriptions of the protocols [1][4].

2 Isolating implementation dependencies

One of the main goals of our project is to build a debugger that is portable across a variety of platforms and programming models. There are a number of factors that stand in the way of this goal. In this section we will discuss those barriers and a strategy for overcoming them.

2.1 Barriers to portability

At the most basic level, a debugger's job is to provide mapping services between the user's conception of the program at the source level, and the machine version at the object level. This is typically done by taking a user request in terms of the program source and

- mapping it to a request at machine level using symbol table information provided by the compiler,
- getting the operating system to perform the request, and finally
- mapping the OS reply back to source level.

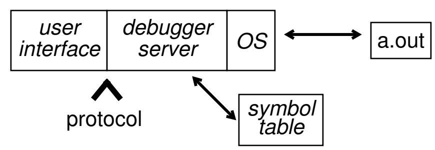
In performing this process, a debugger relies on its execution environment for many services. These services may depend on:

- the target machine architecture for things such as trap instructions used in breakpoint implementation,
- the operating system for process control and address space access,
- symbol table format for mapping information,
- the thread and synchronization abstractions used,
- the message passing implementation used, and
- the HPF implementation for information about how data has been distributed.

The basic approach that *p2d2* uses to provide portability across all of these dependencies is to isolate them from the part of the code that needs to be portable—the user interface. By employing this strategy we can not only achieve portable user interface code, we will also have a great deal of flexibility in choice of target programs through the use of an "interchangeable parts" philosophy in accessing the non-portable code.

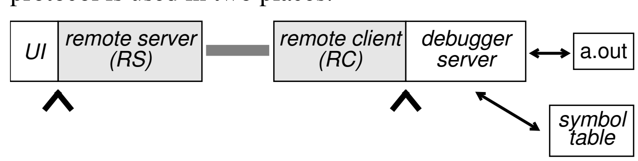
2.2 The first step—a debugger for one process

The basic strategy of the *p2d2* design is to use a client-server model to partition the debugger. The *debugger server* will contain the platform-dependent code while the *client* will have the portable user-interface code. It is intended that the server be implemented by the platform's vendor. The client, on the other hand, can either be implemented by NAS (or some other user community) or by the vendor. The following diagram illustrates the division:



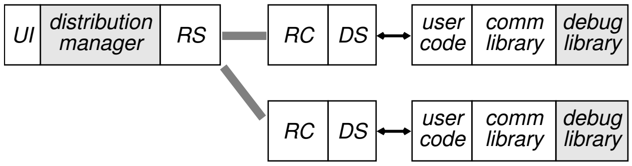
One of the most important parts of the *p2d2* design work is the definition of the protocol for interaction between client and server. This is discussed in Section 3.

This client-server model makes it quite easy to provide for remote debugging, where a debugging session on a workstation is controlling the execution of a job on a remote supercomputer. We need only interpose a communication layer between the client and server in such a way that the UI client still sees a server and the remote server sees a client. This can be done by providing a debugger server implementation that translates operations on objects into messages that are sent to a remote debugger client implementation that takes the message requests and translates them into server requests. Note that the client-server protocol is used in two places:



2.3 Accommodating distributed execution

For *p2d2* to be used in isolating problems in distributed computations, it must be able to get information concerning the execution from the communication library. For example, if the user attaches *p2d2* to the root of his computation, then the debugger will need to determine what processes on what machines are participating in the execution. One way to accomplish this is to secure additional mapping information from the communication library:



In this scenario, the communication library reports important events, such as process creation, by making a procedure call to the *debug library*. The debug library, in turn, records the information in a table and makes it available to the client's *distribution manager*, which is responsible for keeping track of how processes and process groups are distributed on the network. The distribution manager is described in more detail in Section 7.1.

For this approach to work, we must define the protocol between the communication library and the debug library. Section 4 discusses what this protocol must contain.

2.4 Accommodating HPF preprocessors

If *p2d2* is used on HPF target programs, then it will need assistance in responding to user queries such as "`print A(i,j)`". The problem is that the HPF preprocessor will distribute array `A` across the processors in some way that will not be indicated in the symbol tables for the individual executables. In effect, the HPF preprocessor is an additional compilation step whose mapping from source to object needs to be recorded in a symbol table. In its most general form, this symbol table can change dynamically, because the program can redo distribution and alignment during execution.

While the HPF definition provides for dynamic access to distribution information, it is still desirable to have the HPF library interact with a debug library in a manner analogous to the one used for distributed execution. Thus, we employ another protocol for describing the interaction between the HPF run-time library and a *p2d2*-provided debugging stub running in the target address space. This protocol is discussed in more detail in Section 5.

2.5 The resulting architecture

If we merge the design ideas from the preceding sections the result is the architecture shown in Figure 1. This approach has two advantages.

- First there are several options for a particular component's implementation, as indicated in a balloon attached to a component. (Note that the vendors named in the diagram are for illustrative purposes only and do not indicate a commitment on their part to provide the indicated software) By appropriate choice of components out of these balloons, debuggers that share a common user interface can be achieved for a variety of target platforms.
- Second, a vendor can choose to focus its efforts on a debugger server, while giving user communities and third parties the possibility of developing application-domain specific user interfaces.

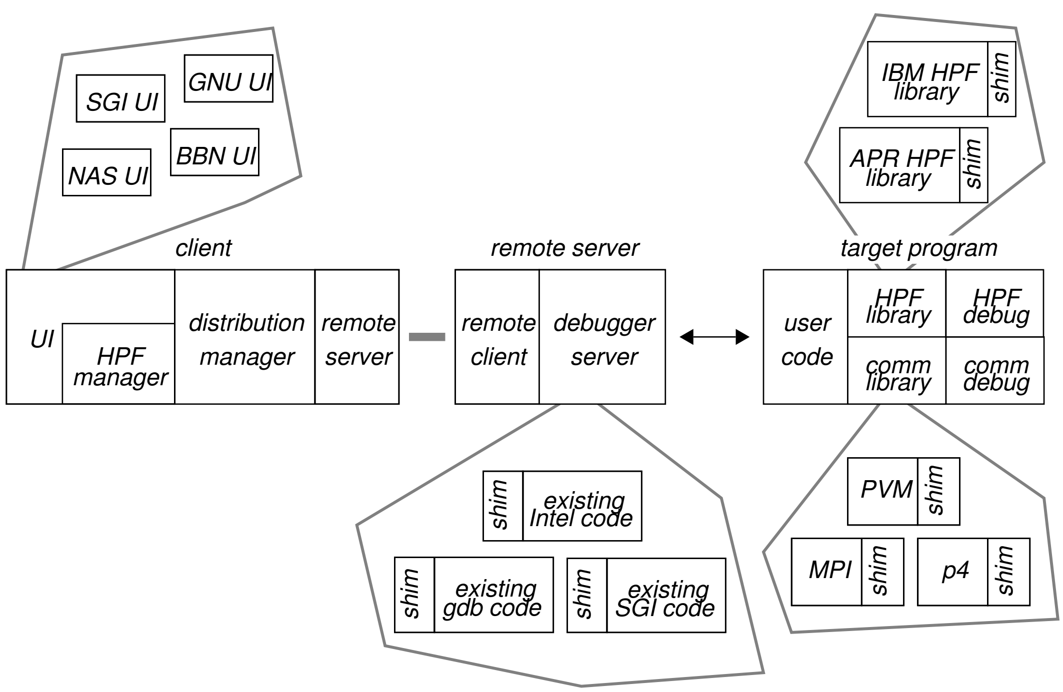


FIGURE 1. Architecture of *p2d2*

We have strived to simplify the re-use of existing debugger and communication library code into the *p2d2* debugging environment. Some modification of such code will be required, however. The modification sites are indicated by boxes labelled "*shim*" in Figure 1.

For *p2d2*'s fundamental strategy of client-server partitioning to work, we must precisely define the interaction protocols that were identified earlier. In the next three sections we describe those protocols in more detail.

3 The debugger server protocol

The *debugger server* (`Ds`) protocol outlines how the user interface client can access debugging abstractions provided by the server. In the interest of flexibility and ease of modification, we made the client-server interface object-oriented. Thus, the server exports a collection of C++ objects that are an abstraction of the executable being debugged and the state of the process running it.

3.1 `Ds` objects

The `Ds` objects are designed to approximately match user abstractions of the debugging process. Besides being a convenient way to represent user abstractions, another consideration for the design of `Ds` objects is that it be straightforward to implement them in terms of operating system objects and mechanisms. In general, the objects fall into the following five categories.

Program objects provide an abstraction for the mapping information between the source representation of the program and its object representation. Example objects here include:

- the `Executable` object, which corresponds to the source-level program being executed, and
- the `Location` object, which encapsulates a point in the program source. It is used when communicating such things as where to place a breakpoint and where execution was stopped.

State objects are used to encapsulate the information that maps between source-level dynamic abstractions and their implementation in the operating system. For example:

- The `Process` object encapsulates the Unix notion of a process, and
- the `Stack` object represents a collection of function activation records, each of which is represented by a `Frame` object.

Event trap objects provide a mechanism for asynchronous notification of events that take place during execution. In general, the client creates an event trap in order to indicate that it wants to be notified when a particular event takes place. Example traps include:

- the `SinglestepTrap` which indicates a single source instruction has been executed,
- the `BreakpointTrap` which triggers when an instruction at a specific location in the code is about to be executed, and
- the `TimerTrap` which goes off after a specified period of time.

To create an event trap, the client provides a callback function and an argument in a manner analogous to that used in the X toolkit intrinsics [11]. When the event occurs, the server will invoke the function, passing the argument.

Instrumentation objects are used to specify code to be executed at event traps. They permit the efficient implementation of conditional breakpoints and user modifications during a debugging session. In addition to code in source languages such as C and Fortran, instrumentation can be provided in a "debugger scripting language" defined by the protocol. The scripting language was designed to facilitate the downloading of instrumentation into the target process, either as a compiled patch, or as byte-codes that can be executed by an interpreter resident in the process. Alternatively, the instrumentation can be interpreted in the server's address space.

Expression evaluation objects provide abstractions for producing and maintaining displays of data values. They include `Expression` objects and a hierarchy of `Type` and `Value` objects. The design of these objects was influenced by the desire to support the display of large data structures efficiently. For example, the type and value information for components of a large data structure can be constructed part-by-part on demand. In addition, there is a mechanism that permits a remote implementation to optimize the amount of message traffic that will take place when many evaluations are being performed.

Unfortunately, space limitations in this paper do not permit a complete description of the `Ds` objects. Another paper provides more details [4]. To illustrate the use of `Ds` objects, we present examples in the next two subsections.

3.2 Example: setting a breakpoint

Suppose that the client has already created an `Executable E` and a `Process P`, and that the user has requested that a breakpoint be put at line 73 of file "foo.f". In response, the client performs:

```
loc  = E->createLocation("foo.f", 73)
trap = P->createBreakpoint(bkptCB, arg,
                           loc);
```

requesting that `bkptCB` be called when the `loc` is reached.

Later, when the breakpoint is hit, the server performs:

```
bkptCB(<struct containing loc>, arg);
```

The client can then update its display to show the current location of the program counter.

3.3 Example: maintaining a data value display

Suppose that the user wants a data display for the variable `struct foo s`. The client UI wants to put up a window capable of showing 15 lines of `s`. It needs to retrieve type information and values to initialize the display. It first requests:

```
Type *t = fn->getType("s");
```

after which the expression `t[i]` has the type information for the ith leaf in the flattened representation of a `struct foo`. To get the values for `s`, the client requests:

```
Expression *e = frame->
                    createExpression("s");
Value      *v = e->getValue();
```

after which `v[i]` is the value of the ith leaf in the flattened representation of `s`. In addition to this request, the client will advise the server of what values are being displayed:

```
v->displaying(0,14);
```

While the server can choose to ignore this advice, it can be used to optimize the amount of communication between client and server. It allows an ambitious server to anticipate client requests and transmit data values in bulk before they are requested individually.

When the user continues execution the server will notify the client that the displayed values are stale. When execution hits a breakpoint trap, the server will notify the client via the mechanism described in the previous example. The client can then update its data display for `s` by accessing objects `v[0] ... v[14]`.

3.4 Abstractions for multithreaded executions

Additional operations are required to debug interactions between concurrently executing parts of a shared-memory parallel program. For example, a user may ask which threads have reached a barrier or which thread is holding a particular lock that is wanted by another thread. The user may also want to know whether a variable is shared and which threads are sharing the variable.

Parallel debuggers provided by computer vendors support the operations suitable for their own platforms. The debugger server protocol needs to be extended to include the abstractions for these operations. The following are a few examples of the extensions.

- the `Queue` object encapsulates a wait queue for threads, and provides member functions to access its entries,
- the `Barrier` object represents a synchronization barrier, and provides member functions to access arrival status for the threads,
- the `Variable` object has additional members to tell which threads can access it and the type of access each thread can have, and
- the `Process` object supports a "re-execute using trace" operation for isolating race conditions.

The details of the protocol extensions for debugging shared-memory parallel programs can be found in a NAS technical report [1].

4 The communication library protocol

The purpose of the communication library protocol is to give the debugger access to information about the message passing state and the topology of participating processes in a distributed computation. For example, when a program deadlocks, the user needs to find the reason by examining the messages in the queues and the messages for which some processes are waiting. While locating a bug, the user may need to examine the contents of a message. The user may also want to examine the message-passing patterns, frequency, and data volume while optimizing program performance. The communication library protocol is designed to support these operations; it must be able to answer the following questions.

- What processes on which machines are participating in the computation?
- What messages are in a particular message queue?
- What is in a particular message?
- What source line is the origin of a particular message?

The communication library protocol has two components:

- a set of procedural entry points for the library, for example PVM, to register events with *p2d2*, and
- a set of objects that *p2d2* can use to access the internal data structures of the library at debugging time.

This section describes the main points of the protocol; details can be found in a technical report [1].

The protocol specifies a collection of debugger library entry points and requires the communication library to register run-time events. For example, when a new process is created in a computation, the communication library function that created the new process should make the following call to register the creation event:

```
MP_RegisterSpawn(machineName,
                 executableName,
                 osAssignedId,
                 toolAssignedId);
```

The debugger can then use this information to attach a debugger server to the new process and allow the user to interact with its execution.

To support process-group based debugging, the protocol requires a communication library to register events related to group creation, termination, and membership changes. For example,

```
MP_RegisterNewGroup(group, processes);
```

To answer user questions, such as "Show the message header for which process `P1` is waiting" and "Show the header of the i^{th} message in the input queue of process

P2," the debugger must access the internal data structure of a communication library. To isolate the semantic and implementation differences of communication libraries, objects representing user-world entities are chosen as abstractions. For example, in addition to the `Queue` objects listed in the Section 3.4, a `Message` object represents structured data being passed between processes provides member functions to access information such as the sender, receiver, size, and the tag of the message. The library developers can choose whether or not to support the member function and how to access such information. To support breaking deadlocks, message queue objects provide member functions to send and receive messages.

To support debugging operations such as "Show the source line responsible for the message send," the protocol suggests that the library provide a debugging version of routines for message send/receive and for global operations. The debugging versions will encode sufficient information in the message header for mapping back to the source line or machine instruction.

To allow debugging only selected processes, the protocol defines an environmental variable `MP_DEBUG`, and suggests that communication libraries pass it to all processes spawned by library spawning routines. This variable tells the newly spawned process whether it should be executed under a debugger server control (see Section 7.2 for its use).

5 The distribution preprocessor protocol

The purpose of the distribution preprocessor protocol is to give the debugger access to information about how aggregate objects get distributed in an HPF program by the preprocessor or compiler. In addition to being useful for displaying data values in the user interface, examining how data is actually placed in the memory is also necessary for performance optimization. The distribution preprocessor protocol is designed to support such needs. It has two parts: a set of procedural entry points for the preprocessor to register events to *p2d2*, and a set of procedural entry points for *p2d2* to access the data structures generated by the preprocessor.

The designers of HPF have already taken a major step toward the protocol specifications by requiring that HPF libraries include entry points (callable from the target) that will yield information about the current alignment and distribution. For example, the entry points

- `HPF_alignment(...)`
- `HPF_template(...)`, and
- `HPF_distribution(...)`

give *p2d2* a way of extracting the necessary distribution and alignment information at debugging time.

The debugger also needs a mechanism for being notified when the distribution information changes. This is handled by having the HPF library routines that perform alignment and distribution make a call to the HPF debug library immediately after the data is realigned and/or redistributed. Examples of such registration routines are:

- `HPF_RegisterReAllign (...)` and
- `HPF_RegisterReDistribute (...)`.

Although HPF supports implicit message passing, users as well as compiler and tool developers may want to debug the actual communication generated by a compiler. For this reason, the HPF run-time library that performs the message passing should adhere to the communication library protocol of Section 4.

6 User interface design issues

In addition to the protocol definitions described in Sections 3-5, the NAS effort includes the implementation of a prototype distributed debugger along the lines of the architecture described in Figure 1. This section and the next discuss some of the issues encountered in the design process.

Besides supporting traditional source-level debugging operations such as breakpoints and single-stepping, the *p2d2* user interface is designed to suit NAS users, to support navigation and selection among a large number of processes, and to accommodate vendor-specific user interfaces. This section describes these features.

6.1 Breakpoint visualization

The user community at NAS builds computational fluid dynamics (CFD) programs. The *p2d2* debugger supports visualization of the program state at breakpoints in execution through the use of their familiar visualization tool set FAST. This is accomplished by having the user interface pull values out of the program, format them, and submit them to FAST for display.

6.2 Process navigation

A process spawning tree is used for navigation and selection. The distribution manager mentioned in Section 2.3 informs the UI of events such as process and process group creation, termination and status changes. When a process is created by the application, the new process is added into the spawning tree. When a process terminates, its representative node in the tree is grayed.

The spawning tree display maintains a collection of process groups of interest to the programmer. These include not only the process groups appearing in the computation, but also auxiliary ones created by the user during the debugging session. These supplemental groups can be edited dynamically by selecting and deselecting processes and groups in the spawning tree display.

The currently selected process group is displayed in the spawning tree by highlighting its members. It is deemed to be the current focus process group, meaning that, by default, it will receive requests made in the user interface. For example, when the *Continue* button is pressed, it will be forwarded to all of the processes and groups currently selected in the spawn tree continue execution.

6.3 Vendor-provided user interface components

Logical and performance debugging often require examination of system data unique to a particular vendor. *P2d2* provides two mechanisms that facilitate the use of vendor-provided widgets for abstractly displaying such data.

The first mechanism enables the debugger server to register modifications that it wishes to make in the *p2d2* user interface. For example, a server can request *that p2d2* change the characteristics of a button or add a button to a menu with callback functions registered. When this button is activated, the callback will be made.

The second mechanism supports the creation of vendor provided user interface widgets in the same address space as the server. In order to be consistent with the local case, a remote `Ds` client program will have a main X event loop and will dispatch user interface events of interest to the server launched widget.

7 Distributed debugger design issues

In the course of designing *p2d2* we encountered several problems that stem from the distributed nature of the target computation being debugged. In this section, we describe two of those problems and the solutions we adopted.

7.1 Distribution objects

In a heterogeneous distributed computing environment, the debugger must address:

- the mapping of user requests into operations provided by a set of remote debugger servers, and
- the aggregation of the responses of the remote servers to produce a result at level of abstraction that is helpful to the user.

In *p2d2* these tasks are accomplished by the *distribution manager* working in concert with the remote implementation of `Ds` objects discussed in Section 2.2. In effect, the distribution manager is responsible for the scattering of debugging requests and the gathering of responses. The remote `Ds` implementation takes care of the mechanics of getting `Ds` operations communicated across the network to a vendor `Ds` implementation on a remote machine.

There are five levels of objects that participate in transforming user operations into operating system requests on target programs. They are illustrated in Figure 2.

- *User interface* objects handle widget callbacks and transform them into requests on distribution manager objects.
- *Distribution manager* objects represent a collection of distributed processes and groups and effectively take requests and “scatter” them to members.
- *Remote `Ds` server* objects are in the same address space as the distribution manager and represent `Ds` objects that actually exist on another machine. They take requests in a hierarchical form, “flatten” them to text and put them out on the network.
- *Remote `Ds` client* objects are in the remote debugger address space and take requests off of the network and “unflatten” them back to a hierarchical form, transforming them into normal `Ds` requests.
- *Vendor `Ds`* objects transform requests into operating system calls. These are the objects of Section 3.

For example, we can use Figure 2 to follow how a request in the user interface gets propagated to the target

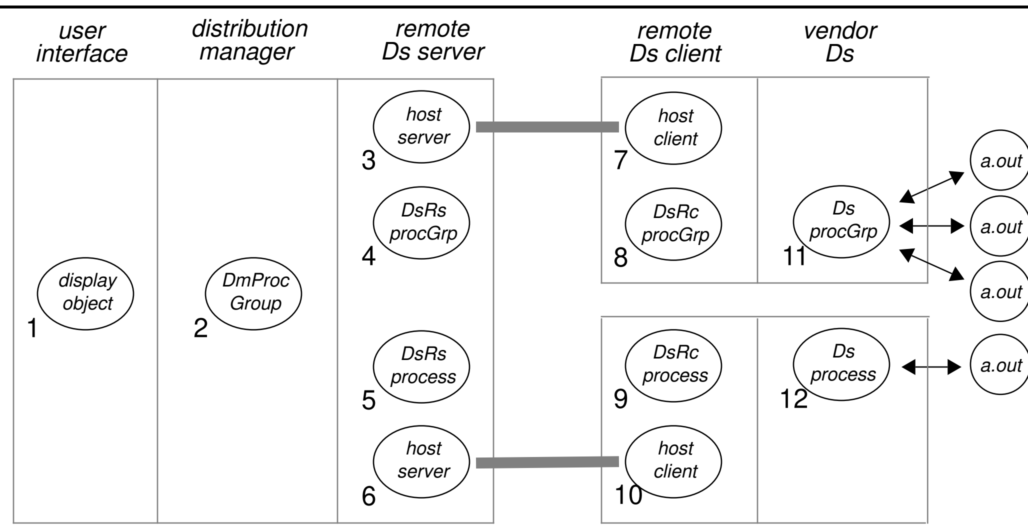


FIGURE 2. Interaction of Distribution Objects

processes being debugged. Suppose that the user has pressed the *Continue* button in the user interface. The display object (labelled "1" in Figure 2) performs a `cont()` operation on its current process group focus variable, the distribution manager process group object (labelled "2"). Object 2, in turn, dispatches the request to its member processes and process groups. In the case of our example, object 2 has two members: a remote `Ds` process group (labelled "4") and a remote `Ds` process (labelled "5"). When the process group 4 gets the continue operation, it "flattens" the request into text form and forwards it to its *host server* object (labelled "3") for transmission to the target machine. Object 3 packetizes the request and sends it out on the network to the *host client* object 7, which unpacketizes the request and forwards it to the remote `Ds` client process (labelled "8"). At this point the flattened request gets the hierarchical structure restored and turned into a request on the `Ds` process group object 11. Remote `Ds` process 5 performs the same flattening action as object 4, potentially using a different host server (in this case 6). From there, the continue request follows an analogous path through objects 10, 9, and 12 before getting to the target process.

The same mechanism can be used for registering callbacks. In that case, each object along the request path transforms the callback registration by recording the passed-in callback function and argument and then substituting one of its own functions and arguments to the next level down the chain. When the monitored event occurs in the target, the chain of callbacks is made back up to the distribution manager level. At that point, the distribution manager aggregates the callbacks it has received and makes the callback to the user interface code at the appropriate time. For example, if the UI wants to be notified when all target processes have reached a breakpoint, the distribution manager would collect the individual callbacks from the remote server and make the UI callback when all of them had been received.

This distribution mechanism also works for arrays distributed by the HPF preprocessor. When the UI requests values from such an array, the distribution manager can map the requests and forward them to the appropriate remote array objects. When the values come back, the distribution manager can assemble them into an appropriate form for the user interface.

7.2 The startup problem

One design goal of *p2d2* is to be able to attach to running computations in a transparent fashion. That is, the user should see no difference in functionality between debugging a collection of processes launched under debugger control and one that was attached by the debugger after starting. In an environment where an application process can be dynamically spawned on a remote machine, a debugger server must be available on the same machine and able to attach to the new process. This is accomplished by the communication library protocol and the distribution manager.

Figure 3 shows how the debugger can control processes initiated as a result of a *Run* request. In this example, the user has requested that the main program be run locally (on machine A). In response, the request is propagated down to the `DsProcess` object (9) using the technique described in Figure 2. Object 9 then forks and execs, creating the root of the user's computation—process 12. Process 12 then spawns process 13 on machine B, which spawns process 14 on machine C. In the rest of this example, we explain how the debugger takes control over the processes that it did not create directly.

After process 12 spawns process 13, it calls the debugger library routine

```
MP_RegisterSpawn(machineName, pid);
```

in its own address space, passing in the target machine name (B) and the process ID of process 13 on that machine. The debugging library then records those pieces of information in a table in process 12's address space.

Upon return from the function call, process 12 traps and `DsProcess` 9 is notified. (The trap was installed by the `DmExecution` object 2 before the *Run* command was issued.) The trap is propagated back up to object 2, which initiates a request to read the spawn information recorded in the table. After the information has been retrieved, object 2 creates a `DsRsProcess` (object 4), which determines whether it needs to create a new host server/client pair in order to communicate with process 13. In this case, there is no existing host server/client pair to machine B, so one is created. The first request transmitted on this new communication link is to create a `DsRcProcess` (object 7). It, in turn, creates the `DsProcess` 10, which is attached to process 13. The same procedure repeats when process 13 spawns process 14.

For this mechanism to work, the newly spawned process must stop execution after being spawned in order to permit its attachment by the `DsProcess` object. This will permit the `DsProcess` to insert any necessary event traps, such as one at the return of `MP_RegisterSpawn`(). The new process can be temporarily stopped by insertion of special code, either in *`crt0`* or in a "*`.init`*" section.

In the event that the processes are not being launched from a debugging session, the only change that is necessary is to avoid the temporary halting at startup. In this case, there will be no event traps installed and execution will proceed normally. Note that the process creation information will still be collected in a table in the address space of the spawning process.

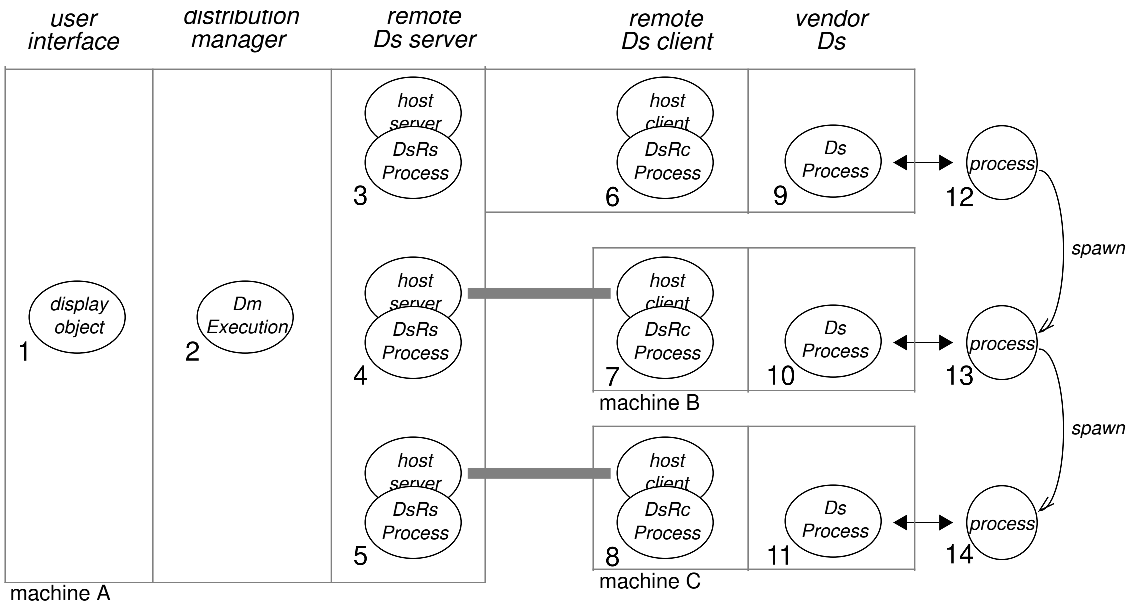


FIGURE 3. The Starting Problem

In order to be able to toggle the startup halting, the special purpose *`crt0`* or *`.init`* section first checks an environment variable `MP_DEBUG` to see if it is supposed to stop. The value of this environment variable must be propagated across the network as part of process creation.

8 Implementation status

The implementation of the complete architecture shown in Figure 1 would require scores of man-years of work. Most of that effort is in the construction of debugger servers for each of the target platforms. The most efficient way to complete that part of the task is to rely on vendors to modify existing debugging systems so that they provide the services required by `Ds`. Indeed, the design process of `Ds` took pains to consider how various vendors would be able to modify existing code to build a server.

NAS's main role in the project thus far has been to formulate the client-server interaction protocols and to implement a prototype for validation purposes. We currently have this prototype working for much of the `Ds` protocol. The implementation includes breakpoints, single stepping, frame exit traps, stack traces, and a limited form of expression evaluation. The code runs on a Sun Sparcstation under SunOS 4.1.

The first production version of *p2d2* will be a distributed debugger for PVM programs. It will include the prototype user interface, together with a distribution manager and a remote `Ds` package. In addition, NAS is modifying PVM to support the distribution protocol outlined in Section 4. As of this writing, we have made modifications to PVM and *`crt0`* to implement the startup scheme described in Section 7.2. Additionally, we have designed the distribution manager and are beginning a prototype implementation. The `Ds` server to be used in this version of *p2d2* will be based on *gdb*.

At the time of the conference, we intend to be able to demonstrate a working prototype of a distributed debugger for PVM programs running on a network of SGI workstations. It will use the *gdb* server and, support for remote debugging will be derived from *gdb* code rather than a remote object package.

After the initial production version is completed, we plan to expand the work along several fronts. First, we will extend a `Ds` server to include abstractions for multithreaded execution. We will also develop versions of the debugger for MPI programs and HPF programs.

9 Conclusions

In this paper, we described the design and implementation of a portable debugger for parallel and distributed programs. The inherent dependence of debuggers on machine architectures, compiler-generated symbol tables, and on interaction with operating systems makes porting debuggers very costly. Our design incorporates a client-server model in order to isolate non-portable debugger code from the user interface. The precise definition of a protocol for client-server interaction permits a high degree of portability of the client user interface.

Using this approach, a vendor can focus on developing a server that provides the data necessary for debugging, whereas a user site can develop an interface tuned to its particular application domain. A debugger for distributed computations can be realized by locating servers on the machines where target processes are executing. Aggregation objects in the distribution manager can perform scatter/gather operations across the collection of processes.

Portability across message passing implementations is achieved with a protocol that dictates the interaction between a message-passing library and the debugger. This permits the same debugger to be used both on PVM and MPI programs.

The process abstractions used for debugging message-passing programs can be easily adapted to debug HPF programs at the source level. This allows the debugger to present information hidden in tool-generated code in a meaningful manner.

Acknowledgments

During the course of the *p2d2* project we have received assistance from many people. In the preliminary stages of designing the `Ds` protocol, we studied existing debugging systems and had the cooperation of several companies. In particular, we are grateful to Don Breazeal, Eric Richards, Ray Anderson, and Karla Callaghan of Intel; Ann Mei Chang of SGI; Stuart Grossman, David Henkel-Wallace, Tom Lord, and Jim Kingdon of Cygnus Support; Mike Lloyd, Ken Hansen, and Frank Wolverton of MasPar; Rich Title of Thinking Machines; Gary Brooks of Convex; Steve Klineman and Charlie Fineman of Sun Microsystems; and Steve Zimmerman and Charlie Bryant of Kendall Square Research.

We also wish to acknowledge the Supercomputing '93 birds-of-a-feather participants who discussed the first version of the `Ds` protocol. In addition to the people named above, the participants included Jeff Brown of Los Alamos National Laboratory, John May of University of California at San Diego, Matt Halls of Verdix, Helene Kulsrud of CCC-P/10A, Hubertus Franke of IBM Research, Jim Lumpp of the University of Kentucky, Seth Hawthorn of Tandem, Suresh Damodaran-Kamal of the University of SW Louisiana, and Dennis Cottel of NRaD.

Finally, we wish to acknowledge the help we have received from colleagues at NAS. The management, especially Bruce Blaylock and Tom Woodrow, have provided encouragement and support. Dave DiNucci and Lou Lopez of the NAS Parallel Tools Group have provided us with insights about distributed computations and user interfaces. Our preliminary user group of Rod Fatoohi, Merritt Smith, and Rob Van der Wijngaart has given us useful feedback about the prototype user interface. Al Globus, Sandy Johan, Bill Nitzberg, and John West provided helpful insights about *p2d2* in a design review. Pam Walatka and Lou Lopez made good suggestions for improving the presentation.

Bibliography

[1] Doreen Cheng and Robert Hood. Protocols for Parallel and Distributed Debugging. A forthcoming NAS RND technical report. Readers with access to the WorldWideWeb can monitor the following URL for its appearance:
`http://www.nas.nasa.gov/Projects-Groups/Parallel.tools/Projects/P2D2/0.html`.

[2] Ming Hao. HP's Distributed Debugger. NAS New Technology Seminar, March 8, 1994.

[3] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software 8*, 5 (Sept. 1991) 29-39.

[4] Robert Hood and Doreen Cheng. Accommodating Heterogeneity in a Debugger for Distributed Computations. Submitted to the 28th Hawaii International Conference on System Sciences. Readers with access to the WorldWideWeb can access the submitted version from URL
`http://www.nas.nasa.gov/Projects-Groups/Parallel.tools/Projects/P2D2/0.html`.

[5] Intel. *iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual*. Intel Corporation, Beaverton, OR, April 1991.

[6] Charles Koelbel, David Loveman, Robert Schreiber, Guy Steele, Mary Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series, The MIT Press, Cambridge MASS. 1993.

[7] John May and Francine Berman. Designing a Parallel Debugger for Portability. *Proceedings of IPPS '94*, Cancun, Mexico, April 1994.

[8] John Michael May. *An Extensible, Retargetable Debugger for Parallel Programs*. Ph.D. dissertation, Department of Computer Science and Engineering Technical Report CS94-375, University of California, San Diego.

[9] Paul Maybee. NeD: The Network Extensible Debugger. *Proceedings of Summer '92 USENIX,* San Antonio, 1992.

[10] MPI committee. Message Passing Interface (draft). Jan. 1994.

[11] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates, Sebastopol, CA, 1990.

[12] C. Pancake. Panel presentation at *the Debugging Workshop for High Performance Computers, Supercomputing'93*, Portland OR, Nov. 1993.

[13] Vaidy Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency*: *Practice and Experience* 2(4):315-339, 1990.

[14] Thinking Machines. *Prism User's Guide*. Thinking Machines Corporation, Cambridge, MA, Dec. 1991.

[15] J. C. Yan. Performance Tuning with AIMS—An Automated Instrumentation and Monitoring System for Multicomputers. *Proceedings of the 27th Hawaii International Conference on Systems Sciences*, ACM, Jan. 1994.