5-1-2013

The Distributed Application Debugger

Michael Quinn Jones
University of Nevada, Las Vegas, mjones112000@gmail.com

THE DISTRIBUTED APPLICATION

DEBUGGER

by

Michael Q. Jones

Bachelor of Science (B.Sc.)
University of Wisconsin-Madison
2003

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

School of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2013

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Michael Q. Jones

entitled

The Distributed Application Debugger

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Jan Pedersen, Ph.D., Committee Chair

Angelo Yfantis, Ph.D., Committee Member

Juyeon Jo, Ph.D., Committee Member

Aly Said, Ph.D., Graduate College Representative

Thomas Piechota, Ph.D., Interim Vice President for Research &
Dean of the Graduate College

May 2013

Abstract

Developing parallel programs which run on distributed computer clusters introduces additional challenges to those present in traditional sequential programs. Debugging parallel programs requires not only inspecting the sequential code executing on each node but also tracking the flow of messages being passed between them in order to infer where the source of a bug actually lies.

This thesis focuses on a debugging too called The Distributed Application Debugger which targets a popular distributed C programming library called MPI (Message Passing Interface). The tool is composed of multiple components which run together seamlessly to provide its users an effective way to remotely launch, replay, and analyze parallel programs both while they are running and after they complete.

Acknowledgements

I would like to thank Dr. Pedersen for pushing me to strive for my best throughout this thesis process. I appreciate him meeting with me personally in the evenings and on the weekends when it was most convenient for me during the school year, and for quickly responding to email over breaks and holidays. When I first asked him to be my advisor, I asked that he push me because I wanted to be a better programmer. I told him that I expected a lot out of myself and that I wanted him to hold me accountable to that, because I wanted to look back and be proud of the work that I had done. He never wavered or compromised the integrity of the work that I was putting together, and I look back on what we have accomplished together with great pride. Thank you.

I want to thank my older siblings Ann, Bill, and Dan for always including me in their lives despite our age differences, and for quietly giving me role models that so many people wander through life without. I want to thank my parents for always taking such a genuine interest in the lives and happiness of their children, for me encouraging along the way well after I left home, and for teaching me what it means to truly <u>earn</u> the things that I want in life. These are lessons that I always appreciate in my life and will pass along to my children as well. May finishing graduate school be another "alligator" that we tracked down.

I want to thank my daughter Erin, for being patient with me during her first year of life. I have not been an active enough father in her life yet and I promise that I will make up for that over the years to come. Most importantly, however, I want to thank my wife Abby, who showed me endless patience, support, and love when I put our lives on hold in order to go back to school. Without your unselfish support and encouragement I would have never finished, and your name deserves to be on the front of this thesis as much as mine. I love you.

<div align="right">

MICHAEL Q. JONES

</div>

University of Nevada, Las Vegas

May 2013

iv

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Anyone who has developed software before will likely agree that bugs will be encountered along the way. No matter how careful you are, logic errors, memory mismanagement, race conditions, inaccurate execution path assumptions and a whole host of other issues will be encountered from time to time. These issues are acceptable, understandable and expected when designing software. As a result of the popularity of sequential programs, and the common understanding that debugging will always be part of their development, many exceptional debugging resources have been created to help the developer analyze and step through sequential code. These tools are flexible in order to let the user pause, rewind, inspect, and compare the execution of the lines of their programs.

Developing parallel programs which run on distributed computer clusters introduces additional challenges to those present in traditional sequential programs. When debugging parallel programs, one needs to be able to inspect both the sequential code executing on each node and track the flow of messages being passed back and forth between them in order to infer where the problem actually lies. One such distributed programming language is called MPI [Don94]. It stands for Message Passing Interface and is a C library which utilizes the power of distributing work across processors while staying in sync and reporting progress by passing messages.

Because MPI code is just sequential code being run on a cluster of computers, it inherits all of the same common debugging errors present in sequential programs. In addition to these familiar 'sequential' bugs, is another set of completely different 'parallel' bugs which impede the engineers even more. The *Cause and Effect Chasm* [Eis97], for instance, is the notion that where a bug becomes noticeable may not be near where the problem actually was introduced. It could be introduced due to any number of mistakes including initializing the value of a counter incorrectly earlier in a method, not allocating enough memory in a constructor, or overwriting a variable incorrectly a thousand lines earlier. Regardless of what the root of the problem turns out to be, when you stumble across its

symptoms, you will inevitably begin retracing to find out where it originated. This is difficult enough in sequential programs running on the same processor, but what if the bug actually initialized much earlier due to the behavior of code running on a completely different machine? What if an error was introduced during the messaging phase because of a wrong address, or data type, or count? Trying to track down the root of a problem across computers can make bug finding even harder. Another major difficulty comes in just organizing tracing information printed to the screen. Because print statements are often the first form of tracing information, the user either has to inspect the screen of all of the computers running on the cluster, or sift through all of the data printed to a common monitor should the processes all be running on the same machine. Regardless, although printing trace data to the screen is a very popular and useful technique in the world of sequential programming, it becomes an unorganized, overwhelming *Information Overload* [PJ12] problem quite quickly in the distributed world. Perhaps the most prominent difference between debugging sequential code and distributed code is the inability to just halt execution of the program when attaching a debugger. Attaching a debugger, such as *GDB* [GDB13], for example, halts the execution of a process immediately which gives the developer the chance to inspect all the variables of the system as it steps through the execution of the program one line at a time. This is very helpful because one line may have a side effect on another one that the developer may never had thought of initially. In a distributed system, however, attaching a debugger to a process does not halt the execution of the program because even though one process stops, the rest of the processes continue on.

These are just some of the problems that I wanted to address with the Distributed Application Debugger. Among other things, I wanted to give the user a centralized and organized space to examine the output of each node while being able to also inspect and match the messages being passed between them. I wanted to provide them with a way to not only replay a session but also to halt a session in order to step through each node's execution while inspecting all the variables of a system. Finally, I recognized that clusters of computers are often housed a universities or super computer centers around the world. This led us to insisting that the application be able to run remotely and behind any number of impeding computer servers that needed to be logged into first before being able to access the actual computer cluster.

Chapter 2

Background and Related Work

The Distributed Application Debugger is a system of applications created because the need for a debugging tool for MPI [Don94] is needed. This chapter focuses on the reasons why tools for debugging parallel programs are necessary, as well as the the motivating factors behind the features that were included in this one. It introduces the MPI programming language and describes the portions of it which became the focus of the Distributed Application Debugger. I also talk about two other commercially available debugging tools used within the community, and what they have in common with mine and also what is different. Finally I discuss the software development procedure that was used in order to ensure that the Distributed Application Debugger would successfully deliver its key features

2.1 Debugging Parallel Programs

Research by Cherri Pancake discussed in a keynote address on parallel computing systems [Pan93] suggests that tools for parallel programming and debugging are often found to be unhelpful for users because their developers do not spend enough time trying to understand what the root problems that their users really need addressed are. In an effort to better understand what problems users debugging parallel programs encountered most often, and what problems took the most time debugging, a survey was given by Dr. Jan Pedersen to two different sets of graduate students at the University of Nevada, Las Vegas over the course of two years. The results of the first survey, presented in [Ped06], and the second survey, presented in [PJ12], agreed with each other and collectively served as the motivation for the development of the Distributed Application Debugger.

2.1.1 The Survey

The survey based much of its foundation around the model for constructing parallel programs known as PCAM [Fos95]. This four part model breaks parallel programs into two areas dealing with correctness, *Partitioning* and *Communication*, as well as two areas dealing with performance, *Agglomeration*, and *Mapping*. Although the performance categories are important when measuring the quality of a parallel system, the survey focused only on the two categories characterized by correctness because it was decided that a program should first be modeled correctly before being optimized.

The first of the correctness categories, *Partitioning*, deals with the task of partitioning both the data and functionality of the algorithm being implemented. It can be further sub-categorized as *Data Decomposition*, which covers developing code structured to deal with managing memory, modeling data structures etc., and *Functional Decomposition* which deals with the organizational side of defining the responsibilities of each node and establishing roles for architectures such as pipelining and master/slave relationships. The second of the correctness categories, *Communication* is the task of implementing interprocess communication. It can also be broken down into two sub-categories: *Message Errors* which deals with correctly addressing send and receive calls between the appropriate destination and source nodes, and *Protocol Specifications* which deals with asynchronous message calling and buffering which lead to unexpected results and side effects. These 4 subcategories from [Fos95], along with a fifth classification, *Sequential*, which deals with bugs that you would find in any sequentially running program such as mistakes with conditionals, method calls, race conditions, pre and post conditions, and algorithm modeling to name a few, make up the 5 categories that students were asked to classify their bugs into.

2.1.2 Survey Results

The students were asked to keep records of each time they spent time debugging their parallel code and log which category their bug best fell into. The graph shown in Figure 2.1 and data recorded in Table 2.1 show the results of the survey as they completed each of seven projects throughout the semester and categorized their bug types in the process.

Complimenting the bug categorization was a questionnaire based on Eisenstadt's research [Eis97] which stated that sequential errors are categorized by 3-dimensions. The 3 dimensions are:

- Dimension 1: Why is the error difficult to find?

- Dimension 2: How is the error found?
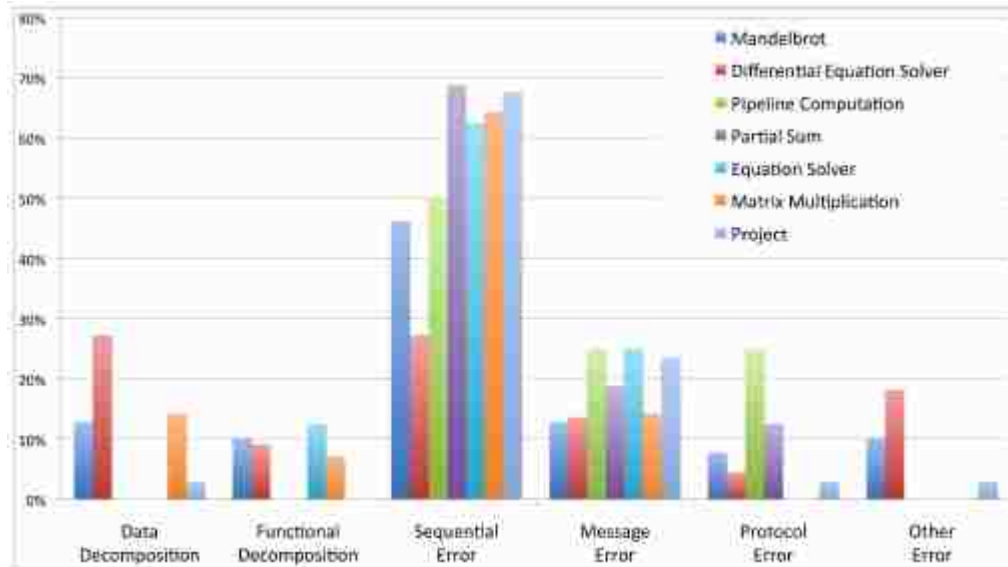
Figure 2.1: Error classifications of a class of graduate students taking a parallel programming class.

	Data Decomp.	Functional Decomp.	Sequential Error	Message Error	Protocol Error	Other
Average Time	19.9	68.1	24.3	61.4	50.0	30.6
Total Time Spent	278	545	1,846	1,536	451	545
# Errors	14	8	76	25	9	8
Total time Spent in %	5.67%	11.12%	37.67%	31.34%	9.20%	5.0%

Table 2.1: Debugging time for the bugs reported in the survey.

- Dimension 3: What is the root cause of the error?

Based on Eisenstadt's 3 dimensional model of sequential errors, the class indicated how their bug was classified within each of the 3 dimensions. In Dimension 1, the *Why*, the popular answers were the Cause and Effect Chasm, mentioned in the introduction, and Inapplicable Tools. In Dimension 2, the *How*, the overwhelmingly most popular answer was by using Print Statements. Finally for Dimension 3, the *What*, the popular answers were pointers, index references outside of an array, and faulty design logic.

2.1.3   Survey Conclusion

The survey, inspired by Pancake's observation that parallel debugging tools were being developed without real understanding of user problems, indicated several results which became the foundation for the development of the Distributed Application Debugger. First, based on the results indicated in Figure 2.1, the overwhelming majority of the bugs where categorized as 'Sequential Type' bugs.

Secondly, based on the amount of time spent on each type of bug, it is clear that 'Message Errors' are very costly errors, accounting for nearly as much time spent as sequential bugs despite having just a fraction of the number of bugs reported. Thirdly, based on the 3 dimensional questionnaire based on Eisenstadt's research, it is clear that print statements are still a very popular debugging technique, which agrees with Pancake's results [cMP94] which found that up to 90% of all sequential debugging is done using print statements.

After analyzing these results, several conclusions were made about features that would be needed for the parallel debugging tool. First, based on the number of sequential bugs reported by the students, it would be helpful if I could present a view of the system broken down as a series of sequential programs running rather than as one large distributed program. Any data reported, such as the order of commands being executed, would have to carefully be recorded in the exact order that it was executed on each node and data such as line numbers would need to be included. Also, care would have to be given to give a view of the code executed sequentially per node, but without overwhelming the user with data extracted from nodes that are known to not be the root of their bug. Secondly, based on the amount of time spent on *Message Error* types of errors, it was determined that the tool would have to help match up send and receive commands. In addition to matching up send and receive commands, it would be valuable if the tool could indicate 'unmatched' messages immediately. This would alert the user that there might be a message addressing error early and allow them to fix the problem proactively which would lessen the *Cause and Effect Chasm* distance between when the bug was introduced and when it was noticed. Thirdly, it is was clear that students leaned on 'print' statements when debugging their code. Although this is generally considered a poor technique, its popularity made it clear that a separate console for each node would have to be displayed on the front end to help the users analyze the print statements of each node. At the same time, however, I felt an obligation to encourage them to use more advanced techniques to replace the use of print statements. It was decided to pursue integrating GDB into the application and allowing the users to conveniently attach it to whatever nodes they wanted. This would allow us to leverage the power of an already mature debugging tool, and show the students that there were more sophisticated alternatives to dividing and inspecting their code with print statements.

In addition to the features derived from the research, it was decided that there were others that would make the application a truly helpful tool. Importing The Runtime of the debugging framework, for instance, should be transparent to the user and work without the user having to annotate special sections of their code as 'debuggable' etc.. Also, the application should include options to record and replay sessions that produced especially peculiar results. Finally, the user should be able to use the application from home. This last feature introduced a tremendous amount of extra work to

manage the logistics of logging into a university cluster, but I felt that it was worth it. If students had to be present at the computer lab in order to gather the debugging data, they likely would not bother using the tool since traveling to the university would take as much time as debugging with rudimentary techniques while logged in from home.

2.2 MPI

The Distributed Application Debugger is a tool meant to investigate Message Passing Interface code written in C. Before moving onto the implementation of the debugging tool, I would like to touch on the framework itself and indicate what parts of it are supported by the tool.

2.2.1 Framework

The MPI framework was developed in order to give a standard implementation of a library that supports message passing between computer processors running designated sections of code of a program in parallel. Its goals include high performance, scalability, and portability.

General Structure

The general structure of an MPI program is to first issue an *MPI_Init()* command and to finish with an *MPI_Finalize()* command. The MPI program, itself, run between these two calls. After initializing, the node will generally determine what rank it has been assigned within the system by calling *MPI_Comm_rank()*. Also, it is common to request the total number of nodes within the system by issuing an *MPI_Comm_size()* command. This allows the system to know the range of nodes available for messaging and begin to split up the work between the available nodes.

Messages, Buffering and Blocking

At the core of MPI is the actual passing of messages between nodes using *send* and *receive* commands. All send commands have a *destination* parameter which indicates the recepient node that the message should be sent to along with a *tag* (integer value) that can act as an identifier when the receiver decodes the message. The send message can either be in the form of a blocking message, *MPI_Send()*, or in the form of a nonblocking message, *MPI_Isend()*. The blocking and nonblocking aspects of the send commands is with regard to whether the user can be ensured that it is safe to edit their application buffer. The blocking *MPI_Send()* is guaranteed to block until the data is either delivered to a receiver or safely copied into a system buffer. The nonblocking *MPI_Isend()* command, however, returns immediately to the caller without ensuring that the data has been copied out of the

7

application buffer, making reusing it unsafe. Like the send commands, the receive commands also come in both a blocking and a nonblocking form. The blocking version, *MPI_Recv()*, halts progress of the process entirely until a message has been received. The nonblocking version, *MPI_Irecv()*, returns immediately as expected and gives the user the option to poll a memory address known as the 'request' to find out if the corresponding receive buffer has safely completed receiving a message. Like the send command, the receive commands have an address parameter, known as *source*, and a tag used to filter multiple messages coming from the same source, but unlike the send commands, these values are optional. When the receiver wishes to forgo filtering on a specific source and/or specific tag value, it can be specified as a wild card value: *MPI_ANY_SOURCE* and *MPI_ANY_TAG* for source and tag respectively. Because these values are optional, receive commands include an extra parameter of type *MPI_Status* which can be referenced to find the actual source and tag values once the receive has been completed.

2.2.2   Supported Commands

The Distributed Application Debugger does not support all of the commands found within the MPI library [ANLD13], but does support 12 core commands making analyzing the initializing, message passing, synchronizing and finalizing of a typical MPI program possible. Table 2.2 displays the scope of the commands available for debugging within the Distributed Application Debugger along with their signatures.

2.3   Other tools

The Open MPI Project's frequently asked questions website [Pro13] characterizes how to debug applications running in parallel as a difficult question to answer. In their words

> "Debugging in serial can be tricky: errors, uninitialized variables, stack smashing, ... etc. Debugging in parallel adds multiple different dimensions to this problem: a greater propensity for race conditions, asynchronous events, and the general difficulty of trying to understand N processes simultaneously executing – the problem becomes quite formidable."

The project recommends two enterprise level debuggers, *DDT* (short for the Distributed Debugging Tool) by Allinea Software [Sof13a] and *TotalView* by Rogue Wave Software [Sof13b], to aid in the complicated task of debugging MPI programs. Because of this endorsement, I felt that it was important to touch on some of their features and how they may be a better tool of choice at times from the Distributed Application Debugger.

Command	Description
MPI_Init(int *argc, char ***argv)	Initializes the MPI environment
MPI_Comm_rank(MPI_Comm comm, int *rank)	Determines the rank of the process in the cluster
MPI_Comm_size(MPI_Comm comm, int *size)	Determines the size of the cluster
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Performs a blocking send
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	Blocking receive for a message
MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)	Begins a nonblocking send
MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)	Begins a nonblocking receive
MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)	Blocking test for a message
MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)	Nonblocking test for a message
MPI_Wait(MPI_Request *request, MPI_Status *status)	Waits for an MPI request to complete
MPI_Barrier(MPI_Comm comm)	Blocks until all processes in the communicator have reached this routine.
int MPI_Finalize(void)	Terminates MPI execution environment.

Table 2.2: The MPI commands supported by the Distributed Application Debugger.

It is remarkable how many features that DDT and TotalView have in common. Both GUIs are very *debugger-centric* in the way that the viewer is always focused on source code and the node specific view is just which line of it the node is located on. Because of this, other features, such as a graphical views of the messages being sent, are done with popup windows. The Distributed Application Debugger took great lengths to not have views presented in popups so that the user is not tasked with juggling them. The Distributed Application Debugger by contrast, is more of an analysis tool which integrates a debugging feature when needed. Because DDT and TotalView are so debugger centric, the MPI program being debugged must be compiled with debugging symbols. This is not the case with the Distributed Application Debugger because this is only a requirement when actually attaching GDB.

Both *DDT* and TotalView offer great features that allow inspection of the code at both the process level and the thread level. They display their message stacks in graphical form, whereas the Distributed Application Debugger displays its messages in tabular form. Also, both not only are able to record and replay MPI sessions, but also allow the user to 'rewind' a session, whereas the Distributed Application Debugger always goes forward. Most impressively, they both can scale to

over 100,000 processes which the Distributed Application Debugger would not be able to keep up with. This scalability comes with a large price, however, with DDT costing $696.00 for an academic or government workstation license, and TotalView costing over $1,000.00.

The Distributed Application Debugger's strengths lie in its simplicity. It can be configured to run remotely, but does not require any applications to be running on the remote machines prior to the session beginning. It copies, compiles, launches, and cleans up any applications needed in the communication line without requiring them to be already running on the remote computers as both DDT and TotalView do. It also leverages the extremely popular and powerful GDB application which most students are already familiar with. This cuts down on its learning curve and further assists the students in focusing on their parallel programs as just a set of sequential programs running. The Distributed Application Debugger's layout presents a useful layout for viewing 2 or 3 nodes at a time because their values are displayed side by side without have to switch between them, while the other two most popular debuggers display one main code display and request that the user keeps switching between which node to report on.

Figure 2.2 and 2.3 displays DDT's main front end GUI display and Figure 2.4 and 2.5 displays the same for TotalView.
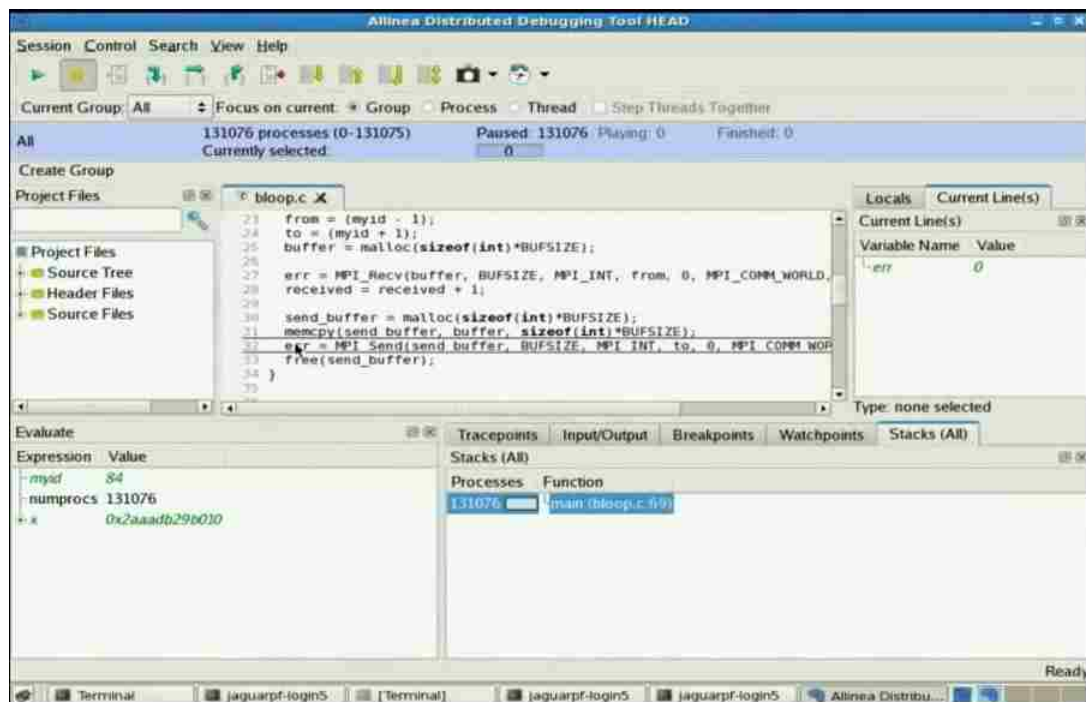
Figure 2.2: The GUI display of Allinea Software's DDT application.
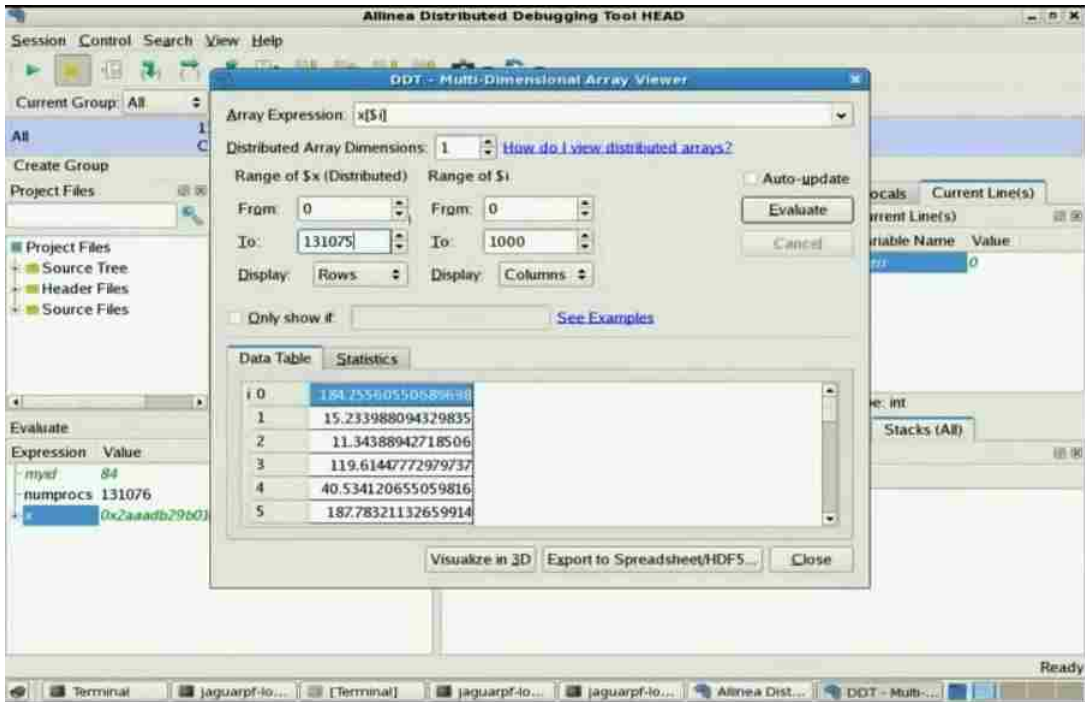
Figure 2.3: A buffer inspection popup within Allinea Software's DDT application.

2.4 Software Development and Risk Assessment

Among much evidence presented in [Gib94] identifying the alarming number of software projects which never produce a successful end product, is a particularly disturbing study performed by the *Software Engineering Institute* [SEI13], a U.S. Department of Defense research institution at Carnegie Mellon University. The study evaluated the abilities of 261 software organization to manage and create software that met its customer's needs on a 5 point scale, where 1 indicates chaos and 5 indicates the paragon of good management. The study found that about 75 percent were stuck at level 1 with no formal process, no measurements of what they do, and no way of knowing when they are on the wrong track or off the track together. The next 24 percent of projects were only at a level 2 or 3. Knowing that so many software projects tend to fail to deliver on their initial intention, and not wanting the Distributed Application Debugger to become one of them, we employed risk management [Boe91] to help us deliver a successful product plan.

The initial phase was spent making a list of core features that we were going to include in the application and, from those, extracting a *risk list* [Boe91] of features which had degrees of uncertainty, technical constraints, or unrealistic development scopes which could eventually make them undeliverable. The goal was to develop, or at least prototype, these features first in order to be sure that they were attainable. I took an iterative development approach, as outlined by Craig

11
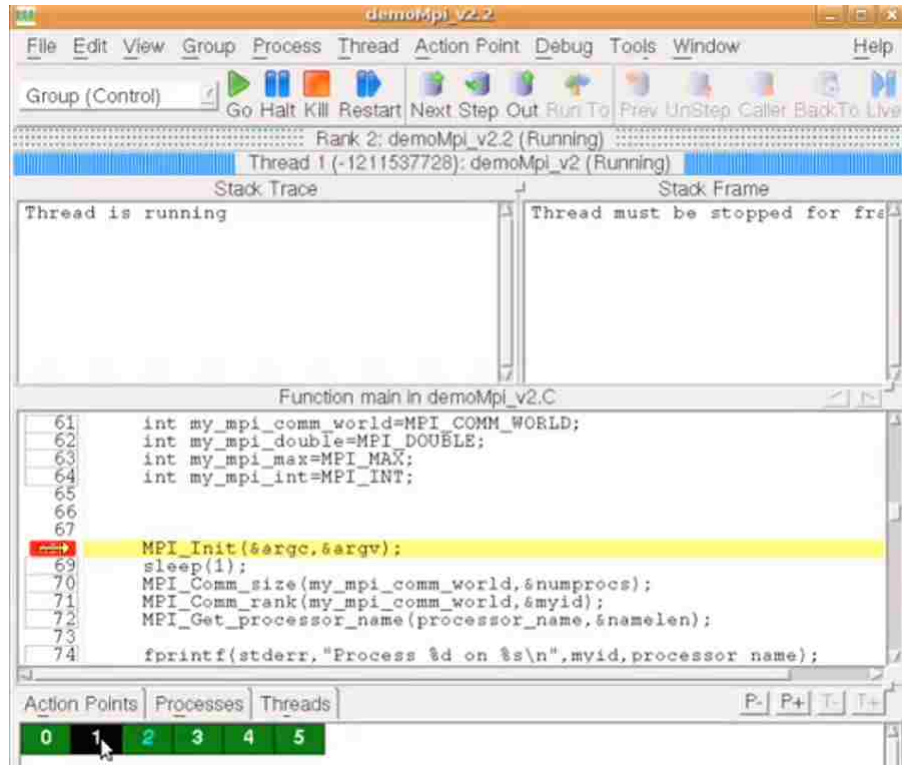
Figure 2.4: The GUI display of Rogue Wave Software's TotalView application.

Larman [Lar07], of writing small, but complete, aspects of the items on the risk list and then met regularly with Dr. Pedersen to test them and discuss their risk level. As Larman put it, "It is better to resolve and prove the risky and critical design decisions early rather than late — and iterative development proves the mechanism for this.". Below is the list of high risk features that I came up with.

The Risk List

1. Integrating GDB
2. Recording and Replaying
3. Running Remotely

Integrating GDB

I decided that integrating GDB into the application was absolutely essential for the product to be successful. I felt that students needed to have an alternative to using print statements that was still easy to invoke when needed. I also felt that if I was going to call the product a debugger, it needed to offer features like stepping into and over lines of code, variable inspection, and call stack retrieval at
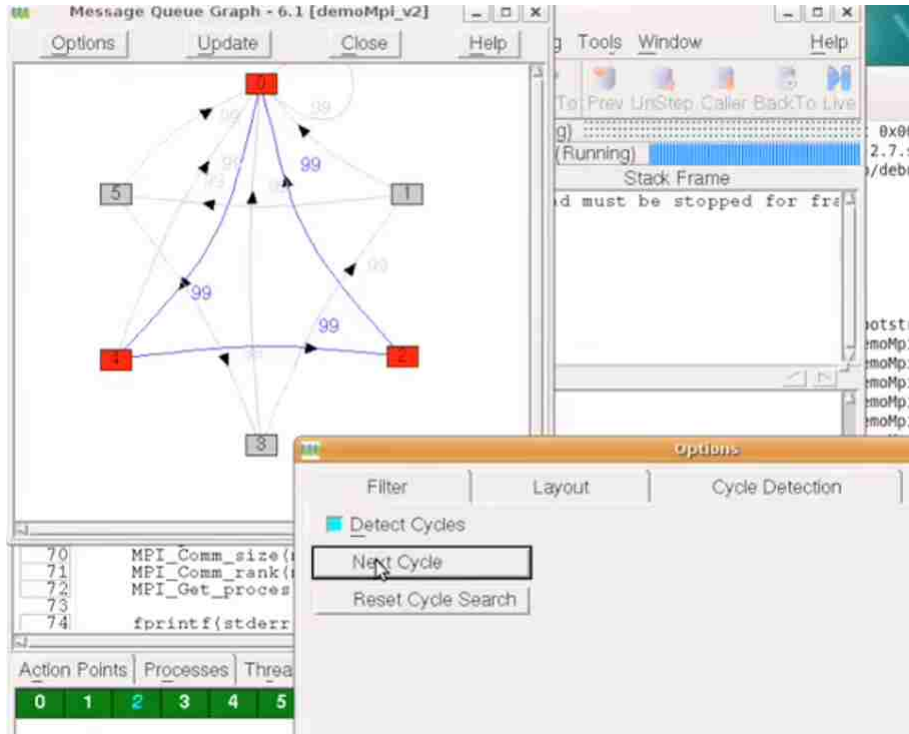
Figure 2.5: A message illustration popup within Rogue Wave Software's TotalView application.

the very least. Without these basic features, the tool would really just be a profiler used to analyze, rather than inspect. Leveraging GDB specifically was an attractive choice for several reasons. As indicated in [MS08], GDB is the most commonly used debugging tool among Unix programmers and the foundation for other front end GUI wrappers such as DDD [DDD13] and Eclipse [Ecl13]. GDB works by first attaching itself to a process and then interacting with the user by taking in instructions from stdin, and printing the results to stdout.

I felt this particular feature lent itself well to prototyping which, as detailed in [War09], offers several advantages, among which is a very low-investment-to-benefit ratio. The initial prototype for this feature was a program which could take in a command from the console to start, and then launch a child GDB process that would attach itself to a test program. The parent process then needed to take in commands from the console and pass them along to the GDB process's stdin and, likewise, read from GDB's stdout and pass those values back to the parent. After another couple of weeks of coding, we met again to talk about the risk of the feature. I had shown that I could duplicate the file descriptors of a child process forked from the prototype and setup up a threading model that could handle piping commands from the prototype's console window into GDB, and also piping the output of GDB back to the prototype. The initial prototype was promising but it lacked

one important feature. Since I was planning a remote debugger, was a prototype that only printed to the console really applicable yet? I went back and did another iteration of development, which extended the prototype to write and read from a TCP port passed in at its command line, rather than its console. I called the prototype the 'GDB Bridge', included in Appendix A.6, and assessed that integrating GDB into the application was possible.

Recording and Replaying

Research by Thomas J. Leblanc and John M. Mellor-Crummey [LMC87] illustrate that since parallel programs often include nodes passing messages asynchronously, the execution behavior of a parallel program in response to a fixed input is clearly indeterminate. Given this information we felt that it was crucial that the Distributed Application Debugger be able to record the execution of an MPI session and allow the user to both inspect it by hand and replay it through the code. This would give the user the ability to inspect unexpected occurrences that were not always recreatable without having to run the application over and over again hoping to go down the same execution path. I also felt that this feature was a high risk one that needed some development early to determine how it would be done. Dr. Pedersen had conceived the idea to redirect the calls to MPI to an external library [Ped03], which became the foundation for The Runtime component described in section 3.3. Given that The Runtime component would be notified each time a user made and MPI call, it was conceivable that we could record the parameters and return values of each call to an MPI method to file. I decided to represent the MPI sessions using the XML standard [XML13] because of its inherent ability to represent levels of scope, its great readability, and its wide adoption due to the endorsement by the World Wide Web Consortium.

I started writing an XML library, the first of several that I would write for this project, and we met regularly to evaluate its progress. I started out first by defining a structure which had XML fields such as *Name*, *Value*, *Attributes*, and *Children*, and then wrote functions to convert those values into XML compliant strings. After that, I wrote more functions to store the XML structures to file and to read them back into memory. Once my XML library was complete, we met to design an MPI XML schema which I could use to serialize each of the 12 MPI commands I was supporting. Once the schema structure was decided on, as illustrated in Appendix A, another iteration was started in which I wrote another library, called *MPI_XML*, which used the XML library created in the previous iteration to serialize each MPI command according to the agreed upon schema. After that, another iteration was made to read the commands back into memory so that a recorded MPI session could be replayed precisely how it was before. We met again and agreed that I now had a solution for recording and replaying an MPI session.

Running Remotely

The final feature from the risk list was the ability for the Distributed Application Debugger to run remotely from anywhere. I felt that if students had to commute to the university computer lab to use the debugging tool, that it would ultimately discourage them from using it. Given this need it became obvious that the application would actually need to be split into several applications, and that I needed to identify what they were before we could proceed. This began a two phase design phase in which we ultimately settled on the four component architecture described in Chapter 3.

In the first phase, I focused on sketching out a high level design of the system. I identified that there would obviously be a front end GUI running at the student's home for them to get their debugging information from. Also there would obviously be a runtime component running at the university which I had already developed record and replay features for. I also devised that there would need to be some Call Center component running between the two which would ultimately be in charge of communicating the front end's commands to The Runtime, along with a series of *Bridges* in case the cluster was not accessible directly. I then turned my attention to designing the user experience that the user would have when interfacing with the client. Unger and Chandler [UC09] suggest the practice of using 'wireframing', which is the idea of laying out the behavior of a front end GUI before you actually code it, as a way of bringing visual ideas to your project team quickly. I installed one implementation of it called *Balsamiq Mockups* [Bal13] and sketched out the front end user experience and a high level communications overview. We discussed several iterations of the wireframes, examples are shown in Figures 2.6 and 2.7, and ultimately agreed on an architecture allowing remote debugging before writing any code at all.

In the second phase, I worked on confirming that a front end written in C# running on Windows, could SSH into a series of computers running UNIX, securely copy folders of debugging files from node to node, and launch a prototyped Bridge and Call Center. I used the *Tracer Bullet* design methodology [HT99], which gets its name from the way that a sniper can confirm that the path of his bullet will hit his target by firing tracer bullets, which leave a phosphorus trail behind, first. The goal of this technique is not to write any actual code, but rather to first connect prototypes together to prove that a communication path will actually work. I found an open source library called SharpSSH [Sha13] which allowed me to initiate an SSH connection from a front end written in C# to a computer running an SSH Server. Once I was able to do that, I wrote C# applications that would temporarily stub in for The Bridge and The Call Center and practiced copying them between Linux computers at my own home. I installed Mono [Mon13], the .NET framework for UNIX, on several Linux machines so that I could actually launch the C# stub applications after SSHing in. After a few iterations I had worked out the logistics of SSHing, copying, launching and connecting,
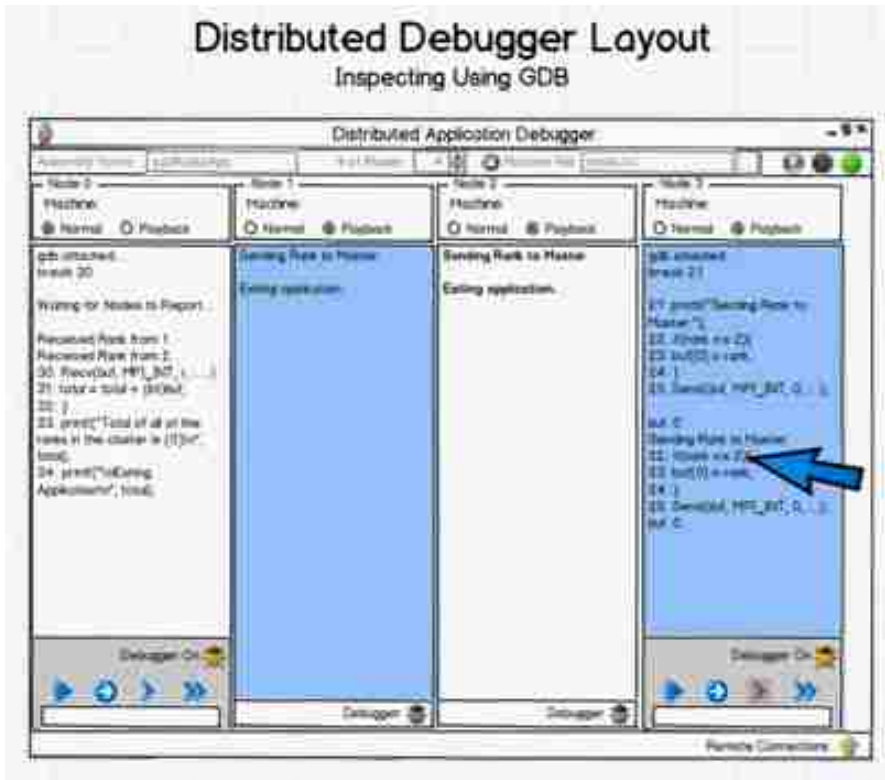
Figure 2.6: A wireframe of The Client component used during the design phase.
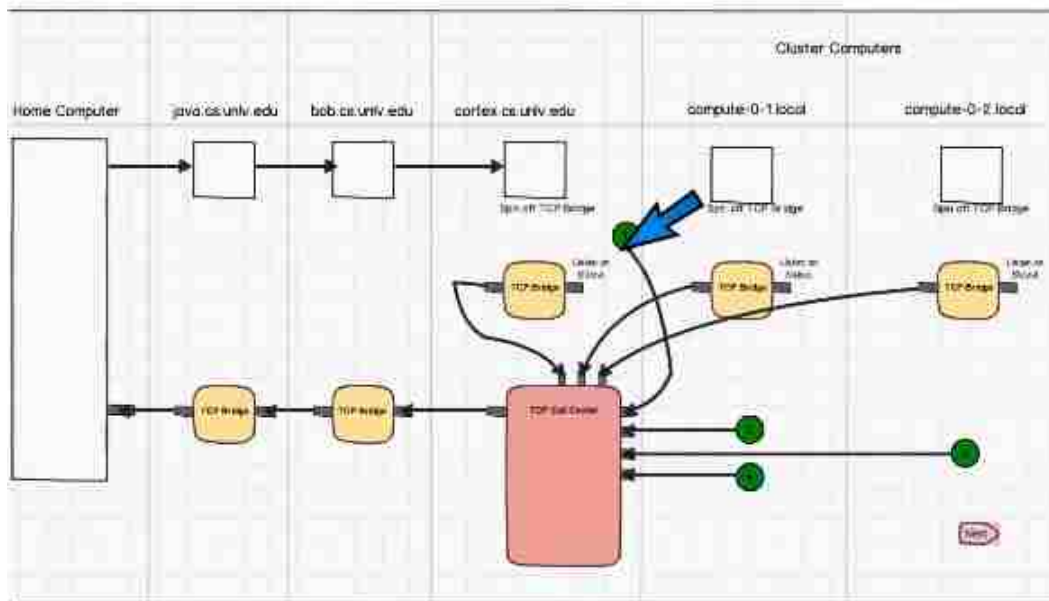
Figure 2.7: A wireframe of the communication architecture made during the design phase.

that gets described in Section 3.1.1, which ultimately gave the user a connection to The Runtime while connected remotely. After this I was confident that my high risk features could be met, and that I would ultimately deliver a working product.

Chapter 3

The Distributed Application

Debugger

The need for a tool for debugging MPI programs became apparent when the results of a survey of graduate students showed that most of their debugging was done using print statements. Although commercial debuggers are available that can monitor distributed processes at the petascale level, research by [BH04] found that 80 percent of developers used less then 4 processes when debugging their code. With this in mind I felt that a debugging tool focused on the common debugging needs of its target audience, rather than on extreme scalability, could still be very effective. This chapter focuses on the implementation details of the Distributed Application Debugger and its three major components: The Client, The Call Center, and The Runtime. It describes the messages passed between them during an MPI debugging session, and concludes with what happens when GDB is introduced.

3.1   The Client

The Client is the user facing portion of the application. It is meant to help the user connect, control, and analyze MPI code running on a remote cluster of computers. The layout is meant to be simple and is broken into 3 areas, the tool bar, the node panels, and the configurations as displayed in Figure 3.1. Initially the user must enter credentials to establish a remote connection to the MPI cluster and then can begin debugging sessions.
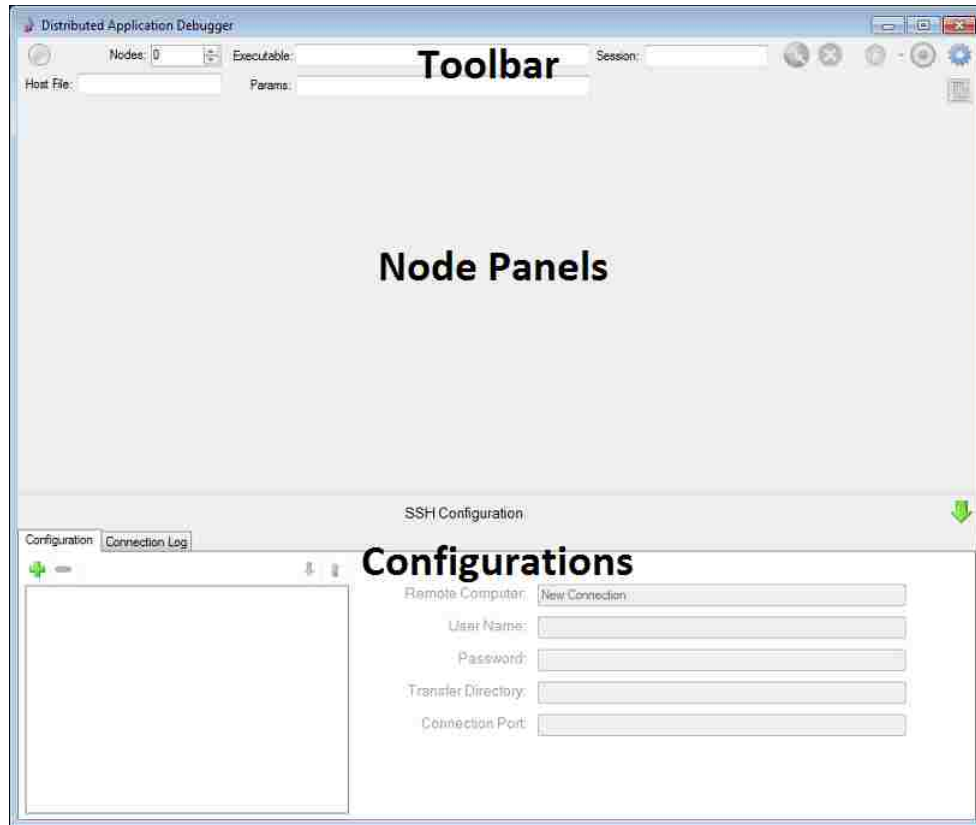
Figure 3.1: The Client without any fields populated.

3.1.1 Establishing a Remote Connection

In order to begin a debugging session, the user must first establish a connection into the MPI cluster. The user supplies his or her credentials in the *Configurations* area of The Client as displayed in Figure 3.2. It is not uncommon for a computer within an MPI cluster, however, to only be accessible from within its private network. In the case of some universities, for instance, students who wish to remotely access a computer within an MPI cluster, are asked to SSH into the campus student computer lab first. Once they establish a connection to a computer within the university network, they can then establish a second session to a computer within the cluster. The Distributed Application Debugger allows the users to input as many connections as they must make in order to finally reach the MPI cluster. When multiple addresses are entered, connections are made in sequential order, starting from the top of the list. Figure 3.3 shows the scenario where a user needed two connections in order to reach the computer cluster.

Once the user has provided the credentials needed to log into the MPI cluster, the Connection button will become enabled in the tool bar, and the user can establish the remote connection by
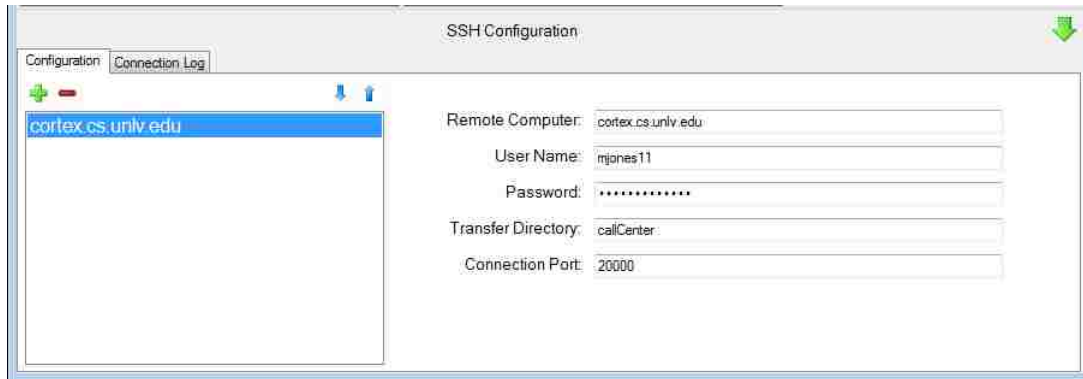
Figure 3.2: The Client configured for a direct connection to The Call Center.
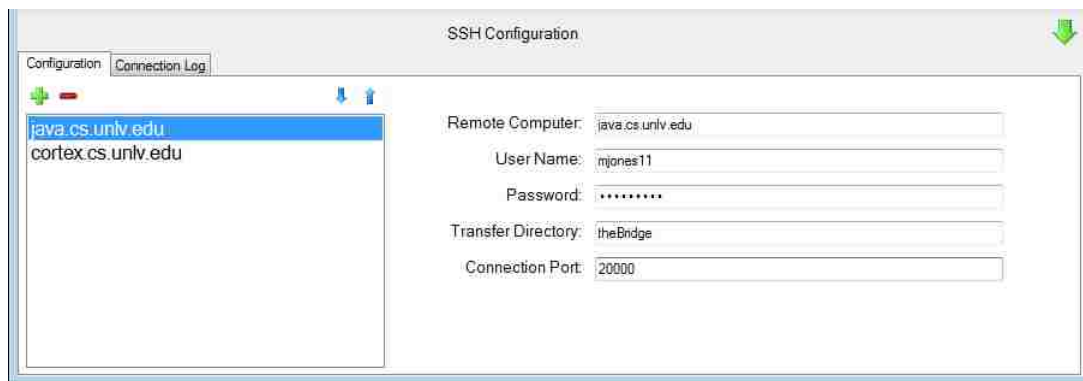
Figure 3.3: The Client configured for an indirect connection to The Call Center using Bridges.

pressing it. Once The Client has established a connection, it copies some debugging files to the folder indicated by the *Transfer Directory* from Figure 3.2. After the debugging information is copied, The Client compiles the debugging files and launches the second main component, The Call Center. In the case that the user provided multiple addresses for The Client to log into, the files will be copied up to each computer in sequence and a helper application called The Bridge, included in Appendix A.7, will be launched until the last connection is reached and The Call Center is launched. Figure 3.4 shows the system when The Client connects directly to the MPI cluster, and Figure 3.5 shows when extra connections are involved.

The configuration area in Figure 3.2 includes one more field which has not been talked about yet- namely the *Connection Port*. Once it has been established that The Bridges and Call Center are running, The Client will then establish an outgoing TCP connection to the first computer that it logged into on the connection port that was supplied. This computer then subsequently makes a connection to the next computer on that computer's connection port and so on until The Call Center has been reached. This connection port is provided to The Bridges and Call Center on their
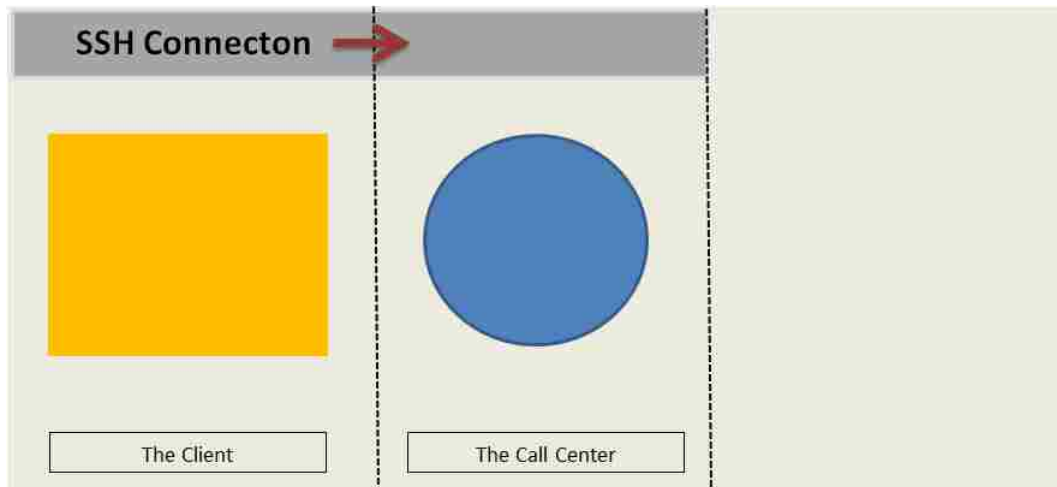
Figure 3.4: The Client after it has successfully connected to a remote computer and launched The Call Center.
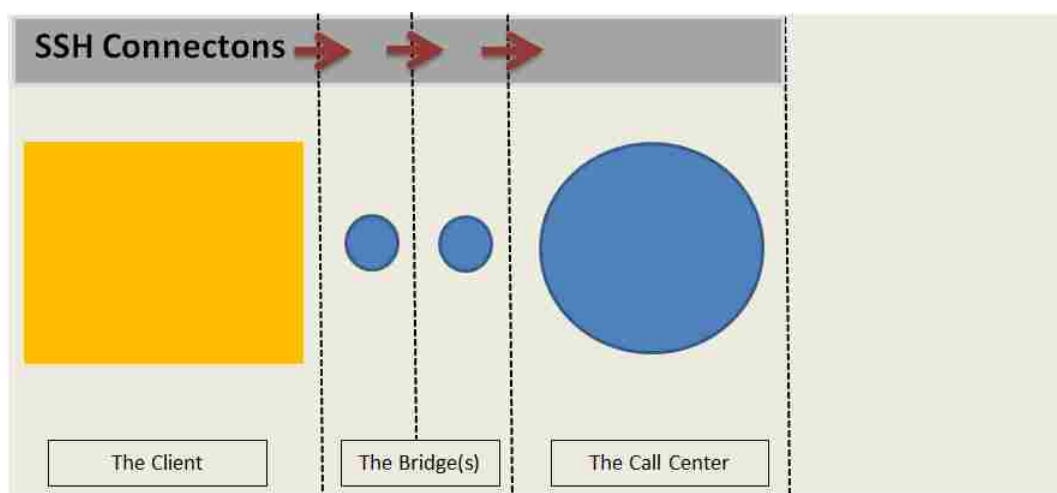
Figure 3.5: An updated overview with bridges included.

command lines as shown in Tables 3.1 and 3.2, and once the TCP connections are established, the remote connection sequence is complete. An updated illustration of the system is shown in Figure 3.6.

Command Line	
./callCenter PORT-NUMBER	
Arguments	
PORT-NUMBER	The TCP port to listen for incoming connections on.

Table 3.1: The command line for launching The Call Center.

Command Line	
./tcpBridge -b SRC-PORT-NUMBER DEST-IP-ADDRESS DEST-PORT-NUMBER	
Arguments	
-b	An indicator that the application is running in 'Bridge' mode.
SRC-PORT-NUMBER	The TCP port to listen for incoming connections on.
DEST-ADDRESS	The address to make an outgoing TCP connection to.
DEST-PORT-NUMBER	The port to make an outgoing TCP connection to.

Table 3.2: The command line for bridges.
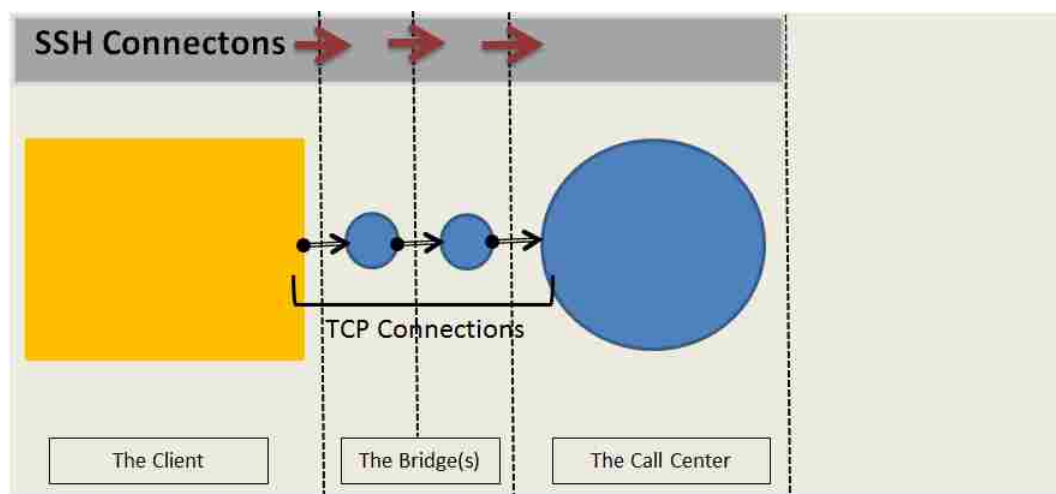
Figure 3.6: The system connected via TCP sockets.

3.1.2 Running a Debugging Session

Once the remote connection has been established, the user can start debugging. In the tool bar area of The Client, shown in Figure 3.1, are the fields for supplying the location of the executable to debug as well as the host file and the parameters. A node counter box is also present and is used to choose how many processes they would like to run. All information extracted from an MPI node during a debugging session is returned and displayed in the Node Panels section of The Client, also shown in Figure 3.1. A panel for every node present in the counter box is displayed in the node panels section. All node panels are initially shown within this area, but they can be collapsed by pressing the collapse button ⬒ present in each one of them. Figure 3.7 shows a session configured to debug a program called `TestAdd` with 2 node panels displayed, and Figure 3.8 shows the same session with one of the nodes collapsed to the tray.

Each node panel has three tabs, consisting of the Console tab, the Messages tab, and the MPI tab, in which it will display data about the debugging session. Figure 3.9 shows an illustration of each of the tabs before they have received debugging data. Although analyzing the data is saved for
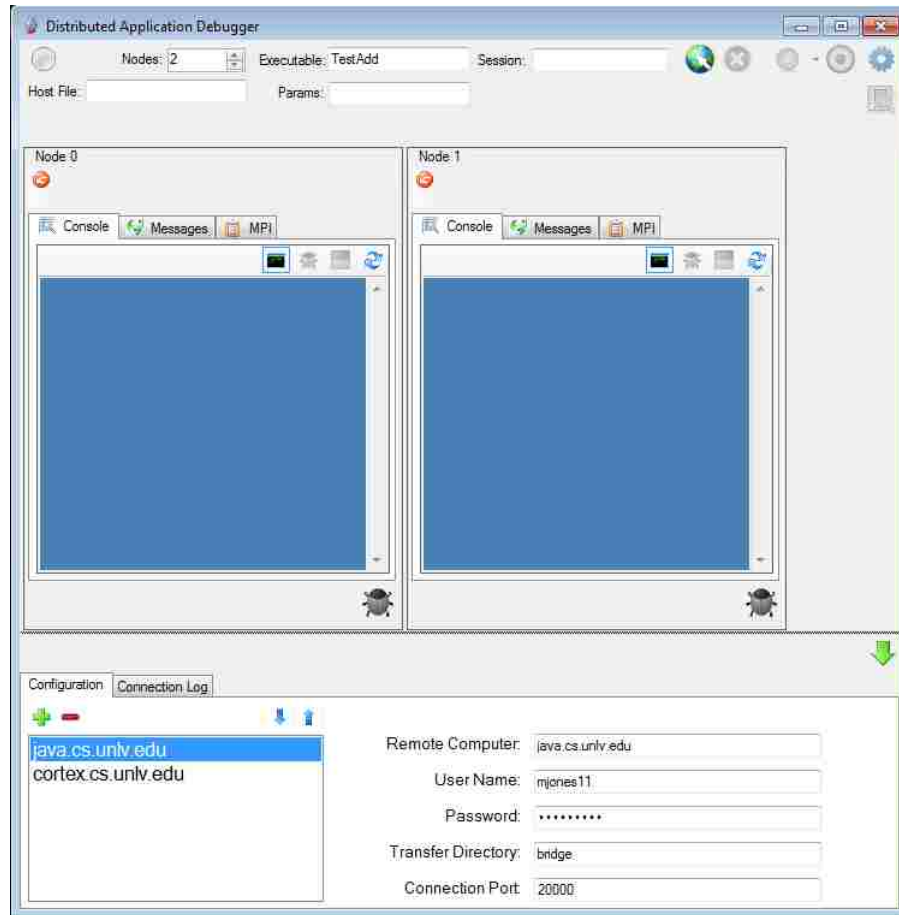
Figure 3.7: A debugging session configured for two nodes with both displayed.

chapter 4, I would like to touch on each of them here to give a general idea of what they are used for.

The *Console* Tab displays anything written to `stdout` from the node. This area primarily helps keep track of print statements used for debugging or for status within each of the nodes. Routing them to their own panel helps the user focus on the status output of one node at a time rather than viewing a jumble of nodes printing to the same monitor or physically moving in the case they get printed to separate monitors. The *Messages* Tab displays all of the interprocess messages sent between nodes within the MPI framework. Analysis can be done on this screen to match send and receive messages, highlight unmatched messages, and request the buffer values of recorded messages as shown in chapter 4. Finally, the *MPI* Tab displays all commands issued by the application. They are displayed in the order that they were executed along with their line numbers, so that the user can quickly see the path that their execution took. Both the Messages and MPI tabs offer a details section which shows parameter values passed into the command's method, as well as a filter, useful in selecting only specific types of commands for viewing.
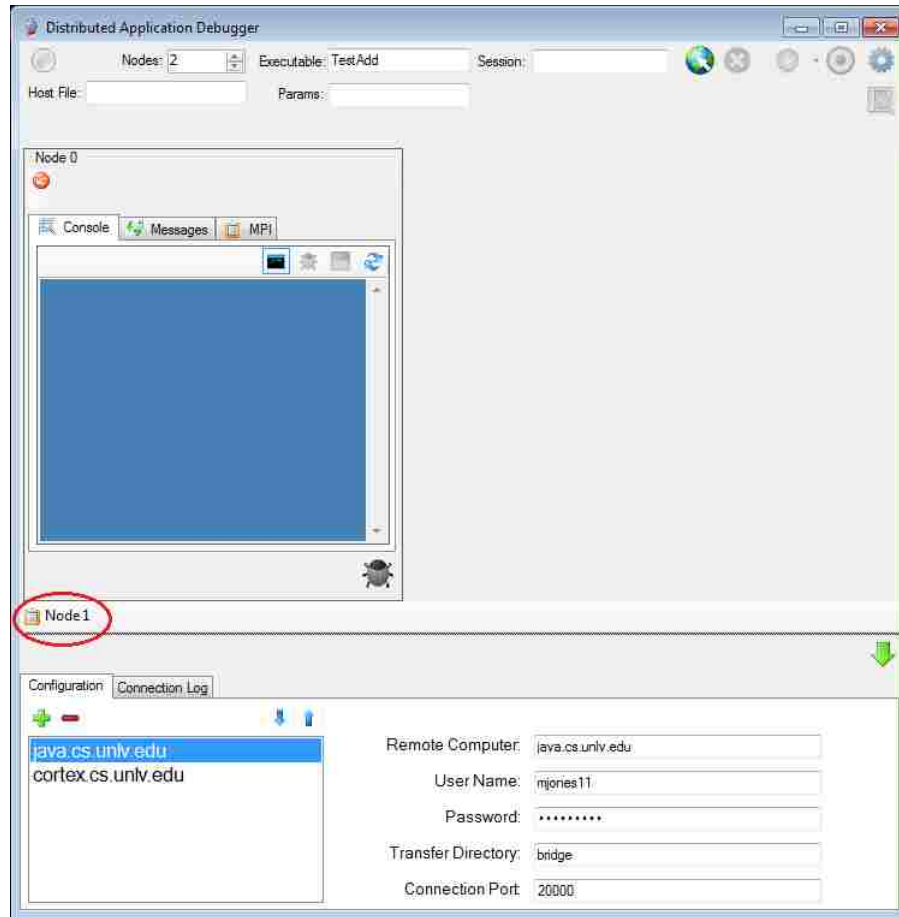
Figure 3.8: A debugging session configured for two nodes with one collapsed.

Once the debugging session information has been entered into the tool bar, the user can now begin to debug. A debugging session is run under one of three modes: PLAY, RECORD, or REPLAY and is issued within a general network messaging envelope used by all commands within the system and illustrated in Figure 3.10.

As detailed in [Hel00], all messages within the envelope start with a *Start of Header* control character, with the value 0x01, indicating that a new message packet is being read. Each message also ends with a control character called *End of Transmission* which is represented by the value 0x04. Sections of the envelope are partitioned by the third and final control character, the pipe character '|'. Going forward these three characters, the Start of Header, End of Transfer, and Pipe characters will be referred to as SOH, EOT, and partition respectively.
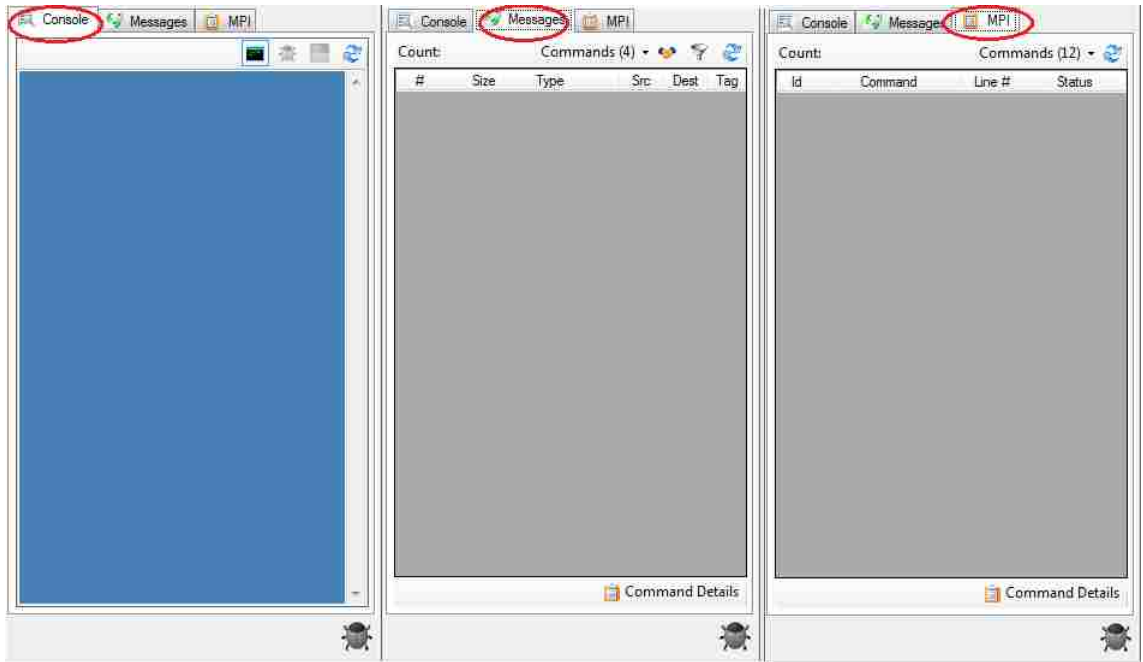
Figure 3.9: The three tabs of each node panel.
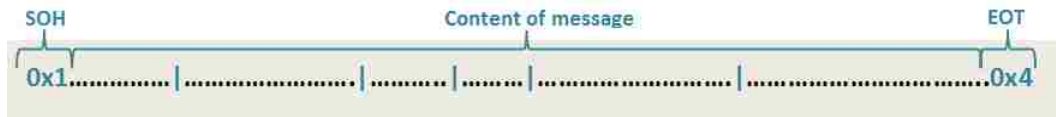
Figure 3.10: The protocol for all messages passed within the system.

Encoding Reserved Characters

Inherently when dealing with reserved characters, one must be assured that they will never be used outside of the score of within message envelopes. This is not possible, unfortunately, in the case of the data the we are dealing with because there is no stipulation on what data the MPI program may print to the screen or send in a buffer. In order to deal with this scenario, some commands, such as `PLAY`, `RECORD`, and `REPLAY`, include replacement strings for the three reserved characters. These replacement values will be used to encode literal uses of these reserved characters while they are being transferred back to The Client who can then decode them. These replacement strings are configurable by the user and can be of any length greater than 1. The default replacement strings for `SOH`, `EOT`, and partition characters are `*SOH*`, `*EOT*`, and `*BAR*` respectively.

Play

The `PLAY` command is the basic command to start a debugging session and is initiated by pressing the *Play* button found within the tool bar. It indicates to The Call Center that The Runtime should not do any recording, redirecting or analyzing of the MPI code. The parameters passed along with the `PLAY` command are the number of nodes to include, the location of the executable to run, the host file to use, the parameters for the mpirun command, the `SOH`, `EOT`, and partition encoding strings, and a comma delimited list of nodes to run under GDB. Figure 3.11 details the contents of a sample `PLAY` command sent from The Client to The Call Center indicating that 4 nodes should run the file called `addPrimes`, with a host file called `host.txt` and pass the parameters `1` and `99999`. `*SOH*`, `*BAR*`, and `*EOT*` are the strings to use to encode the reserved characters and nodes 1 & 2 will be run under GDB.


```
0x1PLAY|4|MPI/bin/addPrimes|hosts.txt|1 999999|*SOH*|*BAR*|*EOT*|1,20x4
```

Figure 3.11: The protocol for a sample `PLAY` command.

Record

A `RECORD` command is initiated by pressing the *Record* button in the tool bar. When sent to The Call Center, it indicates that The Runtime should record each command in the MPI session. When a `RECORD` session is run, every node will record the input parameters, buffer contents, and return values of each MPI command along each step of the program. As with the `PLAY` command, the parameters passed within the `RECORD` command are the number of nodes to include, the location of the executable to run, the host file to use, the parameters for the mpirun command, the `SOH`, `EOT`, and partition encoding strings, and a comma delimited list of nodes to run under GDB. The `RECORD` command also adds two extra parameters which are the location of the *Sessions* directory to save in and a name for the session being run.

When The Call Center receives a `RECORD` command, it creates a time stamp string and passes it, along with the user defined session name, to The Runtime. Each node then creates a folder within their Sessions directory named after the session name and timestamp, and stores the results of the session inside it in a file called `Node#.xml` where `#` is the node's id in the cluster. The complete directory path to the recorded session for each node is thus `Sessions/SessionName/TimeStamp/Node#.xml`. Figure 3.12 details the contents of a sample `RECORD` command sent from The Client to The Call Center with the same parameters as the `PLAY` command example above, along with parameters to save the details of the session to the `MPI/Sessions` folder under the session name `Homework1`.

26

Figure 3.12: The protocal for a sample `RECORD` command.

Replay

A `REPLAY` command is sent by choosing a *replay session* from the drop down below the *Play* button in the tool bar as shown in Figure 3.13. When sent to The Call Center it indicates that the user wants some, or all, nodes to play back values recorded from an early session. All nodes indicated as *replay nodes* will return values read from an XML file, rather than executing them to the MPI runtime. As with the `PLAY` and `RECORD` commands, the parameters passed along with the `REPLAY` command are the number of nodes to include, the location of the executable to run, the host file to use, the parameters for the mpirun command, the `SOH`, `EOT`, and partition encoding strings, and a comma delimited list of nodes to run under GDB. The `REPLAY` command also sends the location of the Sessions folder, the name of the session to replay, the time stamp of the instance of the session to replay, and a comma delimited string of nodes to run in `REPLAY` mode.

The Call Center passes these extra values to each of the nodes in the `REPLAY` session so that each node knows if it is to run in `REPLAY` and if the node that it may be exchanging messages with is in `REPLAY` too. For most commands, if the node is running in `REPLAY` mode, it will just return the values read from the XML recording of the session to the user. The only case in which a node which is running in `REPLAY` mode will actually make an MPI call is if the command is sending messages to, or receiving messages from, a node which is running in play mode. This is because that node may be blocked, waiting for a real message to be received by the MPI runtime system. Most messages however can just be read and returned to the user. `REPLAY` mode is helpful because it allows the users to focus on one particular node that he/she may feel is the source of a problem and to follow along with its XML file while execution is happening to determine what went wrong. Figure 3.14 details the contents of a sample `REPLAY` command sent from The Client to The Call Center requesting that it replay the `Homework1` session recorded in Figure 3.12 with nodes 2, 3, and 4 running in `REPLAY` mode.

Status Indicators

The tool bar offers a visual status indicator as to what state it is in while it is establishing a connection and running a debugging session. Table 3.3 provides a quick reference to what the different status colors indicate.

27
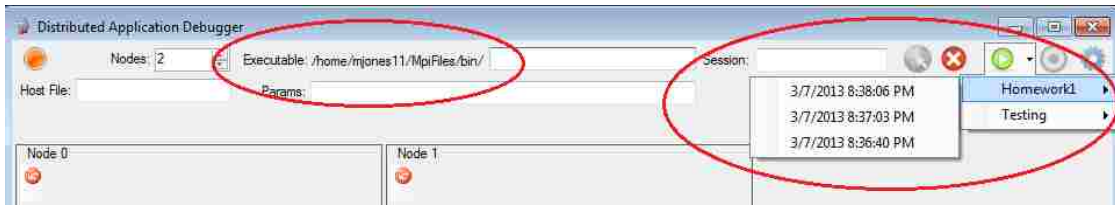
Figure 3.13: The Environment Data response integrated into a connected Client.

Figure 3.14: The protocol for a sample replay command.

Indicator	Status
	Idle
	Connecting
	Connected
	Session Running
	Session Complete
	Error

Table 3.3: The status indicators displayed on The Client.

Attaching GDB

The subject of attaching GDB to each of the nodes from the front end is discussed later in this chapter, in section 3.4, after describing how the rest of the system is implemented.

3.2   The Call Center

The Call Center is the central information processing application to the entire system. It is responsible for accepting connections from The Client, responding to requests to begin MPI sessions, retrieving buffer values, and managing all messages passed back from the MPI runtime in an orderly fashion. After establishing connections to The Runtime, the job of The Call Center is to reliably relay full messages from each node, in the order they were executed, back to The Client for the user to inspect.

3.2.1   Incoming Commands

While The Call Center is running, it can take in seven different requests. Four of these commands `ENVIRONMENT`, `BUFFER REQUEST`, `GDB INPUT`, and `MPI COMPLETE` do not initiate an MPI session, but rather support data in and around the sessions. The other three, `PLAY`, `RECORD`, and `REPLAY` do initiate an MPI run session and prompt The Call Center to spawn extra threads in order to listen to TCP connections and messages sent back from The Runtime.

ENVIRONMENT

Upon connecting to The Call Center The Client will issue an `ENVIRONMENT` request which is meant to retrieve a special folder location configured at startup where the users will be storing their MPI files. Accompanying the MPI folder is an XML section representing the header information of all of the previously recorded sessions. The request consists of the header `ENVIRONMENT`, along with the path of the session folder to pull session data from, and the response starts with the header `ENVIRONMENT DATA` followed by the requested data. Figures 3.15 and 3.16 show examples of the request and the corresponding response values from the `Environment` and `Environment Data` messages.

Figure 3.15: The `Environment` request sent to The Call Center.

Upon receiving the `Environment Data` response, The Client prefixes the location of the MPI folder to the user's execution location, so that the user does not have to type the full path to their executable. It also parses the various session information header packets in order to allow the user to choose to replay any session which was previously recorded. Figure 3.13 shows The Client displaying the prefixed folder, and replay drop down list, which were populated from the result of issuing an `ENVIRONMENT` request to The Call Center.

BUFFER REQUEST

The Distributed Application Debugger sends back everything known about each command requested except for buffer values. This was a conscious decision in order to cut down on the size of the messages sent back. In the case of messages being exchanged that contain buffers that are hundreds or even thousands of bytes long, it was decided that it was in the best interest of the user to, firstly, get the session completed as soon as possible and then, secondly, let the user request individual buffers that they were interested in. The only stipulation, however, is that the session must have been in `RECORD` mode since buffer values are not kept in memory by The Runtime, but are saved to

0x1ENVIRONMENT DATA|/home/mjones11/MpiFiles/bin/|

<SessionInfo>
 <SessionName>Homework1</SessionName>
 <StartTime>2013-03-07T20:38:06</StartTime>
 <Nodes>2</Nodes>
 <HostFile></HostFile>
 <Location>/home/mjones11/MpiFiles/bin/TestAdd</Location>
 <Parameters>999 10000</Parameters>
</SessionInfo>

<SessionInfo>
 <SessionName>Homework1</SessionName>
 <StartTime>2013-03-07T20:37:03</StartTime>
 <Nodes>2</Nodes>
 <HostFile></HostFile>
 <Location>/home/mjones11/MpiFiles/bin/TestAdd</Location>
 <Parameters>999 10000</Parameters>
</SessionInfo>

<SessionInfo>
 <SessionName>Homework1</SessionName>
 <StartTime>2013-03-07T20:36:40</StartTime>
 <Nodes>2</Nodes>
 <HostFile></HostFile>
 <Location>/home/mjones11/MpiFiles/bin/TestAdd</Location>
 <Parameters>100 99999</Parameters>
</SessionInfo>

<SessionInfo>
 <SessionName>Testing</SessionName>
 <StartTime>2013-02-12T18:39:05</StartTime>
 <Nodes>2</Nodes>
 <HostFile></HostFile>
 <Location>/home/mjones11/MpiFiles/bin/test</Location>
 <Parameters>1</Parameters>
</SessionInfo>0x4

Figure 3.16: The `Environment Data` response.

disk in the case of `RECORD`. In order to retrieve buffer values, they must first request them by right clicking on one or more message commands and selecting *Get Buffer* as displayed in Figure 3.17

When the user selects one or more commands to retrieve buffers for, The Client will issue a buffer request in the form of `BUFFER REQUEST|NodeId|DestinationCommand.SourceCommand,...|FileLocation|SOH string|Partition string|EOT string`. The third parameter is a comma delimited string of pairs in the form DestinationCommand1.SourceCommand1, DestinationCommand2.SourceCommand2 etc. The Destination Command is the command id that the buffer has been requested for, but the Source Command is the one which actually contains the buffer value. In most cases the Destination and Source will be the same, but, in the case of asynchronous communication, they could be different. For instance in the case that the buffer is requested for an *MPI_IRecv()* command, the buffer will not be found in the XML stored for that command since it was not known at the time that the command was issued. The buffer was not actually known until an *MPI_Wait()* was issued which will mean inspecting the XML for a different command than the message command that it belongs to. Figure 3.18 shows the contents of a sample buffer request
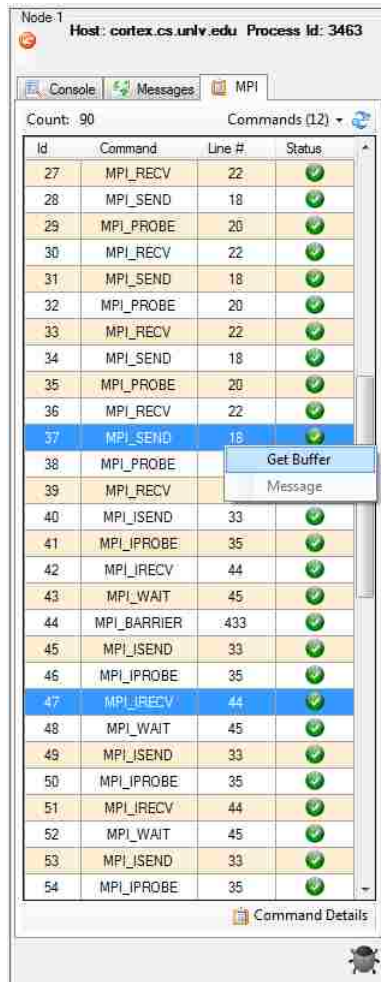
Figure 3.17: The user selecting to retrieve two buffer values.

message issued for two buffers, the second of which has a different destination command id than source command id.

Figure 3.18: A BUFFER REQUEST command issued to retrieve to buffer values from The Call Center.

Upon receiving the BUFFER REQUEST command,The Call Center parses the message, opens the file indicated and returns a Buffer Value response for the buffers requested in separate messages. The Buffer Value command is returned in the form BUFFER VALUE|NodeId|CommandId |EncodingIndicatorByte|Buffer Value1|Buffer Value 2|Buffer Value 3..... Because it is possible that the values in a buffer could be one of the three control characters, the buffer values command

31

will encode any reserved characters based on the encodings passed in from the buffer request command. In order to let The Call Center know if any of the buffer values where encoded, an *EncodingIndicatorBuffer* character is included. If any bytes were encoded an 'E' is returned to encoded data, and if not, a 'U' is returned to indicate unencoded data. The contents of the two responses to the request in Figure 3.18 are illustrated in Figure 3.19.

```
0x1 BUFFER VALUE|1|37|U|0.000000| -45.123400|245.333300|356.134143|76.554430|
68.342213| 6555167.651547|56.000000|586.000000|25.0000000x4

0x1 BUFFER VALUE|1|47|E|*SOH*|*BAR*|*EOT*|$|%|^|&|*|(|)|-|0x4
```

Figure 3.19: Two buffer value responses returned from The Call Center.

The Call Center supplies a buffer inspection panel whose icon becomes enabled within the tool bar whenever The Client is in `RECORD` or `REPLAY` mode. The buffer inspection panel displays the decoded values of each index of the buffer and, in the case of non-numeric buffers such as *MPI_BYTE* or *MPI_CHAR*, includes a hexadecimal translation column as well. Figure 3.20 and 3.21 show the two buffer results returned from the buffer request issued in Figure 3.18.

GDB INPUT

When The Client has attached GDB to one or more nodes, the user can issue individual commands to preform any command supported by GDB. These commands can be issued from The Client by using the `GDB INPUT` command. The `GDB INPUT` request is issued in the form `GDB INPUT|NodeId|GDB Command`. Upon receiving the request, The Call Center parses the command and routes it to the appropriate node. GDB then interprets the command and the result is printed to the console as in any GDB command. Figure 3.22 shows a sample `GDB INPUT` command requesting The Call Center to route the GDB command *display* to node 1 in order to print the value of a variable called `loopCounter` to the screen.

MPI COMPLETE

The last non-session command accepted by The Call Center is the `MPI COMPLETE` command. This command is issued by The Client to indicate that it has received and processed an *MPI_Finalize()* command for every node in the cluster and that it does not intend to listen for any more commands from this session. Upon receiving the `MPI COMPLETE` command, The Call Center releases any memory allocated to that session, cleans up any MPI nodes still controlled by GDB, and any other cleanup processes required. It is important to note, however, that The Call Center does not close itself down,
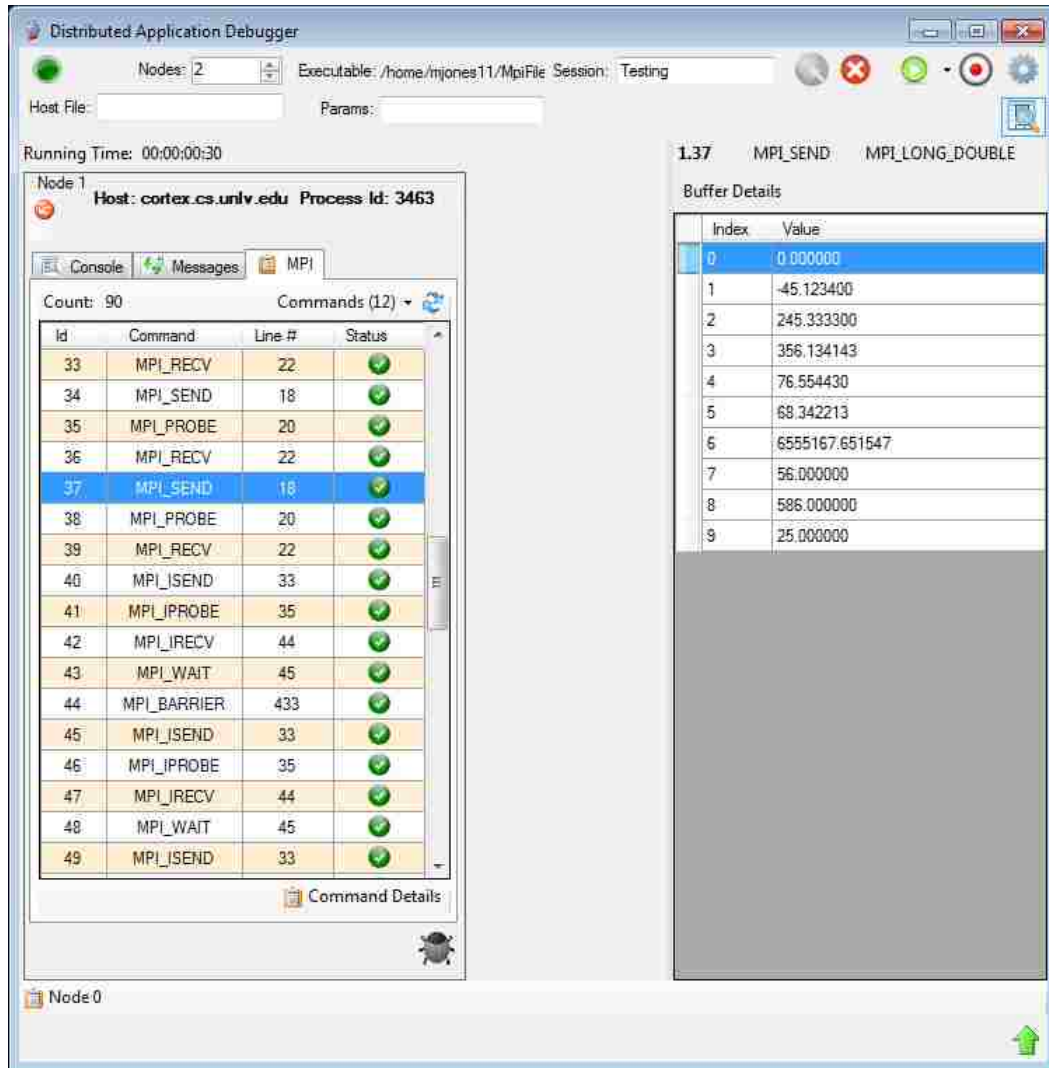
Figure 3.20: The Client displaying the buffer contents returned from The Call Center.

nor does it close the SSH session it was launched from or the TCP connection it has back to The Client. This allows for further sessions to be deployed quickly from The Client, without the user having to incur all of the overhead associated with starting up The Call Center. The envelope for the MPI COMPLETE command is illustrated in Figure 3.23.

Session Commands

The Call Center accepts the three session requests detailed earlier: PLAY, RECORD and REPLAY. Upon receiving one of these commands, The Call Center initiates a request to the command line and appends extra parameters for The Runtime to interpret. As with starting MPI from any other command
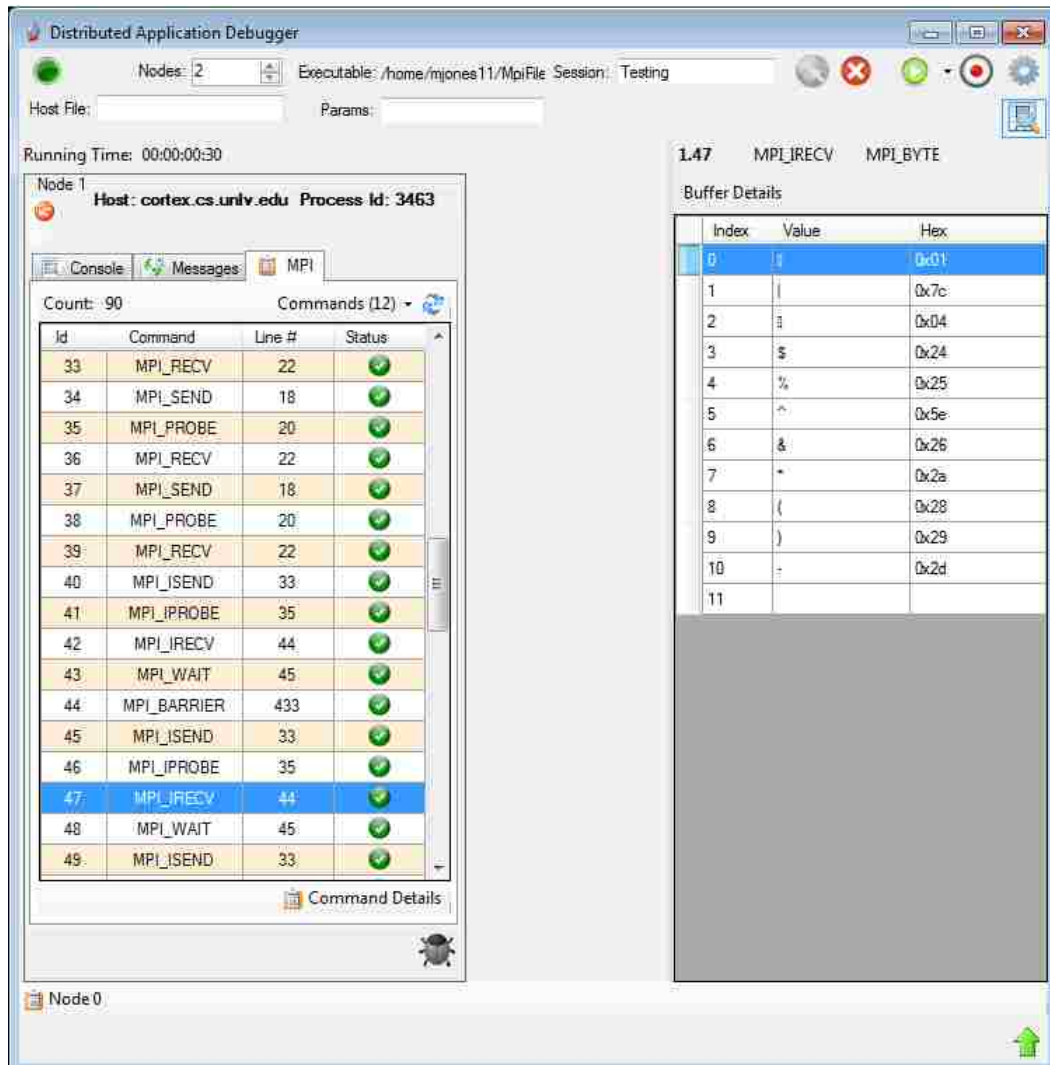
Figure 3.21: The Client displaying the buffer contents returned from The Call Center with Hex values.

Figure 3.22: A gdb input command issued to request a variable value from GDB.

Figure 3.23: The MPI Complete command.

line, it is initiated with an `mpirun` command with the corresponding parameters dictated from the contents of the `PLAY`, `RECORD` and `REPLAY` requests. The MPI `mpirun` command line arguments are displayed in Table 3.4.

Command Line	
mpirun -np <number of processes> -machinefile <hostFile > <program> <arg1 arg2 ....>	
Arguments	
-np <number of processes>	Specification of the number of processors to run.
-machinefile <hostFile>	A file of names of possible machines to run.
<program> <arg1 arg2...>	The MPI program and arguments to run.

Table 3.4: The standard mpirun command line arguments.

PLAY

As described in Figure 3.11, the `PLAY` request contains the number of nodes, host file name, file name, file arguments, encoding replacements for the `SOH,EOT`, and partition characters, and the list of nodes to be controlled by GDB. The `PLAY` command then appends the flags and values displayed in Table 3.5 to the end of the standard `mpirun` parameters list.

Arguments	
-s <SOH, Partition, EOT replace>	The values to substitute for the control characters.
-g <GDB Node 1, GDB Node 2 ...>	A command delimited list of nodes to run under GDB.
-f <Full file path>	The file path of the executable to be supplied to GDB.
-c <Address:Port>	The Call Center address and port to connect back to.

Table 3.5: The extra command line arguments appended to mpirun from a `PLAY` request.

RECORD

As described in Figure 3.12, the `RECORD` request contains all of the parameters supplied in the `PLAY` request along with the location of the session folder to store the results of an MPI session in and the name of the session to record them in. Like the `PLAY` command, the `RECORD` command appends its values to the end of the standard `mpirun` parameters list as dictated in the Table 3.6.

Arguments	
-s <SOH, Partition, EOT replace>	The values to substitute for the control characters.
-g <GDB Node 1, GDB Node 2 ...>	A command delimited list of nodes to run under GDB.
-f <Full file path>	The file path of the executable to be supplied to GDB.
-c <Address:Port>	The Call Center address and port to connect back to.
-r	Indicator for `RECORD` mode.
-d <Folder Path>	The directory to store XML recordings to.

Table 3.6: The extra command line arguments appended to mpirun from a `RECORD` request.

REPLAY

As described in Figure 3.14, the `REPLAY` request contains all of the parameters supplied in the `RECORD` request along with the time stamp of the specific session to replay and the comma delimited list of nodes to replay. Like the `PLAY` and `RECORD` commands, the `REPLAY` command appends its values to the end of the standard `mpirun` parameter list as detailed in Table 3.7.

Arguments	
-s <SOH, Partition, EOT replace>	The values to substitute for the control characters.
-g <GDB Node 1, GDB Node 2 ...>	A command delimited list of nodes to run under GDB.
-f <Full file path>	The file path of the executable to be supplied to GDB.
-c <Address:Port>	The Call Center address and port to connect back to.
-p <Replay Nodes>	The comma delimited list of nodes to replay.
-d <Folder Path>	The directory to read XML recordings from.

Table 3.7: The extra command line arguments appended to mpirun from a `REPLAY` request.

3.2.2 Message Routing

Once The Call Center has started an MPI session, it will begin to receive one call back connection per MPI node as described in Section 3.3.2. Once all of these connections are made the system becomes fully connected, as illustrated in Figure 3.24, with The Call Center managing multiple incoming connections from the MPI runtime and only one outgoing connection back to The Client. In order to efficiently route messages from multiple sources back to The Client, The Call Center employs a multi-threaded model to multiplex between reading from the incoming nodes, queuing the messages read, transferring them to the output queue, and then writing them back to The Client. This system of message routing is illustrated in Figure 3.25 and described further in this section.

Upon receiving the initial incoming connection from The Client, The Call Center creates one outgoing message thread which blocks while waiting on a semaphore known as the *outgoingQueueNotification*. When this semaphore is released, due to a thread posting to it, the outgoing message thread attempts to acquire the *outgoingQueueLock* which ensures that it can pop messages off of the outgoing queue, (see Appendix A.2), and write back to The Client in a thread safe manner. The outgoing message queue is populated from the messages passed in from the MPI nodes. After launching the MPI runtime due to receiving a `PLAY`, `RECORD`, or `REPLAY` request, The Call Center receives connections back from the MPI nodes. When The Call Center detects a call back connection from an MPI node, it creates a *clusterNode* structure (see Appendix A.4) which stores the connected socket to read from and a local message queue to write to. It then launches two threads, one Reading and one Processing, which work in tandem to read from an MPI node and move the messages read to
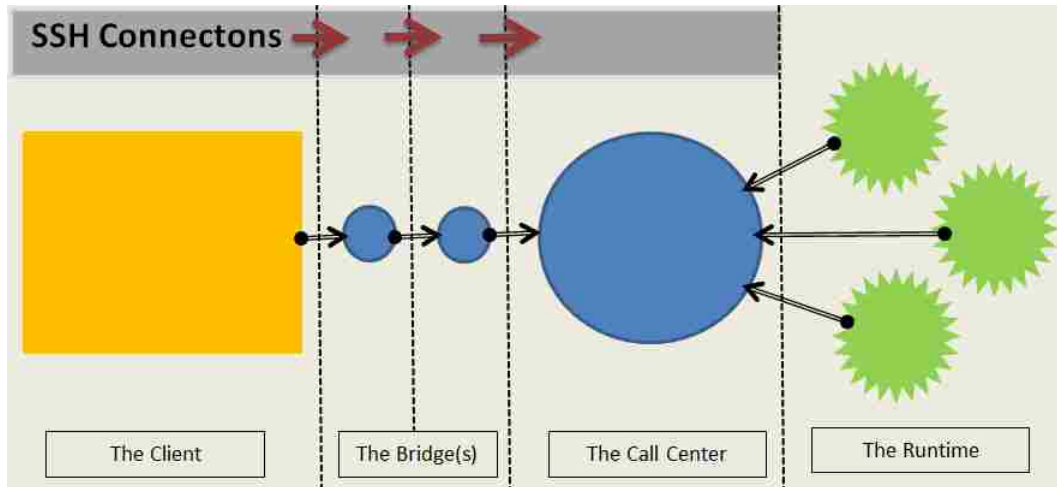
Figure 3.24: The system connected fully connected from The Client to the MPI nodes via The Call Center.

the outgoing message queue. It is important to note that while each MPI node has a local message queue and a set of Reading and Processing threads, that The Call Center has only one outgoing queue and one thread writing messages from it back to The Client.

The Reading thread's purpose is to read in messages from an MPI node and populate a local message queue without worrying about pushing these messages to The Client. It creates a *charList* structure (see Appendix A.1) dedicated to the MPI node it is reading from. Whenever something is read from the node, the data is appended to the *charList*. The reading thread then examines the list to see if there were any full messages written by detecting SOH and EOT characters and, upon detecting a full message, acquires the *clusterNodeLock* semaphore, transfers that portion of the charList to the node's *messages* queue, and alerts the process thread to handle transferring those messages to the output queue by posting to the *messageNotification* semaphore.

The Processing thread's purpose is to move these messages to the outgoing queue when no other thread is adding or removing from it. It blocks by waiting on the *messageNotification* semaphore and, upon acquiring it by being notified by the Reading thread, acquires the outgoing thread's *outgoingQueueLock* semaphore. Upon acquiring the *outgoingQueueLock* semaphore, the processing thread then acquires the *clusterNodeLock* semaphore to stop any new messages read from the MPI node from being placed on the queue. It then transfers all messages from the node's queue to the outgoing queue, notifies the outgoing thread that messages are waiting for it by posting to the *outgoingQueueNotification* semaphore, releases the *clusterNodeLock* and *outgoingQueueLock* queue semaphores, and waits on the *messageNotification* semaphore again. The outgoing thread, as stated earlier, will detect the *outgoingQueueNotification* semaphore, acquire the *outgoingQueueLock*
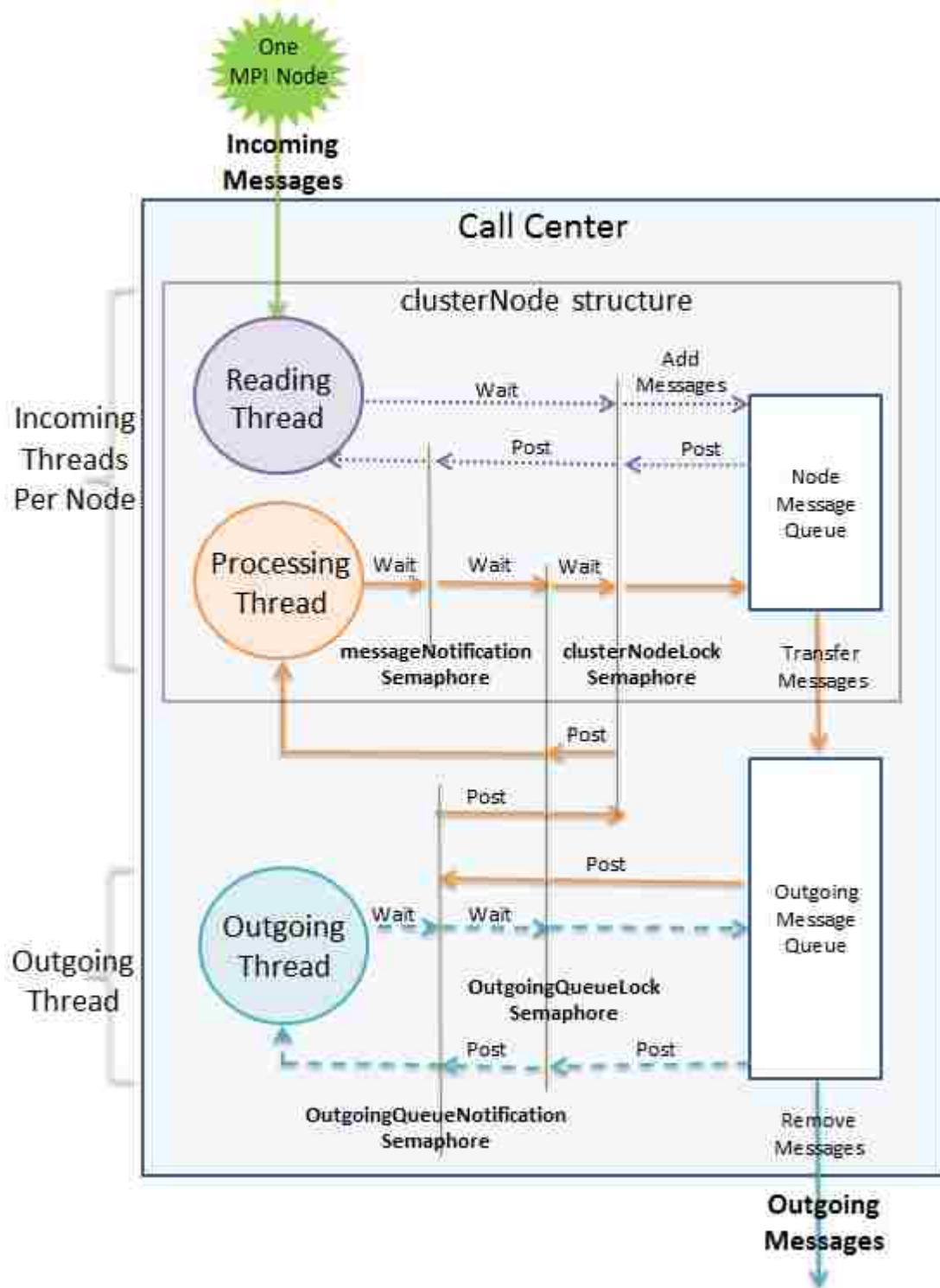
Figure 3.25: Two threads per MPI node populating one outgoing message queue.

38

sempahore, and safely write all queued messages back to The Client.

3.3   The Runtime

The Runtime is the final part of the system which links the user's MPI code to the remote debugging Client. The Runtime runs below the user's MPI code which is being debugged, and provides a level of indirection which handles recording, replaying, and sending data back to The Call Center which will relay them back to The Client. The key to the user passing control of their MPI code that they want to be debugged is in the header file that they import.

3.3.1   Importing `mpi.h`

Appendix B.3 details the 4 step set of instructions on how to integrate The Runtime into a user's code. Step 3 indicates that, in order to allow The Runtime to reflect on the user's code, the user needs to replace the inclusion of the MPI framework's `mpi.h` header file, show in Figures 3.26, with The Runtime's `mpi.h` header file shown in Figure 3.27. This one step completely transforms the user's code at compile time, without them having to know, and redirects their MPI library calls to The Runtime which can then provide debugging support.

Figure 3.26: Including the standard MPI framework.

Figure 3.27: Including the debugging MPI framework.

The contents of The Runtime's `mpi.h` file, which is displayed in Figure 3.28, becomes included in the user's code. On line 1 of the `mpi.h` header file is an `include` statement for the MPI framework's real `mpi.h` file, which ensures that calls to the MPI library can made. After this is a conditional statement depending upon if the *MPIDEBUG* flag has been compiled as described in the last step in Appendix B.3. When the *MPIDEBUG* flag has been set because The Runtime libraries have been compiled and included, the file now imports two other header files: `debug.h`, shown on line 4 of Figure 3.28, and `mpidebug.h` as shown on line 5. After that the user is set to begin debugging with The Runtime because the contents of `debug.h`, shown in Figure 3.29, and `mpidebug.h`, included in Appendix B.1, contain the necessary source code to redirect the user's source code to The Runtime's assemblies.

The `debug.h` file only contains macros as shown in Figure 3.29. What these effectively do is replace any calls to the MPI library methods, with new ones which are prefixed with an underscore. An example program which gets its MPI calls replaced at compile time is displayed in Figure 3.30. In

```
1  #include <mpi.h>
2
3  #ifdef MPIDEBUG
4  #include "debug.h"
5  #include "mpidebug.h"
6  #endif
```

Figure 3.28: The contents of the `mpi.h` file included with The Runtime.

```
#ifndef _DEBUG_
#define _DEBUG_
#define  MPI_Init(A,B)              _MPI_Init(__FILE__,__LINE__,A,B)
#define  MPI_Finalize()            _MPI_Finalize(__FILE__,__LINE__)
#define  MPI_Comm_rank(A,B)        _MPI_Comm_rank(__FILE__,__LINE__,A,B)
#define  MPI_Comm_size(A,B)        _MPI_Comm_size(__FILE__,__LINE__,A,B)
#define  MPI_Send(A,B,C,D,E,F)     _MPI_Send(__FILE__,__LINE__,A,B,C,D,E,
    F)
#define  MPI_Recv(A,B,C,D,E,F,G)   _MPI_Recv(__FILE__,__LINE__,A,B,C,D,E,
    F,G)
#define  MPI_Isend(A,B,C,D,E,F,G)  _MPI_ISend(__FILE__,__LINE__,A,B,C,D,E
    ,F,G)
#define  MPI_Irecv(A,B,C,D,E,F,G)  _MPI_IRecv(__FILE__,__LINE__,A,B,C,D,E
    ,F,G)
#define  MPI_Wait(A,B)             _MPI_Wait(__FILE__,__LINE__,A,B)
#define  MPI_Barrier(A)            _MPI_Barrier(__FILE__,__LINE__,A)
#define  MPI_Probe(A,B,C,D)        _MPI_Probe(__FILE__,__LINE__,A,B,C,D)
#define  MPI_Iprobe(A,B,C,D,E)     _MPI_IProbe(__FILE__,__LINE__,A,B,C,D,
    E)
#endif
```

Figure 3.29: The contents of the `debug.h` file included with The Runtime.

this figure *MPI_Init()* has been replaced with *_MPI_Init()* at compile time, *MPI_Barrier()* has been replaced with *_MPI_Barrier()*, *MPI_Send()* has been replaced with *_MPI_Send()*, and so on because of the macros in `debug.h`. In this scenario all calls to the *underscore* versions of these methods, which are contained in `mpidebug.h` from line 6 of Figure 3.28 and implemented by The Runtime, provide The Runtime a chance to connect back to The Call Center up *MPI_Init()* being called, as detailed in Section 3.3.2, and then send back debugging information for The Client, as detailed in Section 3.3.3.

3.3.2   Connecting to The Call Center

When each node joins the MPI system by initiating an *MPI_Init()* command, a series of runtime initialization steps happen. First, The Runtime version of *MPI_Init()* issues a real *MPI_Init()*

Figure 3.30: Compile time changes made from including The Runtime's `mpi.h` file.

command to join the MPI group that will run. It also issues an *MPI_Comm_size()* and an *MPI_-Comm_rank()* command so that when it parses the input commands it can determine which nodes, including itself, are `-p` nodes, in the case of a `REPLAY` session, and store each node's role in a lookup table.

Once The Runtime knows what the size of the cluster is and what its rank is, it then parses the extra command line parameters passed in from The Call Center. It assumes that the entire system will run in regular play mode, but if a `-r` is detected it notes that it is running in `RECORD` mode, and if a -p is detected it notes that it is running in `REPLAY` mode and determines which nodes in the cluster are replay nodes and which are normal. It also stores the data passed from `-d` for the directory path, `-f` for the executable file location, `-s` for the `SOH`, `EOT`, and partition control character

replacement strings, `-g` to determine if it is supposed to let GDB attach, and `-c` for the address of The Call Center and the port that it is listening for connections on.

After parsing the input strings, the system redirects `stdout` to a different file descriptor and spawns off a thread dedicated to listening to this file descriptor as detailed in Appendix B.2. Finally the system makes a TCP connection back to The Call Center, passes its first message back which contains its node id, its process id, and computer name, and then waits for The Call Center to acknowledge that all of the nodes have connected back to The Call Center by reading a *Continue* command. Once all of the nodes have connected back to The Call Center and The Call Center acknowledges it is ready to start the session by writing *Continue* on each of there ports, the system is fully connected as shown in Figure 3.24.

3.3.3   MPI Session

Once The Runtime has connected to The Call Center during the *MPI_Init()* command, the system follows a four step pattern of executing the users code, providing status information back to the user, and recording and validating when appropriate. Figure 3.31 illustrates the 4 steps within a sample command, *_MPI_Send()*, and the other 11 commands follow the exact same pattern.

Step 1

The first thing that each command does is to create an 'expectedValues' XML node which will be read from an XML file, in the case that the node is running in `REPLAY` mode, or remain null otherwise.

Step 2

The second step is to write a *PRE* message, detailed in section 3.3.4 and illustrated in Figure 3.34, back to The Client to let it know what parameters are about to be executed on the next MPI command. In the case that the *expectedValues* XML was actually populated because the node is running in `REPLAY` mode, the method creating the *PRE* message will also validate that the parameter values passed in where the same as those read in from the XML file. In the case that parameters read in where different than the expected ones read from the XML file, the system will append those expected values to the end of the *PRE* message to, as detailed in **red** within Table 3.8, to let The Client warn the user that the session being played back is producing different values than the original instance.

Step 3

After the *PRE* message has been sent back to The Client, the system handles executing the actual MPI command based on the role that the node is running in. In normal `PLAY` mode, the system

always executes the real MPI command. In the case of `RECORD`, the system executes the real MPI command and then logs the XML representation of the command to file using the XML library I wrote and included in Appendix A.5. In the case that the node is running under `REPLAY` mode the results will vary based on the message. If the message is a send or a receive message and the source or destination is running in normal `PLAY` mode, the `REPLAY` node will actually execute the MPI command. When the code snippet below for the _MPI_Send()_ command is running in `REPLAY` mode, for instance, it must see if the node that it is sending to is running in normal `PLAY` mode first. If it is, then it must send the message because the destination node would be blocked waiting for the message to arrive. If the destination is, instead, running in `REPLAY` mode too, than there is no reason to send the message. Regardless whether if the node's mode requires that it actually executes the statement or not, the MPI command's return value is saved from the result of sending it or reading it from the replay XML in this step.

Step 4

Finally, the command sends a _POST_ message, detailed in section 3.3.4 and illustrated in Figure 3.35, back to The Client to let it know that the command actually completed and to obtain the parameter values that may have changed because of it along with the command's return value. Table 3.9 shows the values returned, along with which parameters are validated and flagged when incorrect in *red*.

3.3.4 Runtime Commands

F The Runtime posts five different commands to provide debugging status to The Client. Each is sent back to The Call Center without consideration of the other nodes who are also sending back data and do not invoke responses. It is the responsibility of The Call Center to keep the message packets received from being corrupted as detailed in Section 3.2.2. This section details the five messages sent back to The Call Center from The Runtime.

NODE ID

The first command, _NODE ID_ has already been touched on briefly. It is sent back to The Call Center immediately after The Runtime establishes a connection to give the the details of the MPI node's process id and the name of the computer it is running on. In the example illustrated in Figure 3.32, an MPI node reports that its rank is 1, its process id is 21809, and it is running on the machine `cortex.cs.unlv.edu`.

Unlike other commands sent back from the MPI nodes, The Call Center inspects this message first and makes a record of each node's details before passing this command back to The Client.

```
//MPI Send implementation
int _MPI_Send(char pname[100], int line, void *buf, int count,
              MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) {
    int commandId = 0;
    int returnValue = 0;

    //Get the next node if this is a playback situation
    XMLNode *expectedValues = NULL;
    if(_role == PLAYBACK_NODE)
        expectedValues = getNextNode();

    //Send a 'PRE' message that we are processing this command
    if(CommunicationConfigured() == TRUE)
        writeToClient(preSerializeMPISend(_rank, line, count,
                        datatype, dest, tag, comm, &commandId, expectedValues));

    switch(_role)
    {
      case NORMAL_NODE:
        //Just send as normal if the mode is normal
        returnValue = MPI_Send(buf, count, datatype, dest, tag, comm);
        break;

      case RECORD_NODE:
        {
            //Send the message as normal and serialize it to XML
            returnValue = MPI_Send(buf, count, datatype, dest, tag, comm);
            XMLNode *xmlNode = xmlMPISend(_rank, buf, count, datatype, dest, tag, comm,
            returnValue, commandId);

            //xmlWrite the XML node to file
            logAndDispose(xmlNode);
        }
        break;

      case PLAYBACK_NODE:
        {
            if(_playbackRoleList[dest] == NORMAL_NODE)
                returnValue = MPI_Send(buf, count, datatype, dest, tag, comm);
            else
                returnValue = atoi(xmlGetText(
                    xmlGetChildNode(expectedValues, RETURN_VALUE_ELEMENT)));
        }
        break;
    }

    //Send a 'POST' message with the actual return value
    if(CommunicationConfigured() == TRUE)
        writeToClient(postSerializeMPISend(_rank, datatype, count, buf, commandId,
            returnValue, expectedValues, _sohReplace, _partitionReplace, _eotReplace));

    return returnValue;
}
```

Figure 3.31: The four step pattern applied to The Runtime versions of MPI commands.

0x1NODE ID|1|21809|cortex.cs.unlv.edu0x4

Figure 3.32: A Node Id command reporting a node's process id and computer name.

When The Client receives a *Node Id*, it displays the details of each node's data at the top of their node panel as illustrated in in Figure 3.33.

Figure 3.33: The details of a node displayed in the header of its node panel.

PRE

The *PRE* command is meant to let The Client know that an MPI command is going to be executed, which is the second step of the MPI Session pattern identified in Figure 3.31 and discussed in section 3.3.3, and serves two purposes. First, it lets the person debugging their application know the line number, command name and command input parameters of the MPI command about to be executed. This information can help quickly identify what command a program may be blocking on in the case of a program that never completes or crashes. The second purpose it serves is to send back invalid data in the case of a `REPLAY` node. In this case the *actual values* parameter will be sent first and the *expected values* parameter, which is read from the XML file, is sent at the end when discrepancies from the *actual values* are found. The common structure of this command is displayed in Figure 3.34, as well as a break down of the command specific details displayed in Table 3.8.

Figure 3.34: The structure of the PRE command.

Details returned per command										
Command	Variable length details, delimited by '|' character.									
Init										
Rank	Comm									
Size	Comm									
Send	Count	Type	Dest	Tag	Comm	Count	Type	Dest	Tag	
Recv	Count	Type	Src	Tag	Comm	Count	type	Src	Tag	
Isend	Count	Type	Dest	Tag	Comm	Req	Count	Type	Dest	Tag
Irecv	Count	Type	Src	Tag	Comm	Req	Count	Type	Src	Tag
Probe	Src	Tag	Comm	Src	Tag					
IProbe	Src	Tag	Comm	Src	Tag					
Wait	Request									
Barrier	Comm									
Finalize										

Table 3.8: The *Actual Values* sent back within *PRE* commands, with detected discrepancy fields highlighted in red.

45

POST

The *POST* command confirms to the user that an MPI Command has finished which is the last step of the MPI Session pattern identified in Figure 3.31 and discussed in section 3.3.3. Like the *PRE* command, it serves two purposes of giving details of the results of executing the command such as the return value, and, in the case of a `REPLAY` node, includes expected values in case they did not match the actual values. The command id included in the *POST* command will always match up with a *PRE* command sent earlier. The two commands paint the complete picture of the values passed into the command and the results after they have completed. The common structure of this command is displayed in Figure 3.35, as well as a break down of the command specific details displayed in Table 3.9.

`0x1POST|Node Id|Command Id|Return Value| Expected Return|Actual Values|Expected Values0x4`

Figure 3.35: The structure of the POST command.

Details returned post command				
Command	Variable length details, delimited by '|' character.			
MPI_Init()	Return Value	Rank		
MPI_Rank()	Return Value	Size		
MPI_Size()	Return Value			
MPI_Send()	Return Value			
MPI_Recv()	Return Value	Status	Status	Buf
MPI_Isend()	Return Value			
MPI_Irecv()	Return Value			
MPI_Probe()	Return Value	Status	Status	
MPI_IProbe()	Return Value	Flag	Status	
MPI_Wait()	Return Value	Status	Status	Buf
MPI_Barrier()	Return Value			
MPI_Finalize()	Return Value			

Table 3.9: The *Return Values* sent back within *POST* commands, with detected discrepancy fields highlighted in red.

CONSOLE

Whenever The Runtime detects that something has been printed to `stdout`, it will send a *Console* message back to The Call Center. Like all messages, The Call Center will relay this back to The Client who will append it to the console window of the corresponding node's panel. The message format starts out with the message header *Console*, followed by the *node id*, and then, just like in The Call Center's *Buffer Value* command, it has an *EncodingIndictor* character that indicates if any

part of the final section, the actual printed message, was encoded because of the detection of reserved characters. Figures 3.36 and 3.37 display two *Console* message examples. Figure 3.38 represents the phrase 'Hello World' being printed to the screen and Figure 3.39 indicates 'Hello|World'.

0x1CONSOLE|1|U|Hello World0x4

Figure 3.36: An example CONSOLE message that does not have any encoded characters.

0x1CONSOLE|1|E|Hello*BAR*World0x4

Figure 3.37: An example CONSOLE message that contains encoded characters.
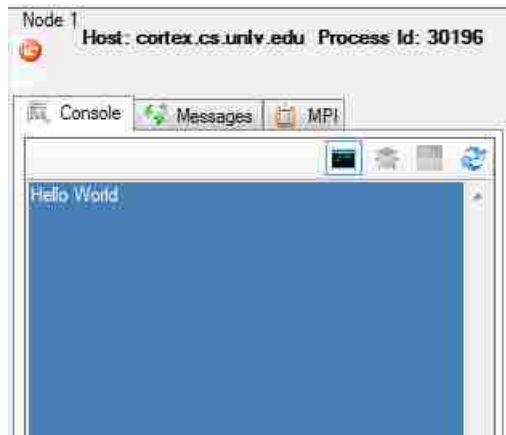
Figure 3.38: Unencoded message printed to console.
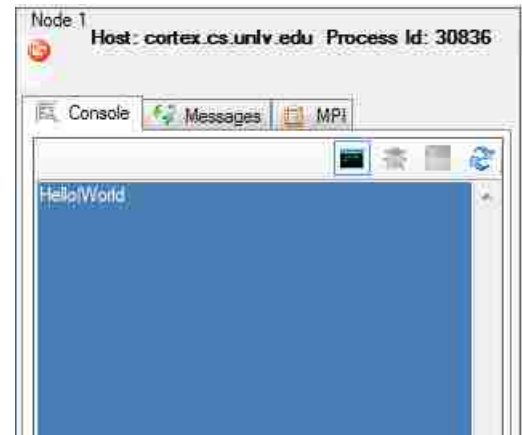
Figure 3.39: Encoded message printed to console.

GDB

The last command sent back from The Runtime is the *GDB* command. Like the *CONSOLE* command, the *GDB* command is meant to relay data, that is printed to `stdout`, back to The Client to display while having GDB attached. The Runtime listens to the `stdout` of the process running GDB and reports back every character GDB has written to the console. By way of the *GDB* command, the Distributed Application Debugger is able to route just the data written to `stdout` from GDB to a dedicated window for the user to focus on. The format of the *GDB* command is displayed in Figure 3.40 and the data printed to The Call Center in Figure 3.41.

0x1GDB|1|U|Reading symbols from /home/mjones11/MpiFiles/bin/TestAdd...0x4

Figure 3.40: A GDB command issued to tell some partial text of what has been written to screen.
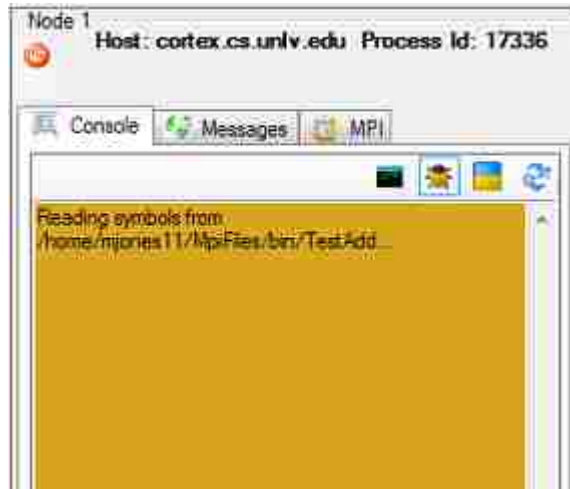
Figure 3.41: The content of the GDB routed to the GDB console display.

3.4 Integrating GDB

Perhaps the most useful and powerful feature of the Distributed Application Debugger is the integration of the GNU Debugger known widely as GDB. GDB is able to read the debugging symbols produced by the gcc [GCC13] compiler which is the same one used to compile MPI code. GDB can be used to launch applications or attach to already running applications. Once attached, the user has commands available at their disposal to break, step over, step into, and continue through lines of their program as they see their code's execution path. They can view variable values, recall stack traces, examine memory and perform a whole host of operations to help them figure out what is causing a problem in their software.

3.4.1 Attaching GDB

To use the GDB feature of the system, all the user has to do is click the gray debugger icon, , on the bottom of any node that they wish to attach the GDB debugger to. Once pushed, the icon turns yellow, , the debugging panel slides up and the panel's console panel defaults to the GDB view as displayed in Figure 3.42.

As described earlier, when the user initializes an MPI session through a PLAY, RECORD, or REPLAY command, the list of nodes to run under GDB mode are included within the request. When The Call Center receives the message, the list of nodes running under GDB get passed along to The Runtime,
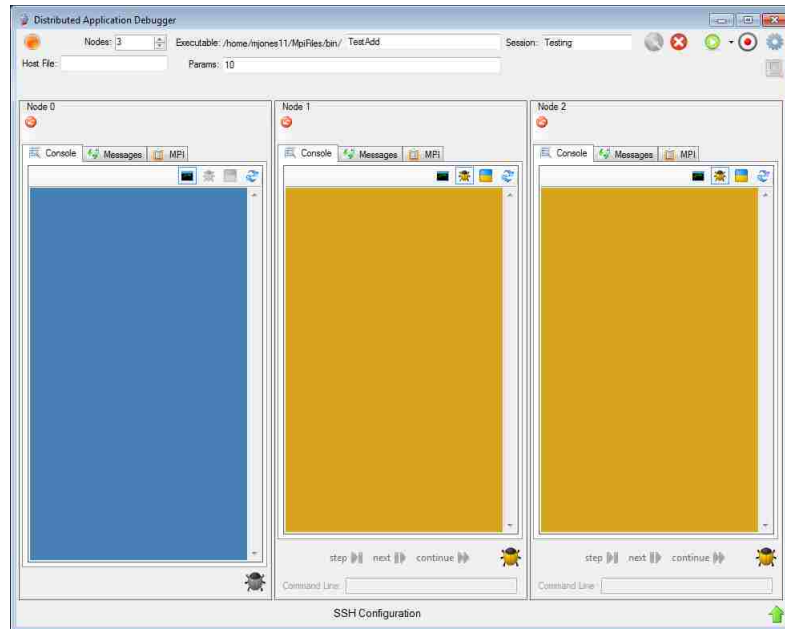
Figure 3.42: The Distributed Application Debugger with two nodes selected for GDB.

and the nodes start making connections back. After each node has connected back and The Call Center has issued a *Continue* statement over the TCP connection, it now becomes the responsibility of each of the nodes to determine if they are supposed to be running under GDB or not.

The nodes that will run under GDB go through a five step process in order to let GDB take over control of their process as shown in Figure 3.43. First, the node forks a second process. The parent process starts listening to the socket that was established at startup from The Call Center for a *GO* command that indicates that the GDB process has been completed. Secondly, the child creates two unidirectional pipes, named *fromParent* and *fromChild* and forks again. Next, the child process from this latest fork routes stdin to read from the reading side of the *fromParent* pipe and routes stdout and stderr to the writing side of the *fromChild* pipe. After that, the process launches GDB by issuing an *execvp* commnd which leaves the process as just GDB waiting for the name of the process to attach to.

The last step of the process is done by the parent process from the first fork. This process effectively becomes a bridge between the MPI process which is blocked from listening to The Call Center and the GDB process listening to stdin by way of the output side of the *fromParent* pipe. This bridging process makes a TCP connection of its own back to The Call Center who knows that this must be a GDB node since all of the initial callback connections have been made. After a connection has been established back to The Call Center, the bridging process creates two threads, one writing whatever it reads from the new Call Center connection to the GDB process, and one

49
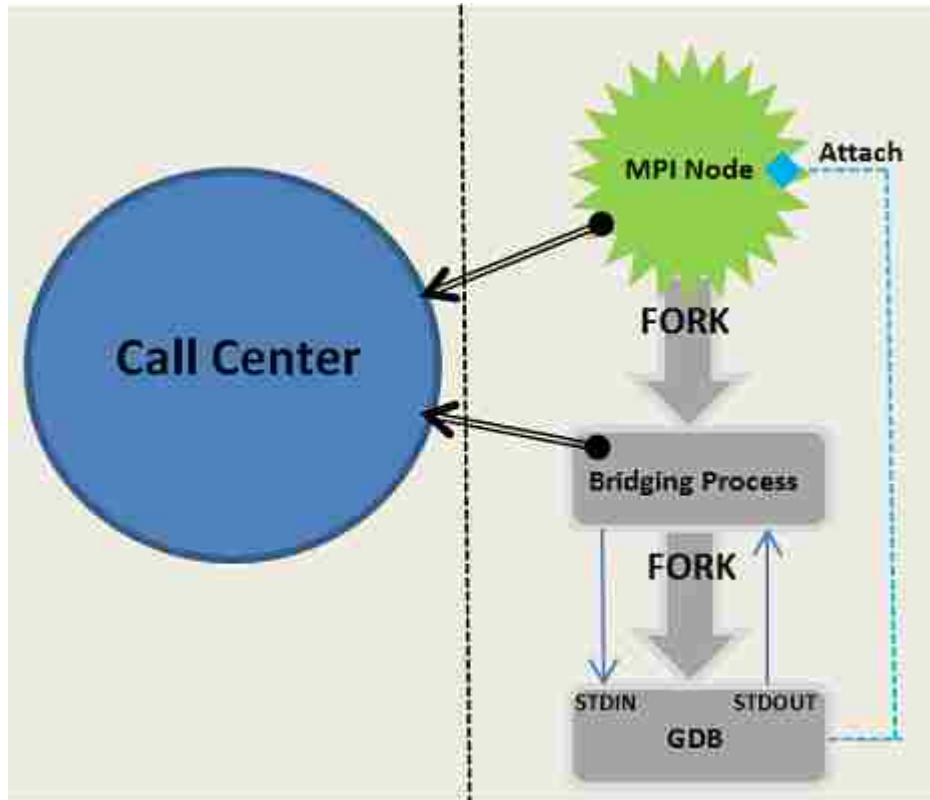
Figure 3.43: Forked processes make attaching GDB to the MPI node possible.

that writes whatever it from the GDB process to back to The Call Center. After this, the bridging process sends a message to The Call Center stating that it has completed setup and blocks for the rest of the session.

In the final step of the process, The Call Center writes *attach [pid]*, where [pid] is the process of the MPI node, to the bridging node. When the bridging node writes this to the pipe which it shares with GDB, GDB sees this command on its `stdin` line, and attaches to the MPI node completing the circle. The Call Center also writes the command *GO* to the TCP connection that it shares with the MPI Node which allows it to proceed from blocking on a receive. After this the setup is complete; The Call Center has two TCP connections to the node, one being to the real process and one being to GDB which has attached to the process.

3.4.2 Controlling GDB

Once all of the GDB nodes have called back and attached, the user now has full control over their process. The GDB control panel allows the user to issue commands both through a command line and through hotkeys as displayed in Figure 3.44.

50
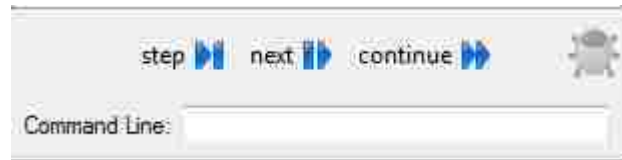
Figure 3.44: The control panel for a node with GDB attached.

All commands issued from the control panel will be formatted into a *GDB INPUT* command as documented in section 3.2.1. When The Call Center receives one of these commands, it parses out the destination node and writes whatever is written in the message section to the TCP connection that it has with the bridging process. This process writes the message to the `stdin` of GDB and thus it gets applied to the MPI Node. GDB will then write its result to `stdout` which will get picked up by the bridging process, who will then write it back along the TCP port back to The Call Center who will send a *GDB* command back to The Client as described in 3.4.1.

Chapter 4

Analyzing Data

The Client is designed to not only process the tremendous number of messages that come into it from The Runtime, but to also organize them in a way that helps the user quickly debug their program. On top of launching bridges and a Call Center capable of connecting the user to a cluster hosted remotely, coordinating `PLAY`, `RECORD` and `REPLAY` sessions, recalling MPI buffers and attaching GDB to MPI programs, the user has many options to organize the data within The Client workspace to help them focus on finding what the problem in their program really is.

Command Details

The *MPI tab*, displayed in Figure 3.9, displays the names and line number of all of the commands executed by a node during the MPI session. Because there is a lot more information about a message than just its name and line number that might be useful to a user, each MPI tab displays a *Command Detail* section in it. The *Command Details* section displays the command specific details of the command highlighted in the tray of the tab. This is a useful area to see the parameters and return value of a command in question and, by just keying down, the user can step through every command executed in progression with the knowledge of what got passed into all of them. The *Messages tab*, which shows the subset of commands which are just incoming or outgoing messages, also provides the *Command Details* section as well. Figure 4.1 shows The Client displaying the parameters and return value of an *MPI_Send()* command using the *Command Details* panel.
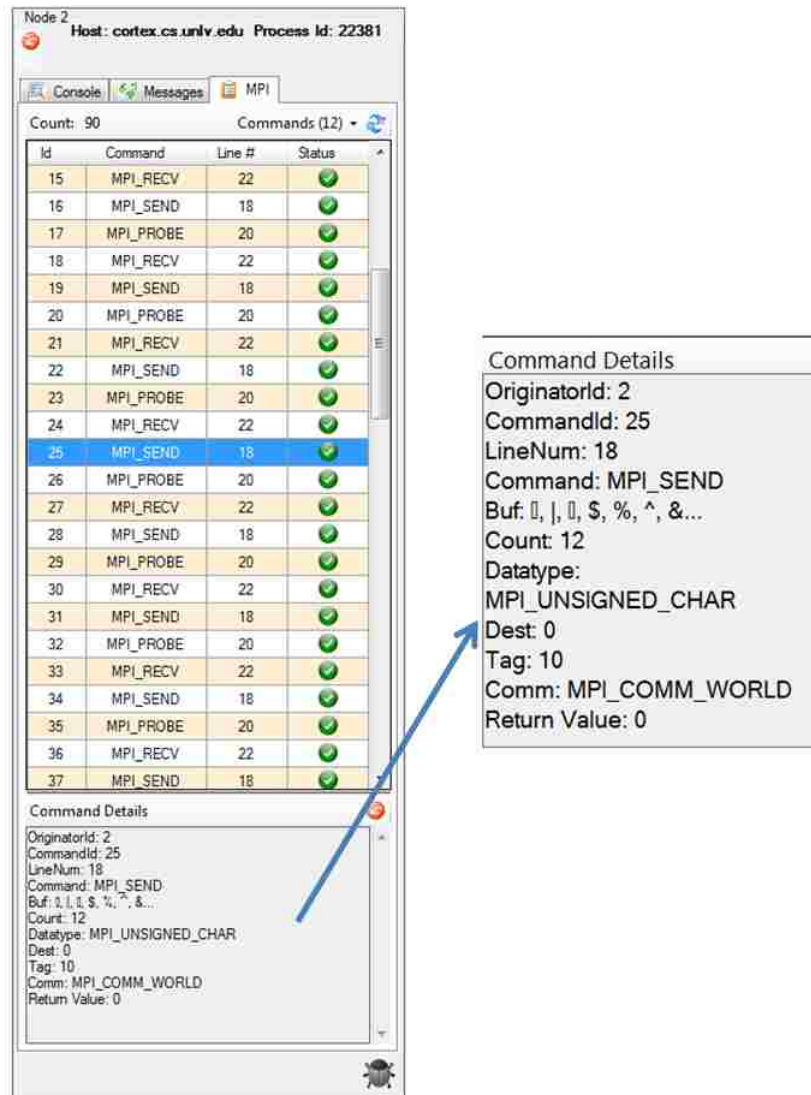
Figure 4.1: The extra information displayed in the Command Details panel of each node.

Matching Messages

Under the *MPI tab*, the user can see that every command is given a unique command id within each node. This identifier is issued by The Runtime and sent back within *PRE* and *POST* commands to give The Client a key to use when requesting The Call Center to retrieve buffer values from recorded sessions. Although The Client does not issue ids per command, it does issue ids per matched message pair. These matched ids are the ones displayed on the *Messages tab*. They are global ids across all nodes and are meant to uniquely link two messages together as a send/receive pair. Although a user can start with a given message command and search through each node for a corresponding message

match, the Distributed Application Debugger has a *message matching* feature which makes this much easer. When the user enables *message matching* mode by clicking the Handshake ♥ icon, the user can step through each node's messages and see their complimenting receive or send get highlighted as displayed in Figure 4.2. This becomes helpful as the users see the parameters and buffers passed between all the nodes in the cluster which can help them determine if they made a bad assumption, had an unexpected result, or just made a mistake when passing messages between nodes.
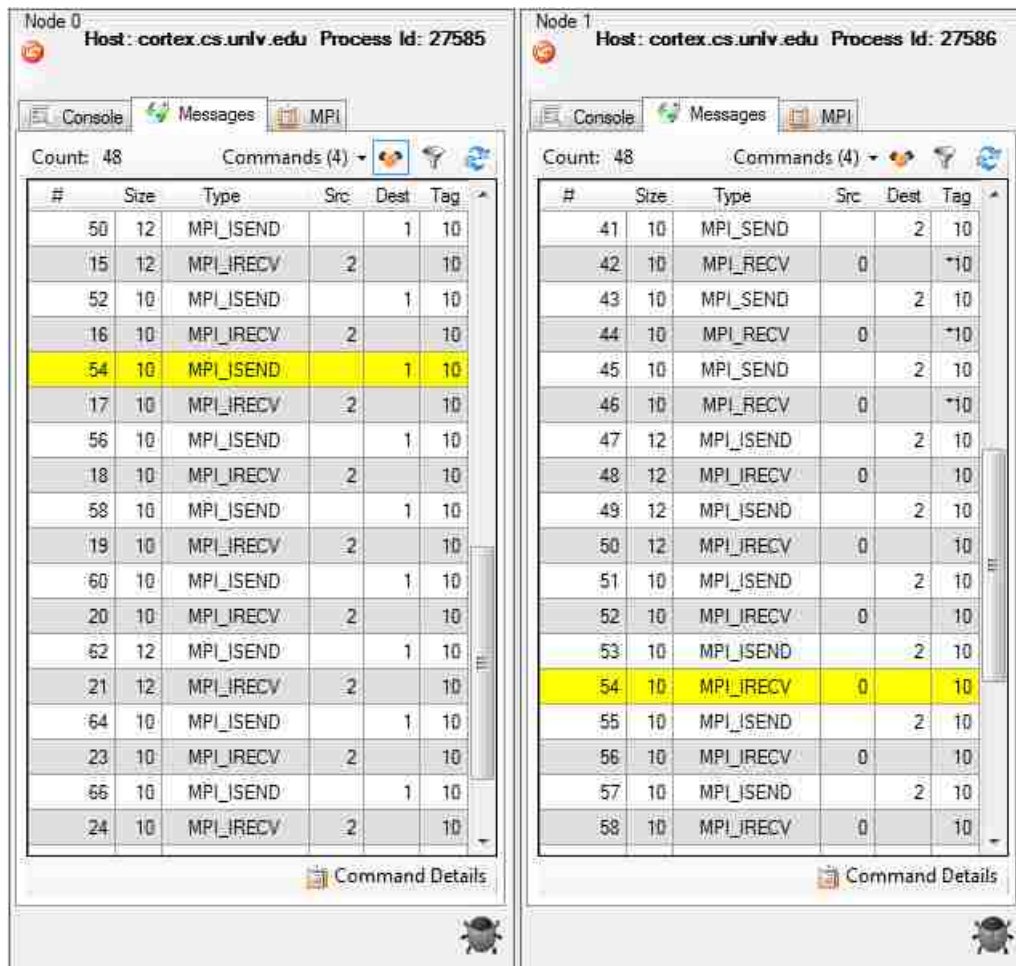
Figure 4.2: Two nodes displaying automated message matching.

Although it is helpful to be able to quickly recall and display which messages are matched as pairs, it is perhaps even more important to quickly be able to display messages which were NOT matched. An application which has unmatched messages could produce the expected outcome at times, but it is very possible that it points to a poorly engineered application which contains race conditions which may not be visible today, but may become visible sometime in the future. Regardless of whether the user suspects that their application has a bug or not, the Distributed Application Debugger will always point out mismatched messages to the user by changing the font color to red as displayed in Figure 4.3.

Figure 4.3: An MPI receive message without a matching send command.

Filters

The Distributed Application Debugger offers two differnt types of filters to cut down on the overwhelming amount of data that is present within each tab. The first type has to do with identifying any of the mismatched messages described above. The messages tab offers a filter button, ∇ which removes all matched messages from the node's messages tab. This leaves the user with just the suspect mismatched messages for the user to investigate. The second type is aimed at helping the user cut down on the number of commands displayed within the *Messages* and *MPI* tabs. It is a drop down which allows the users to simple check or uncheck the commands that they want displayed within the tab as illustrated in Figure 4.4.
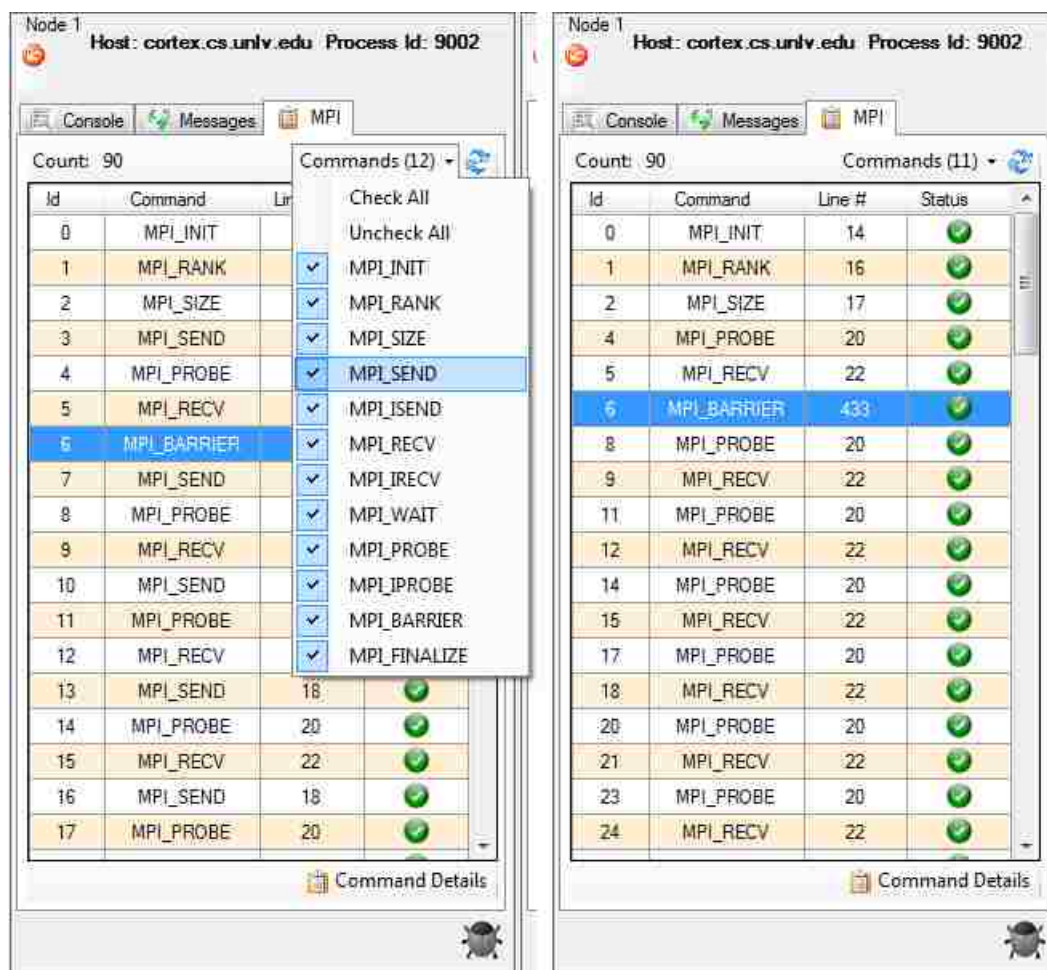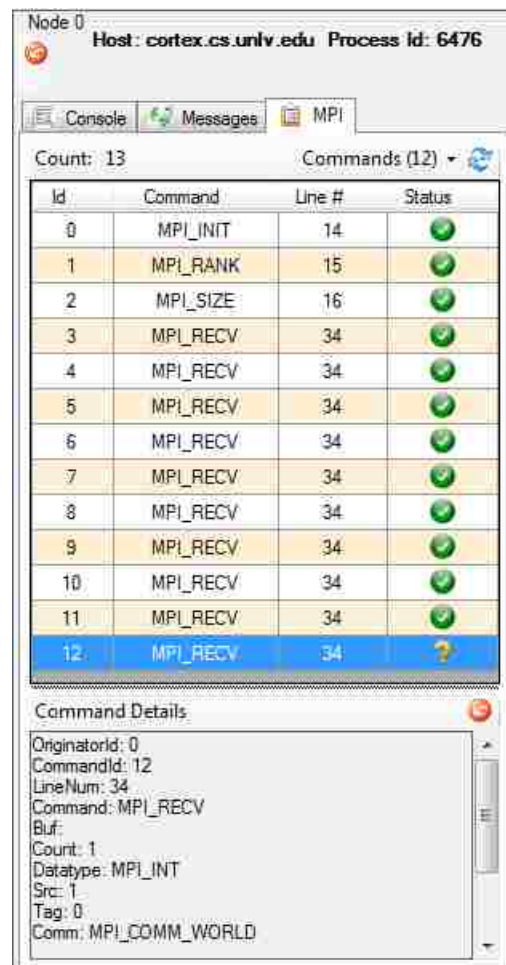
Figure 4.4: The MPI panel before and after a command filter was applied.

Command Statuses

The MPI panel displays all commands relayed back to The Client categorized with one of four statuses: Incomplete, Validation Warning, Error, and Success. Since every MPI command is represented by one *PRE* and one *POST* command, The Client may have messages which have received their *PRE* command without its corresponding *POST* command yet. These *in between* commands are categorized with a status of *Incomplete*. Although all commands start out as *Incomplete* because the *PRE* command is, of course, a separate command from the *POST* command, it should not stay Incomplete long. To the user it is unlikely that they will ever see the time between the *PRE* and *POST* command because the *POST* command, when sent, is always sent closely behind the corresponding *PRE* command.

Figure 4.5: An MPI command displayed with a status of Incomplete.

A command that is noticeably stuck in the *Incomplete* status gives the user the information that the line associated in the MPI code was executed but never finished. This implies in most cases that a command is blocked and execution of the application has effectively halted. Incomplete messages are indicated in the *MPI tab* with the 🔳 icon as displayed in Figure 4.5. While a user who is not using the Distributed Application Debugger may be left wondering why the application is hanging, the users who are will quickly see that the application is blocking on an *MPI_Recv()* which, given the list of other successful *MPI_Recv()* commands preceding it with the same line number, appears to be in a loop. Upon investigating this loop, the user may quickly realize that they are doing an extra *MPI_Recv()*, or forgetting a send, and can efficiently fix the bug and move on their way.
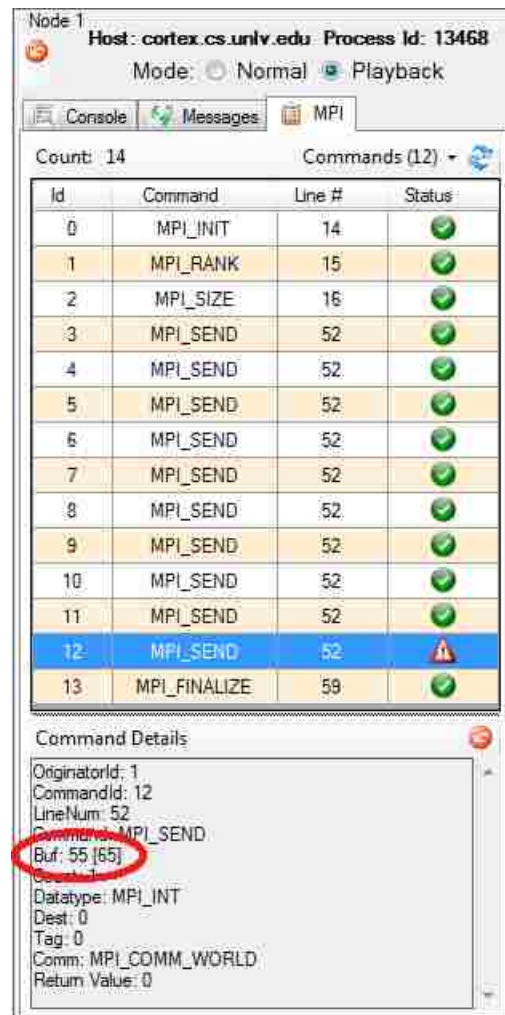
Figure 4.6: An MPI command displayed with a status of Validation Warning.
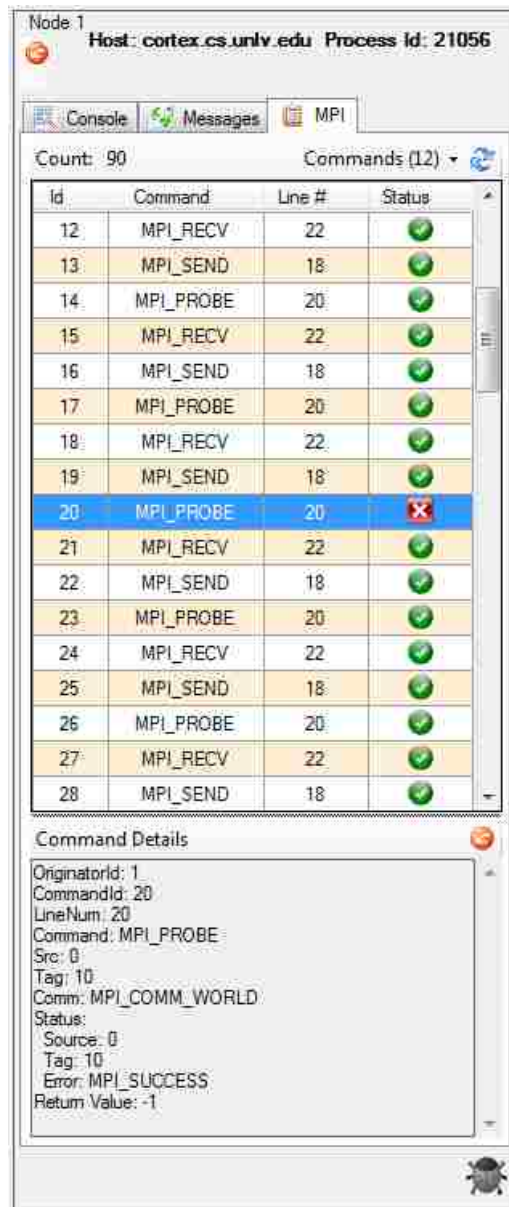
Figure 4.7: An MPI command which returned an error code from The Runtime.

In the case that the user is Replaying a session, The Runtime will be reading values from an XML file created earlier during a `RECORD` session. Because parameters are both passed into the MPI command and recorded in the XML file, The Runtime provides automatic validation of actual values vs expected values. In the case that a command fails validation and executes a command with different parameters from before, The Runtime will send back the discrepancies to The Client and a warning status indicator of ⚠ will be displayed. In order to see the specific discrepancy, or

discrepancies, the user just needs to look at the *Command Details* section of the command with the warning and look for data placed between square brackets. This will give an indication of what the expected values were. In Figure 4.6, The Runtime issued a validation warning that it had recorded a buffer with a 55 in it, but during `REPLAY` the buffer had a value of 65.

The final two statuses are error and success. They are based on the return values of each MPI command which are either zero for success or a non-zero indicating an error. In the case of an error, the icon is displayed. In the case of success, the icon is displayed. Figure 4.7 shows the MPI tab values with an error reported amongst other successes.

Chapter 5

Conclusion and Future Work

The Distributed Application Debugger has a lot of great features derived from a two year survey of graduates students reporting their difficulties in debugging their parallel programs. It works remotely from home, even when the parallel cluster is not accessible directly. It graphically displays the nodes of a cluster in a way that represents both the sequential nature of the code being executing within each process, as well as the parallel nature of the messages being passed between them. It handles thousands of messages being passed back within any given session to give valuable debugging data to the user about the MPI commands that are starting and completing and all of the data being printed to `stdout`. It offers features for recording and playing back sessions which can help programmers focus on a problem that may be hard to recreate. It offers buffer value inspection to aid in debugging common sequential bugs along with message matching to cut down on the timely message error debugging. Finally it seamlessly integrates GDB to encourage alternatives to inefficient print statements.

The most complicated part of the Distributed Application Debugger was establishing a Client that was connected to a Call Center that was connected to a Runtime. Once that was established, the number of features that could be built into the system were endless because the system has a reliable TCP communication path and a pattern for request and response pairs contained within the established message envelope pattern. Although I feel that the Distributed Application Debugger will save programmers countless hours of debugging their applications, this section is dedicated to enhancements of the system that I feel could increase its value even more.

Session History Cleanup

The Distributed Application Debugger's *RECORD* feature is very helpful and will likely be used quite a bit to help students examine MPI sessions and recall buffer values. Because the system stores a physical folder worth of data for each MPI session, there is a maintenance burden of removing these directories when they are no longer needed that is not implemented in the application at this time. The users need to manually remove subdirectories from the *Sessions* directory in order to clean up the data from sessions they no longer want to recall. I think that The Call Center could expose a *Cleanup* request command which would delete folders of recorded sessions that the user no longer wants to persist.

Mismatched datatypes

Matching send and receive message pairs between nodes is an invaluable feature that the system provides for the user. When the system is producing unexpected results and there are sends or receives that do not complete, the user is instantly clued in to an area that likely is leading to their results. The problem, of course, is what if all of your messages did match up but you still got unexpected results. Is there any more data analysis that the system could do to help the user inspect their code for the error? One such error happens with mismatched datatypes. MPI does not throw an error if the the datatypes between a send and a receive command are not of the same type. It is only concerned if they fit in the alloted memory space or not. An example of this situation would be if a node sent a message with the data type *MPI_Int* and the receive received it into an unsigned integer of datatype *MPI_Unsigned* as displayed in Figure 5.1.

In this example the user has coded for Node 1 to send 3 integers of values -100, 0, and 100 to Node 0 who will receive them in their input buffer. Since MPI allows this, the application will finish with the user being none the wiser that he or she accidentally received those values using an unsigned integer datatype instead of a signed integer. Although if the user looks closely using the buffer recall feature they will see that the number -100 was sent and the value positive 4,294,967,196 was received, it is not called out to them immediately. Also, even if they see that the values are different, the root problem, that the datatypes on the send and receive command were different, is quite subtle. I think that in the future, a feature which highlighted mixed datatypes between matched by, perhaps, highlighting the matched messages in a different color or by producing a warnings report would add value to the product. As the product is now, the user will have to use the values at their disposal now to track down the root of their problem as displayed in Figures 5.2 and 5.3.

```
1
2  #define _GNU_SOURCE
3
4  #include "mpi.h"
5
6  #include <stdio.h>
7  #include <string.h>
8  #include <stdlib.h>
9
10 int main(int argc, char *argv[]) {
11    int rank, size;
12
13    MPI_Init(&argc, &argv);
14
15    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16    MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18    int bufLengths = 3;
19    int bufSizes = bufLengths*sizeof(int);
20
21    if(rank == 0)
22    {
23      unsigned int* inBuf =  (unsigned int*)malloc(bufSizes);
24      MPI_Status sta;
25      MPI_Recv(inBuf, bufLengths, MPI_UNSIGNED, MPI_ANY_SOURCE,
26             MPI_ANY_TAG, MPI_COMM_WORLD, &sta);
27    }
28    else if(rank == 1)
29    {
30      int* outBuf = (int*)malloc(bufSizes);
31
32      outBuf[0] = -100;
33      outBuf[1] = -0;
34      outBuf[2] = 100;
35
36      MPI_Request sendRequest;
37      MPI_Isend(outBuf, bufLengths, MPI_INT, 0, 11, MPI_COMM_WORLD,
38             &sendRequest);
39    }
40
41    MPI_Finalize();
42    return 0;
43 }
```

Figure 5.1: A sample program matching mixed datatypes.

63
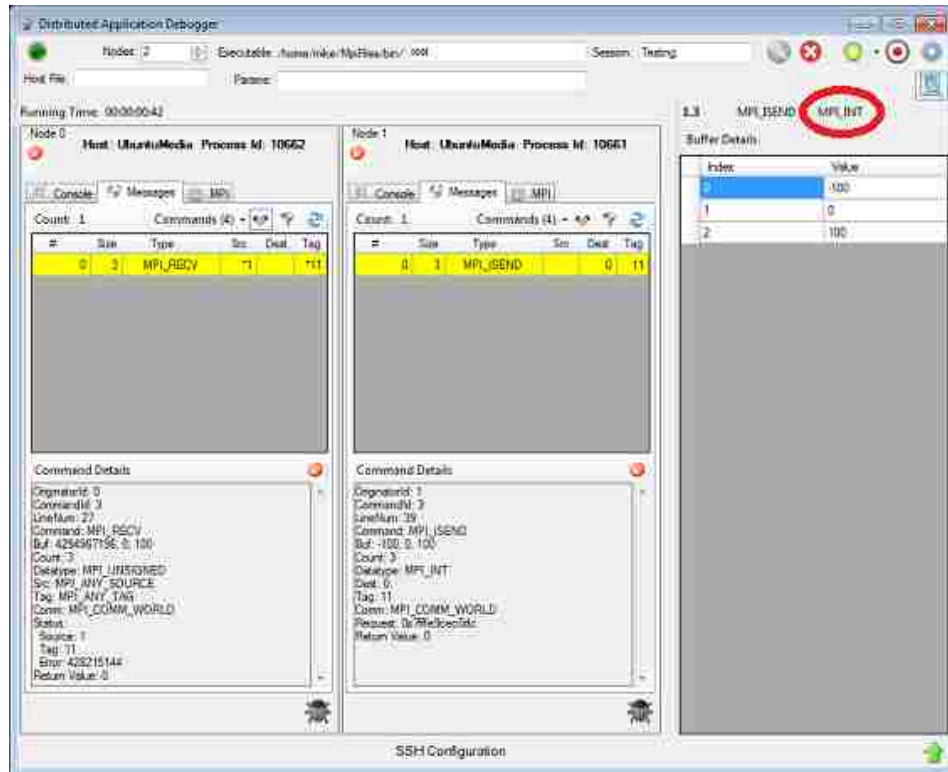
Figure 5.2: Send buffer type as *MPI_INT*.

MPI Crashes

An area that the Distributed Application Debugger does not do a very good job dealing with is the when the MPI framework crashes. This is because The Runtime is included within the application rather than actually running on top of it. When the user divides by zero, writes to unallocated memory, runs off the end of a buffer, writes to a file pointer that is closed, or does any other common mistake that causes crashes, the system is really garnered to be helpless. The Call Center will still be running, but it will not detect that The Runtime has crashed and report any of this status to the user. An example of this behavior is displayed in Figure 5.4 in an application similar to Figure 5.1 described in the emphMismatched datatypes subsection. Node 1 once again writes a buffer to Node 0 who does not receive with the same buffer type as the send again, but this time Node 1 sends a buffer of type *MPI_FLOAT* and the receive message expects a buffer of type *MPI_CHAR*.

In the case of the MPI framework crashing, The Client will keep running and waiting for feedback from The Runtime and, since it will not hear anything, The Client will also remain running. The MPI framework does report an error to `stderror`, however, which would be displayed had the user run from the command line as displayed in Figure 5.5.
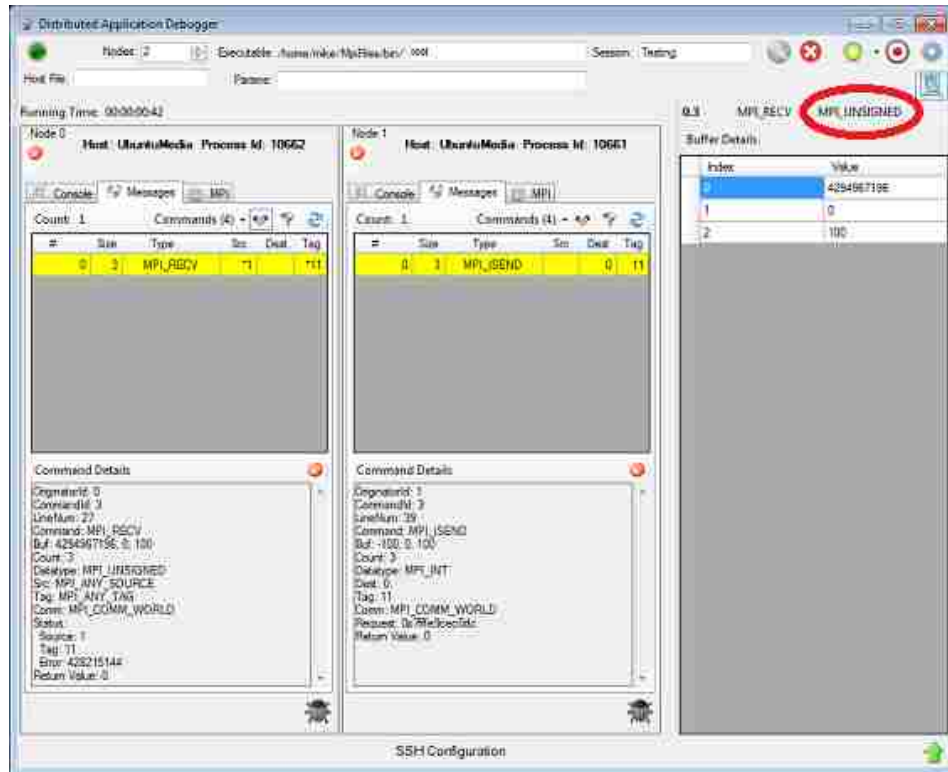
Figure 5.3: Receive buffer type as `MPI_UNSIGNED`.

This error clearly indicates that the receiving buffer is too small for the buffer being received, but, as it works today, the Distributed Application Debugger does not have the ability to convey this back to the user. I would like to see The Call Center attempt to read from `stderror` and/or issue a heartbeat command to The Runtime to test for connection status periodically and, upon detecting that MPI has crashed, be able to report this back to The Client, cleanup any processes still running, and reset itself to be able to start a new session.

```
1
2  ⊟#define _GNU_SOURCE
3
4  #include "mpi.h"
5
6  #include <stdio.h>
7  #include <string.h>
8  #include <stdlib.h>
9
10 ⊟int main(int argc, char *argv[]) {
11     int rank, size;
12
13     MPI_Init(&argc, &argv);
14
15     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16     MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18     int bufLengths = 3;
19     int bufSizes = bufLengths*sizeof(float);
20
21     if(rank == 0)
22     {
23       char* inBuf =  (char*)malloc(bufSizes);
24       MPI_Status sta;
25
26       MPI_Recv(inBuf, bufLengths, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
27               MPI_COMM_WORLD, &sta);
28     }
29     else if(rank == 1)
30     {
31       float* outBuf = (float*)malloc(bufSizes);
32
33       outBuf[0] = 0.0f;
34       outBuf[1] = -45.1234f;
35       outBuf[2] = 245.3333f;
36
37       MPI_Request sendRequest;
38       MPI_Isend(outBuf, bufLengths, MPI_FLOAT, 0, 11, MPI_COMM_WORLD,
39               &sendRequest);
40     }
41
42     MPI_Finalize();
43     return 0;
44 }
```

Figure 5.4: A sample program with mixed datatypes that will crash MPI.

Integrated Development Environment

I think that the biggest improvement, or next progression would be to evolve the Distributed Application Debugger into a 'complete solution for both development and debugging'. Integrated Development Environments (IDE), like Eclipse [Ecl13] or Microsoft Visual Studio [MSN13] have become very popular with developers as a place that they feel comfortable both developing and debugging their programs in. In the current system, the user is expected to develop their code as they would normally do so, and to attach the debugger when they encounter bugs and need more data to resolve them. Because the system has setup a persistent SSH session all the way to the cluster, I

Figure 5.5: The output sent to the command line.

would like to see a command line built into the application where users could add, update and delete file and directory names. I think it would good if it could keep track of project files, even interface with popular source control software such as Subversion [Sub13] in order to extend its contributions to the user's experience even farther.

Appendix A

Supporting Libraries and Prototypes

In order to mange the data stored and manipulated within the Distributed Application Debugger, several data structures had to be created. In order to promote maintainability and reuse, these structures were encapsulated within there own files and tested individually before incorporating them into the Distributed Application Debugger. This appendix deals with a number of supporting libraries and prototypes which contributed to the overall structure of the application and could be reused and referenced in non-MPI specific code in the future.

A.1   charList

The *charList* is a data structure which contains a list of characters along with the a count of the size of the allocated memory and the actual number of characters contained within it. It provides methods for initializing and disposing the list, manipulating its contents by adding, removing, and replacing chars, clearing the list completely, as well as populating from a file. Each of the supporting methods encapsulates dynamically sizing the list so that the user can just concentrate on the characters within it. The *charList* souce code is contained within the *collections.c* file contained in the *MpiFiles* directory mentioned in the step by step process to compiling the *MPI Runtime* in Appendix B.3.

Listing 1: charList data structure and supporting methods.

```c
typedef struct charList_item{
  char* Items;
  int ItemCount;
  int ListSize;
} charList;

void InitializeCharList(charList* newCharList)
{
  newCharList->ItemCount = 0;
  newCharList->ListSize = STARTING_CHAR_LIST_SIZE;
  newCharList->Items = (char*)malloc(newCharList->ListSize * sizeof(char));
  memset(newCharList->Items, '\0', newCharList->ListSize);
}

void CleanUpCharList(charList* list)
{
  free(list->Items);
  free(list->Items = '\0');
  free(list);
  list = '\0';
}

void SizeCharList(charList* list, int desiredMinimumSize)
{
  int initialListSize = list->ListSize;

  //Handle the case where it is too big
  while (list->ListSize < desiredMinimumSize)
  {
    list->ListSize = 2 * list->ListSize;
  }

  //Handle the case where it is too small
  while (list->ListSize / 2 > desiredMinimumSize)
  {
    list->ListSize = list->ListSize / 2;
  }

  //Check if we changed the list size and, if so, reallocate the memory
  if (list->ListSize != initialListSize)
  {
    list->Items = (char*)realloc(list->Items, list->ListSize * sizeof(char));
  }
}

void AddChars(charList* dest, char* newChars, int count)
{
  //first make sure we are big enough to allocate this many bytes
  SizeCharList(dest, dest->ItemCount + count);

  //Append the new chars to the end of the list
  memcpy(dest->Items + dest->ItemCount, newChars, count);

  //update the size of our list
  dest->ItemCount += count;
}

void RemoveChars(char* dest, charList* source, int count)
{
  //Copy the first 'x' characters from the beginning of the list
  if (dest != NULL)
    memcpy(dest, source->Items, count);

  //Move the rest of the buffer to the beginning
  memmove(source->Items, source->Items + count, source->ListSize - count);

  //Free up the buffer if it is less than half full
  SizeCharList(source, source->ItemCount - count);

  //Record the new size of the buffer
  source->ItemCount -= count;
}

void ClearChars(charList* source)
```

```
{
  if (source->ListSize != STARTING_CHAR_LIST_SIZE)
  {
    source->ListSize = STARTING_CHAR_LIST_SIZE;
    source->Items = (char*)realloc(source->Items, source->ListSize * sizeof(char));
  }

  memset(source->Items, '\0', source->ListSize);
  source->ItemCount = 0;
}

int ReplaceChars(charList* source, char* value, char* replacement)
{
  charList* dest = (charList*)malloc(sizeof(charList));
  InitializeCharList(dest);

  int startChar = 0;
  int sectionLength = 0;
  char* startPtr = strstr(source->Items, value);
  int result = FALSE;

  while (startPtr != NULL)
  {
    result = TRUE;
    sectionLength = startPtr - (source->Items + startChar);
    AddChars(dest, source->Items + startChar, sectionLength);
    AddChars(dest, replacement, strlen(replacement));
    startChar = startChar + sectionLength + strlen(value);
    startPtr = strstr(source->Items + startChar, value);
  }

  int remainingLeft = source->Items + source->ItemCount -
                             (source->Items + startChar);

  if (remainingLeft > 0)
    AddChars(dest, source->Items + startChar, remainingLeft);

  ClearChars(source);
  AddChars(source, dest->Items, dest->ItemCount);
  CleanUpCharList(dest);

  return result;
}

void ReadChars(char* fileName, charList* resultList)
{
  FILE* fp = fopen(fileName, "r");
  int bufferLength = 255;
  char inBuff[bufferLength];

  int length = 0;

  while (!feof(fp))
  {
    memset(inBuff, '\0', bufferLength);
    length = fread(inBuff, sizeof(char), bufferLength - 1, fp);
    AddChars(resultList, inBuff, length);
  }

  fclose(fp);
}
```

A.2 queue

At several points in the application I required a classic queue to manage first-in, first-out behavior (FIFO). One example of where this was used was during the queuing of messages read in from the MPI nodes detailed in Section 3.2.2. Listing 2 details my source code for the queue used in within the application along with the methods which encapsulate initializing and disposing the structure, enqueing and dequeing nodes and iterating over the contents of the structure. The *queue* source code can be found within the same *collections.c* file mentioned in Appendix A.1 along with the *charList* code.

Listing 2: queue data structure and supporting methods.

```c
typedef struct node_item{
  void* Value;
  struct node_item* NextNode;
} node;

typedef struct queue_item{
  node* FirstNode;
  node* LastNode;
  node* IterateNode;
  int Length;
} queue;

void InitializeQueue(queue* newQueue)
{
  newQueue->FirstNode = NULL;
  newQueue->LastNode = NULL;
  newQueue->IterateNode = NULL;
  newQueue->Length = 0;
}

void CleanUpQueue(queue* queueToCleanUp)
{
  while(queueToCleanUp->FirstNode != NULL)
  {
    node* nodeToCleanUp = queueToCleanUp->FirstNode;
    queueToCleanUp->FirstNode = nodeToCleanUp->NextNode;

    CleanUpNode(nodeToCleanUp);
  }

  queueToCleanUp->FirstNode = NULL;
  queueToCleanUp->LastNode = NULL;
  queueToCleanUp->IterateNode = NULL;
  free(queueToCleanUp);
}

int IsQueueEmpty(queue* initializedQueue)
{
  if(initializedQueue->FirstNode == NULL)
    return TRUE;
  else
    return FALSE;
}

void Enqueue(queue* source, void* value)
{
  //Make a new node
  node* newNode = (node*)malloc(sizeof(node));
  newNode->Value = value;
  newNode->NextNode = NULL;

  //Point the queue's last node to the new one
```

71

```c
  if (source->LastNode != NULL)
    source->LastNode->NextNode = newNode;

  //Update the last node to the new one
  source->LastNode = newNode;
  //Check if this was the first node added
  if (source->FirstNode == NULL)
    source->FirstNode = newNode;

  source->Length++;
  //No iterating while adding or removing from the queue
  source->IterateNode = NULL;

}

void Dequeue(void** dest, queue* source)
{
  //Get the first node
  node* nodeToDequeue = source->FirstNode;

  //Get the value from the node
  if (dest == NULL)
  {
    //dellocate the space of the dequed item, its not getting returned.
    free(nodeToDequeue->Value);
  }
  else
  {
    *dest = nodeToDequeue->Value;
  }

  //Point the queue at the next node
  source->FirstNode = nodeToDequeue->NextNode;

  if (source->FirstNode == NULL)
    source->LastNode = NULL;

  //Clean up the dequeued node
  CleanUpNode(nodeToDequeue);

  source->Length--;

  //No iterating while adding or removing from the queue
  source->IterateNode = NULL;
}

void StartIterateQueue(queue* source)
{
  source->IterateNode = source->FirstNode;
}

void IterateQueue(void** dest, queue* source)
{
  if (source->IterateNode != NULL)
  {
    *dest = source->IterateNode->Value;
    source->IterateNode = source->IterateNode->NextNode;
  }
  else
  {
    *dest = NULL;
  }
}

void CleanUpNode(node* nodeToCleanUp)
{
  nodeToCleanUp->NextNode = NULL;
  free(nodeToCleanUp);
}
```

A.3 String Helpers

Along with the *charList* and *queue* described in Appendices A.1 and A.2 respectively are other useful methods contained within the file *collections.c.* Listing 3 contains the *Split* and *CleanupStringArray* methods used by the Distributed Application Debugger when inspecting and cleaning up the strings partitioned within the MPI message envelope illustrated in Figure 3.10.

Listing 3: *Split* and *cleanup* methods for characters arrays.

```c
//Splits the source on the delimiter passed in and places it in the destination
int Split(char* source, char* delimiter, char*** dest, int startChar, int endChar)
{
    //determine length of the string to split
    int strLen = endChar - startChar;
    //Trim the source down first
    char* splitStr = (char*)malloc(strLen * sizeof(char));
    splitStr = memcpy(splitStr, source + startChar, strLen);

    //Initialize a list for the result
    int itemCount = 0;
    *dest = (char**)malloc(sizeof(char*));
    int sectionStartChar = 0;
    int sectionLength = 0;

    //Get location of the first delimiter
    char* startPtr = strstr(splitStr, delimiter);

    //Loop through until we don't find the delimiter anymore
    while (startPtr != NULL)
    {
        //We found one add one to the length of the list
        itemCount++;
        *dest = (char**)realloc(*dest, itemCount * sizeof(char*));

        //Get the length of this string and allocate a string to hold the value
        sectionLength = startPtr - (splitStr + sectionStartChar);

        char* value = NULL;
        if (sectionLength > 0)
        {
            value = (char*)malloc((sectionLength + 1) * sizeof(char));
            //Copy the section to the result string and add it to the list
            value = memcpy(value, splitStr + sectionStartChar, sectionLength);
        }
        else
        {
            //This must be running delimiters, make a null item
            value = (char*)malloc(sizeof(char));
            value[0] = '\0';
        }

        value[sectionLength] = '\0';

        (*dest)[itemCount - 1] = value;

        //Move the section start forward past the delimiter and look for the next instance
        sectionStartChar = sectionStartChar + sectionLength + strlen(delimiter);
        startPtr = strstr(splitStr + sectionStartChar, delimiter);

    }

    //Get any remaining characters in the string after the last delimiter
    int remainingLeft = splitStr + strLen - (splitStr + sectionStartChar);

    //Check if there were an remaining characters
    if (remainingLeft > 0)
    {
        //Add another space to the list
        itemCount++;
```

```
    *dest = (char**)realloc(*dest, itemCount * sizeof(char*));

    //Allocate a new string for the remaining characters
    char* value = (char*)malloc((remainingLeft + 1) * sizeof(char));

    if (remainingLeft == 1)
      value[0] = '\0';
    else
      //Add the last part to the list
      value = memcpy(value, splitStr + sectionStartChar, remainingLeft);

    value[remainingLeft] = '\0';
    (*dest)[itemCount - 1] = value;
  }

  //clean up
  free(splitStr);

  return itemCount;
}

void CleanupStringArray(char*** source, int itemCount)
{
  int i = 0;
  for (i = 0; i < itemCount; i++)
  {
    (*source)[i] = NULL;
    free((*source)[i]);
  }

  *source = NULL;
  free(*source);
}
```

A.4   clusterNode

The *clusterNode* data structure contains data for managing the nodes of The Runtime in a thread
safe way, as described in The Call Center's Message Routing Section 3.2.2. Listing 4 details the
source code of the structure, found in *callCenter.c*, which is used to manage communication and
concurrency between The Call Center and a node from the cluster. Also included are the methods
used to initialize and dispose of the structure.

Listing 4: clusterNode data structure and supporting methods.

```
typedef struct clusterNode_item{
  sem_t clusterNodeLock;
  sem_t messageNotification;
  int nodeId;
  int processId;
  int clientSocket;
  int gdbSocket;
  queue* messages;
} clusterNode;

void InitializeClusterNode(clusterNode* newClusterNode, int clientSocket, int nodeId, int
    processId)
{
  sem_init(&(newClusterNode->clusterNodeLock), 0, 1);
  sem_init(&(newClusterNode->messageNotification), 0, 0);

  newClusterNode->clientSocket = clientSocket;
  newClusterNode->nodeId = nodeId;
  newClusterNode->processId = processId;
  newClusterNode->gdbSocket = FALSE;
```

```
    newClusterNode−>messages = ( queue ∗) malloc ( sizeof ( queue ) ) ;

    InitializeQueue ( newClusterNode−>messages ) ;
}

void CleanupClusterNode ( clusterNode∗ disposingClusterNode )
{
    //clean up the connection created.
    close ( disposingClusterNode−>clientSocket ) ;
    close ( disposingClusterNode−>gdbSocket ) ;
    free ( disposingClusterNode−>messages ) ;
    disposingClusterNode−>messages = NULL;
    free ( disposingClusterNode ) ;
    disposingClusterNode = NULL;
}
```

A.5 XML Library

Section 2.4 details the high priority features which were considered risky. Within the features identified as the riskiest was ability to record MPI sessions to file and then be able to replay them back to the user. In order to save off each of the commands along with their parameters and return values in a readable and easily parsable format, XML was decided to be used as the protocol for serializing each of the commands. Listings 5-11 contain the XML library I wrote for formatting, parsing, and storing the XML needed within the Distributed Application Debugger.

A.5.1 xml.h

In order to use the full XML library included in the *XML* directory contained within the *MpiFiles* directory mentioned in Appendix B.3, a C file needs to include the *xmlDoc*, *xmlReader*, and *xmlWriter* header files included in Listings 6, 8, and 10. The file *xml.h* included in Listing 5 was used as a convenient header file to reference when intending to include the entire XML library.

Listing 5: The common `xml.h` header file.

```
#ifndef XML_H_INCLUDED
#define XML_H_INCLUDED

#include "xmlDoc.h"
#include "xmlReader.h"
#include "xmlWriter.h"

#endif
```

A.5.2 xmlDoc

The core XML structure code is contained within *xmlDoc.h* and *xmlDoc.c*. Listing 6 displays the source code used for the XML and the Attributes structures used to parse the values stored within XML elements. Listing 7 presents the library of methods created in order to create, parse, and dispose of the XML structures.

Listing 6: The XML structures stored in the `xmlDoc.h` file.

```c
#ifndef XML_DOC_H_INCLUDED
#define XML_DOC_H_INCLUDED

typedef struct Attribute_item{
  char* Name;
  char* Value;
} Attribute;

typedef struct XMLNode_item{
  char* Name;
  char* NodeType;
  char* Value;
  int ChildrenCount;
  int ChildrenArraySize;
  int AttributesCount;
  int AttributesArraySize;
  struct XMLNode_item **ChildNodes;
  struct Attribute_item **Attributes;
} XMLNode;

#define TRUE 1
#define FALSE 0

#define DOCUMENT_NODE "Document"
#define DECLERATION_NODE "XmlDecleration"
#define ELEMENT_NODE "Element"
#define TEXT_NODE "Text"
#define TEXT_NAME "#text"
#define DOCUMENT_NAME "Main_Document"

XMLNode* xmlCreateNode(char* nodeType, char* name, char* value);
XMLNode* createStringElementNode(char* elementName, char* nodeValue);
XMLNode* createIntElementNode(char * elementName, int nodeValue);
XMLNode* xmlAddChildNode(XMLNode *parentNode, XMLNode *childNode);
XMLNode* xmlGetChildNode(XMLNode *parentNode, char* childName);
Attribute* xmlGetAttribute(XMLNode *node, char* attributeName);

char* xmlGetText(XMLNode* node);
void xmlAddAttribute(XMLNode *xmlNode, char *atrribute, char *value);
void xmlFree(XMLNode *node);

#endif
```

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xmlDoc.h"


//Copies the string from the value into the value for the field
void assignString(char** field, char* value)
{
  if(value == NULL)
  {
    *field = NULL;
  }
  else
  {
    *field= (char*)malloc((strlen(value) + 1)*sizeof(char));
    memmove(*field, value, strlen(value));
    (*field)[strlen(value)] = '\0';
  }
}

//Gets the first child of the node passed in as text
char* xmlGetText(XMLNode *node)
{
  return node->ChildNodes[0]->Value;
}

//gets the attribute of a node with the name passed in
Attribute* xmlGetAttribute(XMLNode *node, char* attributeName)
{
  int i = 0;
  Attribute* attribute = NULL;
  for(i = 0; i < node->AttributesCount; i++)
  {
    if(strcmp(node->Attributes[i]->Name, attributeName) == 0)
    {
      attribute = node->Attributes[i];
      break;
    }
  }

  return attribute;
}

//Creates an empty new XML Node and returns it
XMLNode* createEmptyXMLNode()
{
  XMLNode* newNode = (XMLNode *)malloc(sizeof(XMLNode));
  newNode->ChildrenCount = 0;
  newNode->ChildrenArraySize = 1;
  newNode->ChildNodes = (XMLNode **)malloc(sizeof(XMLNode*));
  newNode->AttributesCount = 0;
  newNode->AttributesArraySize = 1;
  newNode->Attributes = (Attribute **)malloc(sizeof(Attribute*));

  //Set pointers to null by default?
  newNode->NodeType = '\0';
  newNode->Name = '\0';
  newNode->Value = '\0';

  return newNode;
}

XMLNode* createStringElementNode(char* elementName, char* nodeValue)
{
  XMLNode *elementNode = xmlCreateNode(ELEMENT_NODE, elementName, NULL);
  xmlAddChildNode(elementNode, xmlCreateNode(TEXT_NODE, TEXT_NAME, nodeValue));

  return elementNode;
}

XMLNode* createIntElementNode(char * elementName, int nodeValue)
{
```

```
    XMLNode *elementNode = xmlCreateNode(ELEMENT_NODE, elementName, NULL);

    char *value;
    int length = 0;
    length = asprintf(&value, "%d", nodeValue);

    xmlAddChildNode(elementNode, xmlCreateNode(TEXT_NODE, TEXT_NAME, value));
    free(value);

    return elementNode;
}

//Create an XML Node with the type, name, and value assigned
XMLNode* xmlCreateNode(char* nodeType, char* name, char* value)
{
    XMLNode* newNode = createEmptyXMLNode();
    assignString(&(newNode->NodeType), nodeType);
    assignString(&(newNode->Name), name);
    assignString(&(newNode->Value), value);

    return newNode;
}

char* GetText(XMLNode* node)
{
    return node->ChildNodes[0]->Value;
}

//Adds an attribute to the node passed in with the corresponding attribute/value pair
void xmlAddAttribute(XMLNode *xmlNode, char *atrribute, char *value)
{
    if(xmlNode->AttributesCount == xmlNode->AttributesArraySize)
    {
        xmlNode->AttributesArraySize = 2 * xmlNode->AttributesArraySize;
        xmlNode->Attributes = realloc(
        xmlNode->Attributes, xmlNode->AttributesArraySize * sizeof(Attribute*));
    }

    xmlNode->AttributesCount++;
    xmlNode->Attributes[xmlNode->AttributesCount - 1] = (Attribute *)malloc(sizeof(Attribute
        ));

    assignString(&(xmlNode->Attributes[xmlNode->AttributesCount - 1]->Name), atrribute);

    int valueLength = (int)strlen(value);
    if(value[valueLength - 1] == '\n')
        value[valueLength - 1] = ' ';

    assignString(&(xmlNode->Attributes[xmlNode->AttributesCount - 1]->Value), value);
}

//Adds a child node to the parent node passed in
XMLNode *xmlAddChildNode(XMLNode *parentNode, XMLNode *childNode)
{
    if(parentNode->ChildrenCount == parentNode->ChildrenArraySize)
    {
        parentNode->ChildrenArraySize = parentNode->ChildrenArraySize * 2;
        parentNode->ChildNodes =
        realloc(parentNode->ChildNodes, parentNode->ChildrenArraySize * sizeof(XMLNode*));
    }

    parentNode->ChildrenCount++;
    int currentChildIndex = parentNode->ChildrenCount - 1;

    parentNode->ChildNodes[currentChildIndex] = childNode;

    return childNode;
}

XMLNode* xmlGetChildNode(XMLNode *parentNode, char* childName)
{

    XMLNode *returnNode = NULL;
    int i = 0;
    for(i = 0; i < parentNode->ChildrenCount; i++)
    {
```

```
      if(strcmp(parentNode->ChildNodes[i]->Name, childName) == 0)
      {
        returnNode = parentNode->ChildNodes[i];
        break;
      }
  }
  return returnNode;
}

void freeAttribute(Attribute *attribute)
{
  free(attribute->Name);
  free(attribute->Value);
}

void xmlFree(XMLNode *node)
{
  free(node->Name);
  free(node->NodeType);
  free(node->Value);

  int i = 0;
  for(i=0; i < node->AttributesCount; i++)
  {
    freeAttribute(node->Attributes[i]);
  }

  i = 0;
  for(i=0; i < node->ChildrenCount; i++)
  {
    xmlFree(node->ChildNodes[i]);
  }

  free(node->ChildNodes);
  free(node->Attributes);

  free(node);
}
```

A.5.3   xmlWriter

The methods used to record XML are contained within *xmlWriter.h* and *xmlWriter.c*. Listing 8 displays the source code of *xmlWriter.h* which can be included in order to reference the writing portion of the XML library. Listing 9 presents the actual implementation of the methods used to write XML to file.

Listing 8: The methods exposed by the `xmlWriter.h` file.

```
#ifndef XML_WRITER_H_INCLUDED
#define XML_WRITER_H_INCLUDED
#include "xmlDoc.h"
void xmlPrint(XMLNode *node);
void xmlWrite(XMLNode *node, FILE *outputFile);
#endif
```

Listing 9: The methods available to write XML to a FILE pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <time.h>
#include "xmlWriter.h"

//Recursively xmlPrints the values of the node passed in with
//a margin based on the level passed in
void xmlPrintXMLHelper(XMLNode *node, int level, FILE *outputFile)
{
  int i = 0;
  char margin[100] = "\0";
  while(i < level * 2)
  {
    margin[i] = '_';
    i++;
  }

  margin[level * 2] = '\0';


  if(strcmp(node->NodeType, DOCUMENT_NODE) == 0)
  {
    int j =0;
    while(j < node->ChildrenCount)
    {
      xmlPrintXMLHelper(node->ChildNodes[j], 0, outputFile);
      j++;
    }
  }
  else if(strcmp(node->NodeType,DECLERATION_NODE) == 0)
  {
    fprintf(outputFile, "%s<?%s_%s_?>\n",margin, node->Name, node->Value);
  }
  else if(strcmp(node->NodeType,TEXT_NODE) == 0)
  {
    fprintf(outputFile, "%s", node->Value);
  }
  else if(strcmp(node->NodeType,ELEMENT_NODE) == 0)
  {
    fprintf(outputFile, "%s<%s", margin, node->Name);
    i = 0;
    while(i < node->AttributesCount)
    {
      fprintf(outputFile, "_%s=\"%s\"", node->Attributes[i]->Name, node->Attributes[i]->
          Value);
      i++;
    }
    fprintf(outputFile, ">");
```

```
    if(node->ChildrenCount > 0 && strcmp(node->ChildNodes[0]->NodeType,ELEMENT_NODE) == 0)
    {
      fprintf(outputFile, "\n");
    }
    else
    {
      margin[0] = '\0';
    }

    i=0;
    while(i < node->ChildrenCount)
    {
      xmlPrintXMLHelper(node->ChildNodes[i], level+1, outputFile);
      i++;
    }

    fprintf(outputFile, "%s</%s>\n",margin, node->Name);
  }
}

//Recursivley xmlPrints ths XML node's values to the screen
void xmlPrint(XMLNode *node)
{
  xmlPrintXMLHelper(node, 0, stdout);
}

//Recursivley xmlPrints ths XML node's values to file
void xmlWrite(XMLNode *node, FILE *outputFile)
{
  xmlPrintXMLHelper(node, 0, outputFile);
  fprintf(outputFile, "\n");
}
```

A.5.4 xmlReader

The methods used to replay XML are contained within *xmlReader.h* and *xmlReader.c*. Listing 10 displays the source code of *xmlReader.h* which can be included in order to reference the reading portion of the XML library. Listing 10 presents the actual implementation of the methods used to read XML from file and build XML structures in memory to traverse the elements and attributes.

Listing 10: The methods exposed by the `xmlReader.h` file.

```
#ifndef XML_READER_H_INCLUDED
#define XML_READER_H_INCLUDED
#include "xmlDoc.h"
XMLNode *xmlRead(char* file);
#endif
```

Listing 11: The methods to read XML files and recreate XML structures.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>
#include <time.h>
#include "xmlReader.h"
#include "xmlWriter.h"

//The xml array from file held in memory
char *xmlArray;
//The current index we are reading from the xml array
int chrPtr = 0;

char* myFileName;

//Load the xml from the file passed in into the xml array in memory
void loadXMLFile(char* fileName)
{
   //Open the file and determine its length
   FILE *xmlFile = fopen(fileName, "r");
   fseek(xmlFile, 0, SEEK_END);
   long fileSize = ftell(xmlFile);
   rewind(xmlFile);

   //instantiate and populate the xml array
   xmlArray = (char*)malloc(fileSize *sizeof(char));
   int length = 0;
   length = fread(xmlArray, 1, fileSize, xmlFile);

   //close the file
   fclose(xmlFile);
}

//reads back from the length of the text to the current index
//and returns it as a string
char *readFromXML(int length)
{
   //Create a new string to hold the value int
   char *field = (char*)malloc((length + 1)* sizeof(char));
   //Copy the xml array starting from 'x' characters back until the current char
   strncpy(field, (xmlArray + chrPtr) - length, length);
   //Delimiate the last character so it terminates
   field[length] = '\0';

   //Return the read field
   return field;
}

//moves the current character pointer until it finds a non space, tab or newline character
void clearWhiteSpace()
{
```

```
  while((xmlArray[chrPtr] == '␣' || xmlArray[chrPtr] == '\t' || xmlArray[chrPtr] == '\n')
      && xmlArray[chrPtr] != '\0')
  {
    chrPtr++;
  }
}


//moves the current character pointer until it finds the character passed in or gets to
    the end of the array
void moveToChar(char target)
{
  while(xmlArray[chrPtr] != target && xmlArray[chrPtr] != '\0')
  {
    chrPtr++;
  }
}

//Returns a string of all of the characters until the next space or '>' character
char* getName()
{
    //First move through all the white space
  clearWhiteSpace();

  //Loop until the next space or a closing tag
  int nameLength = 0;
  while(xmlArray[chrPtr] != '>' && xmlArray[chrPtr] != '␣')
  {
    nameLength++;
    chrPtr++;
  }

  //Read the characters from the array
  char* returnValue = readFromXML(nameLength);

  //Return the new string
  return returnValue;
}

//Gets the key/value pairs of the atributes until it
//gets to the end of the encapsulating tag.
void getAttributes(XMLNode *xmlNode)
{
  //Clear the white space
  clearWhiteSpace();

  //Loop until we get to the end of the current tag we are in
  while(xmlArray[chrPtr] != '>')
  {
    //The attributes end with an equals sign and can not contain spaces
    int textLength = 0;
    while(xmlArray[chrPtr] != '␣' && xmlArray[chrPtr] != '=')
    {
      chrPtr++;
      textLength++;
    }

    //read the attribute
    char* attribute = readFromXML(textLength);

    //The value starts after a quotation mark after the equals sign
    moveToChar('=');
    moveToChar('"');
    chrPtr++;

    //Loop until we get to the end of the quotation mark
    textLength =0;
    while(xmlArray[chrPtr] != '"')
    {
      chrPtr++;
      textLength++;
    }

    //read the value
    char * value = readFromXML(textLength);
```

```c
        //Add the attribute/value pair to the attributes of the node
        xmlAddAttribute(xmlNode, attribute, value);

        //clean up the buffers
        free(attribute);
        free(value);

        //Continue moving on
        chrPtr++;
        clearWhiteSpace();
    }
}

//Gets the value of an XML declaration node
char* getDeclerationValue()
{
    //XML decleartion is the string between the question makr signs of an decleration node
    clearWhiteSpace();
    int length = 0;
    int insideQuotation = FALSE;

    //Loop until we finish reading outside of the quotes or we reach the question mark
    while((xmlArray[chrPtr] != ' ' && xmlArray[chrPtr] != '?') || insideQuotation == TRUE)
    {
        length++;
        chrPtr++;

        //Everything inside of quotation marks is part of the decleration
        if(xmlArray[chrPtr] == '"')
        {
            if(insideQuotation == TRUE)
            {
                insideQuotation = FALSE;
            }
            else
            {
                insideQuotation  = TRUE;
            }
        }
    }

    //read the decleration
    char *declerationvalue = readFromXML(length);

    //return the decleration that was read
    return declerationvalue;
}

//Get everything until the next node begins
char* getTextValue()
{
    //clearWhiteSpace();
    int textLength = 0;

    //Loop until the next tag starts
    while(xmlArray[chrPtr] != '<')
    {
        textLength++;
        chrPtr++;
    }

    //Read the text
    char *textValue = readFromXML(textLength);

    //Return the text that was read in
    return textValue;
}

//Creates a new decleration node based on the decleration it reads
XMLNode *readXMLDeclerationNode()
{
    char* name = getName();
    char* value = getDeclerationValue();
    XMLNode* newNode = xmlCreateNode(DECLERATION_NODE, name, value);

    free(name);
```

```
      free ( value ) ;
      return newNode ;
}

// Creates a new element node based on the name it reads
XMLNode *readXMLElementNode ( )
{
   char* name = getName ( ) ;
   XMLNode* newNode = xmlCreateNode (ELEMENT_NODE, name, NULL) ;
   free (name) ;

   return newNode ;
}

// Creates a new text node based on the text that it reads
XMLNode *readXMLTextNode ( )
{
   char* textValue = getTextValue ( ) ;
   XMLNode* newNode = xmlCreateNode (TEXT_NODE, TEXT_NAME, textValue ) ;
   free ( textValue ) ;

   return newNode ;
}

// Passes through the xml array and adds children to currently passed in node
void parseNode (XMLNode *node )
{
   //Save off the previous chr ptr in case white spaces should be part of the text value
   int prevChrPtr = chrPtr ;

   clearWhiteSpace ( ) ;

   //Loop until the end of the file
   while (xmlArray [ chrPtr ] != '\0'){
      //Check if war are at the beginning of a tag
      if (xmlArray [ chrPtr ] == '<' && xmlArray [ chrPtr + 1] != '/'){
         //Check if we are reading an xml decleration node
         chrPtr++;
         if (xmlArray [ chrPtr ] == '?'){
            //This muse be an xml decleration monde. Move passed the
            //question mark just read in and get the decleration part.
            chrPtr++;
            xmlAddChildNode ( node , readXMLDeclerationNode ( ) ) ;

            //Move to the end tab
            moveToChar ( '>') ;

            //Move past the end tab and fall off
            chrPtr++;
            clearWhiteSpace ( ) ;
         }
         else {
            //This must be a regular element node. Read the element node
            XMLNode* childNode = xmlAddChildNode ( node , readXMLElementNode ( ) ) ;

            //Get the attributes for the element
            getAttributes ( childNode ) ;
            //Move to the closing tab
            moveToChar ( '>') ;
            chrPtr++;

            //Recursivley call up and see if the current node has any children
            parseNode ( childNode ) ;

            //Now that we have all of its children, move past our closing tag.
            clearWhiteSpace ( ) ;

            if (xmlArray [ chrPtr ] == '<'){
               moveToChar ( '>') ;

               if (xmlArray [ chrPtr ] == '>'){
                  chrPtr++;
               }
            }

            //Now are are passed our closing tag. See if we were are not at our
```

```
                 //parents closing tag and, if so, fall off.
                 clearWhiteSpace();

                 if(xmlArray[chrPtr] == '<' && xmlArray[chrPtr + 1] == '/')
                   break;

                 //We were not the last sub element of our parent.  Loop again and add the next one
                       as a child.
            }
      }
      else{
         //move back to the beginning of this section
         chrPtr = prevChrPtr;
         //printf("About to make text node, next 3 char are '%c%c%c'", xmlArray[chrPtr],
               xmlArray[chrPtr + 1], xmlArray[chrPtr + 2]);
         //We are not at the beginning of a tag, so we must be in the text
         //the node.  Read its text and fall off.
         xmlAddChildNode(node, readXMLTextNode());
         break;
      }
   }
}

//Reads in the file passed in and returns it as an XML node
XMLNode *xmlRead(char* file)
{
   chrPtr = 0;
   loadXMLFile(file);
   XMLNode *docNode = xmlCreateNode(DOCUMENT_NODE, DOCUMENT_NAME, NULL);

   parseNode(docNode);

   free(xmlArray);

   return docNode;
}
#endif
```

A.6 The GDB Bridge

During the earliest stages of the development of the Distributed Application Debugger, I did prototyping of some of the high priority features in order to prove that the features were feasible. The most important of these was the GDB Bridge included in Listing12. It was created in order to experiment with launching GDB from within a running program and then controlling GDB by duplicating its **stdin**, **stdout**, and **stderr** file descriptors.

After successfully creating a bridge to launch and control GDB from the command line, the code was enhanced to be controlled by commands read in from a TCP port. Once this code was worked out as a proof of concept, the Distributed Application Debugger specific development began, knowing that remote GDB features could confidently be integrated into later.

Listing 12: The contents of the GDB Bridge prototype.

```c
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>

//Define some constants for read ability
#define TRUE 1
#define FALSE 0
#define QUIT_COMMAND "quit"
#define GDB_COMMAND "gdb"
#define STRING_EQUALS_INDICATOR 0
#define NEWLINE "\n"
#define SPACE " "

//Max size of read buffer
#define BUFFER_SIZE 1024

//Listen and Write socket
int list_s;
int conn_s;

//Default Port
int port = 4001;

//Streams representing the parent and
//child's perspective out incoming and outgoing
FILE* streamOutgoing;
FILE* streamIncoming;

//Methods to read from and to a socket
ssize_t Readline(int sockd, void *vptr, size_t maxlen);
ssize_t Writeline(int sockd, const void *vptr, size_t n);

//Methods which will listen to the user and the child on a seperate thread
void childListener();
void tcpListen();

//A sempahore which gets signaled to allow the main thread to exit
sem_t quitingSemaphore;

int main (int argc, const char* argv[])
```

```
{
  if(argc == 2)
  {
    //Read the command line
    port = atoi(argv[1]);
  }

  //A buffer for genearl reading and writing a stream
  char buffer[BUFFER_SIZE];

  //The id being checked after the fork command
  pid_t pid;

  //The ends of the pipes for communication from the child to parent
  //and parent to child
  int fromChildPipe[2];
  int fromParentPipe[2];

  /* Create a pipe. File descriptors for the two ends of the pipe are placed in fds.  */
  pipe (fromChildPipe);
  pipe (fromParentPipe);


  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0)
  {
  //Overwrite stdin, stdout, and stderror
  close(0);
  dup(fromParentPipe[0]);

  close(1);
  dup(fromChildPipe[1]);

  close(2);
  dup(fromChildPipe[1]);

  //wait for an indication of a connection
  fgets (buffer, sizeof (buffer), stdin);

  //The string sent will be the argument for the gdb command
  char *inputArg = strtok(buffer, NEWLINE);

    //create the gdb command for the test file
    char* gdbCommand[4];
    gdbCommand[0] = "gdb";
    gdbCommand[1] = inputArg;
    gdbCommand[2] = NULL;

    //Shell out the gdb command
    execvp(gdbCommand[0], gdbCommand);

  else
  {
    //Set streams for reading from and writing to the child
    streamIncoming = fdopen (fromChildPipe[0], "r");
    streamOutgoing = fdopen(fromParentPipe[1], "w");

    //Initialize the semaphore to block until signaled
    sem_init(&quitingSemaphore, 0, 0);

    //Launch threads for the child listen and the tcp listen
    pthread_t childThreadId, tcpListenThreadId;
    pthread_create(&childThreadId, NULL, (void *)childListener, NULL);
    pthread_create(&tcpListenThreadId, NULL, (void *)tcpListen, NULL);

    //Wait till the tcp listen thread tells us its time to exit
    sem_wait(&quitingSemaphore);

    //Exit the application
    exit(0);
  }

  return 0;
}
```

88

```
//Thread for listen to the tcp port
void tcpListen()
{
  struct      sockaddr_in servaddr;
  char        *endptr;

  char buffer[BUFFER_SIZE];
  char *token;

  /*  Create the listening socket  */
  if ( (list_s = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
    fprintf(stderr, "Error_creating_listening_socket.\n");
    exit(EXIT_FAILURE);
  }


  /*  Set all bytes in socket address structure to
      zero, and fill in the relevant data members  */
  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port        = htons(port);


  /*  Bind our socket addresss to the
      listening socket, and call listen()  */
  if ( bind(list_s, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 ) {
    fprintf(stderr, "Error_calling_bind()\n");
    exit(EXIT_FAILURE);
  }

  printf("Listening_on_port_%d\n", port);

  //Listen for a connection
  if ( listen(list_s, 1024) < 0 ) {
    fprintf(stderr, "Error_calling_listen()\n");
    exit(EXIT_FAILURE);
  }

    /*  Enter an infinite listen loop */
    if ( (conn_s = accept(list_s, NULL, NULL) ) < 0 ) {
      fprintf(stderr, "Error_calling_accept()\n");
      exit(EXIT_FAILURE);
    }

    //Flag which indicates that we have gotten the gdb command
    int started = FALSE;

    //Create some pointers to the commands comming in
    char* inputCommand0;
    char* inputArg;

    //Loop forever
    while ( 1 )
    {


      /*  Retrieve an input line from the connected socket
          then simply write it back to the same socket.      */
      Readline(conn_s, buffer, BUFFER_SIZE - 1);

      //Strip off the newline at the end
      token = strtok(buffer, NEWLINE);

      //Make sure we got something
      if(token == NULL)
      {
              continue;
      }

      //See if we have actually started the GDB debugger yet
      if(started == FALSE)
      {
        //We have not started the gdb debugger, check if the user typed
        //the gdb command and the file to debug
```

```
                inputCommand0 = strtok(token, SPACE);
                inputArg = strtok(NULL, SPACE);

                if(strcmp(inputCommand0, GDB_COMMAND) == STRING_EQUALS_INDICATOR &&
                        inputArg != NULL)
                {
                    //the user sent the gdb command to start, send the file to debug to the child
                    started = TRUE;
                    fprintf(streamOutgoing, "%s\n", inputArg);
                    fflush(streamOutgoing);

                    //Indicate that connection has been made
                    printf("GDB session begun for %s\n", inputArg);
                    continue;
                }
                else
                {
                    //We either didn't get the gdb command, or we didn't get a file, continue
                        looping
                    continue;
                }
            }

            //Pass the command read in to the child process
            fprintf(streamOutgoing, "%s\n", token);
            fflush(streamOutgoing);

            //Check if the user said to quit
            if(strcmp(token, QUIT_COMMAND) == STRING_EQUALS_INDICATOR)
            {
                printf("Quit detected.  Closing now.\n");

                //Time to quit, release the semaphore which will signal the main thread to exit
                sem_post(&quitingSemaphore);
                break;
            }
            else
            {
                usleep(100);
            }

        }

        /*  Close the connected socket  */
        if ( close(conn_s) < 0 ) {
            fprintf(stderr, "Error calling close()\n");
            exit(EXIT_FAILURE);
        }
}

void childListener()
{
    //Read one char at a time from child infinitly.
    char* echoBuff = (char*)malloc(sizeof(char));

    while(1)
    {
        char c;
        c = getc(streamIncoming);
        fflush(streamIncoming);

        //white each character out to the tcp port
        echoBuff[0] = c;
        Writeline(conn_s, echoBuff, 1);
        usleep(100);
    }
}

ssize_t Readline(int sockd, void *vptr, size_t maxlen) {
    ssize_t n, rc;
    char    c, *buffer;

    buffer = vptr;

    for ( n = 1; n < maxlen; n++ ) {
        if ( (rc = read(sockd, &c, 1)) == 1 ) {
```

```
        *buffer++ = c;
        if ( c == '\n' )
          break;
      }
      else if ( rc == 0 ) {
              if ( n == 1 )
          return 0;
              else
          break;
      }
      else {
        if ( errno == EINTR )
          continue;

        return -1;
      }
    }

    *buffer = 0;
    return n;
}

/*   Write a line to a socket   */
ssize_t Writeline(int sockd, const void *vptr, size_t n)
{
    size_t        nleft;
    ssize_t       nwritten;
    const char *buffer;

    buffer = vptr;
    nleft  = n;

    while ( nleft > 0 ) {
      if ( (nwritten = write(sockd, buffer, nleft)) <= 0 ){
              if ( errno == EINTR )
          nwritten = 0;
              else
          return -1;
      }
      nleft  -= nwritten;
      buffer += nwritten;
    }

    return n;
}
```

A.7   The Bridge

In order to accommodate network configuration in which The Call Center is running on a computer
which is contained within a closed private network, a supporting application called The Bridge
was developed. It supports three modes, *client*, *server*, and *bridge*, which were used in simulating
connecting strings of computers together in some of the early days of development. In all modes,
The Bridge opens two ports and reads from one and writes to the other.

In *client mode*, the user passes a -c in at the command line along with the address and port
of a computer it intends to communicate with. Upon starting up in this mode, The Bridge makes
an outgoing connection to the address and port passed in, and then begins to read in from stdin.
Anything input from the console is read and immediately written to the outgoing socket.

In *server mode*, the user passes a -s in at the command line along with a port to listen to incoming
connections on. Upon starting up in this mode, The Bridge listens on the port passed in and, upon

receiving an incoming connection, writes anything read in from it to `stdout`.

Client mode and *server mode* were used as mostly testing modes for the third, and most important, mode: *bridge mode*. When run in *bridge mode*, The Bridge takes in `-b` at the command line along with an address and port, as in *client mode*, and an incoming port, as in *server mode*. Upon starting up in *bridge mode*, The Bridge writes whatever it reads from the incoming connection's socket to the outgoing connection's socket and, likewise, writes whatever it reads from the outgoing connection's socket to the socket acquired from the incoming connection.

It can not be understated how important The Bridge code is to the Distributed Application Debugger. It was one of the first features written because, without it, The Client and Call Center would not be able to connect in order to make remote debugging possible. All three modes were used together during the prototyping stage to work out the technical details of launching a programming after making a series of SSH connections and then passing command lines from one computer's `stdin` to another computer's `stdout` via TCP connections. The source code for The Bridge is included in Listing 13.

Listing 13: The Bridge source code used to connect The Client to The Call Center.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <semaphore.h>
#include <pthread.h>
#include "Headers/communication.h"
#include "Headers/booleanLogic.h"

#define STRING_EQUALS_INDICATOR 0
#define CLIENT_MODE "-c"
#define BRIDGE_MODE "-b"
#define SERVER_MODE "-s"

//Structure to pair an fd to read from and an fd to write to
struct pipePair_item
{
  int In;
  int Out;
};
typedef struct pipePair_item pipePair;

//Helper Methods
void PipeMessages(void *value);
void RunClientMode(char* outgoingPath, int outgoingPort);
void RunServerMode(int incomingPort);
void RunBridgeMode(int incomingPort, char* outgoingPath, int outgoingPort);
void SetSocketOptions(int *socketPtr);
void InitializeSockAddr(struct sockaddr_in *address, in_addr_t path, int port);
pipePair* CreatePipePair(int in, int out);

//Semaphore to wait on to close out the application
sem_t quitingSemaphore;


int main(int argc, char**argv)
{
        //First argument is the mode -c for client, -b for bridge, -s for server
  if(strcmp(argv[1], CLIENT_MODE) == STRING_EQUALS_INDICATOR)
  {
```

```
              //Client mode - arg 2 is the port to connect to, arg 3 is the ip address
        //RunClientMode(argv[3], atoi(argv[2]));
        if(argc == 3)
        {
          //they did not include the server ipaddress, assume we want to use our own
          char* ipAddress = (char*)malloc(50*sizeof(char));
          GetPrimaryIp(ipAddress, 50) ;
          printf("running_alternate_version,_ipaddress_is_%s\n", ipAddress);
          RunClientMode(ipAddress, atoi(argv[2]));
          free(ipAddress);
        }
        else
        {
          RunClientMode(argv[3], atoi(argv[2]));
        }
      }
    else if(strcmp(argv[1], SERVER_MODE) == STRING_EQUALS_INDICATOR)
    {
          //Server mode, arg 2 is the port to listen to
      RunServerMode(atoi(argv[2]));
    }
    else if(strcmp(argv[1], BRIDGE_MODE) == STRING_EQUALS_INDICATOR)
    {
          //Bridge mode, arg 2 is the incoming port, arg 3 & 4 are the outgoing ip address
                and port
      RunBridgeMode(atoi(argv[2]), argv[3], atoi(argv[4]));
    }

    return 0;
}

/*Establishes an outgoing connection on the specified
        port and address and pipes all data from stdout to it*/
void RunClientMode(char* outgoingPath, int outgoingPort)
{
        //Create outgoing connection
        int socketOut = CreateOutgoingConnection(outgoingPath,outgoingPort);
    if(socketOut == FALSE)
        return;

    //Start a background thread to pipe messages from stdin to the outgoing port
    pthread_t threadId;
    pthread_create(&threadId, NULL, (void *)PipeMessages, CreatePipePair(fileno(stdin),
        socketOut));

        //Post for just 1 thread to release before we finish
        sem_init(&quitingSemaphore, 0, 0);

        //Wait until the piping thread releases to return
        sem_wait(&quitingSemaphore);

        //clean up the connection created.
    close(socketOut);
}

/*Establishes an incoming connection on the specified port
                and pipes all data from it to stdout*/
void RunServerMode(int incomingPort)
{
        //Create an incoming connection
        int socketIn = CreateIncomingConnection(incomingPort);
        if(socketIn == FALSE)
                return;

        //Start a background thread to pipe messages from the incoming port to stdout
    pthread_t threadId;
    pthread_create(&threadId, NULL, (void *)PipeMessages, CreatePipePair(socketIn, fileno(
        stdout)));

        //Post for just 1 thread to release before we finish
        sem_init(&quitingSemaphore, 0, 0);

        //Wait until the piping thread releases to return
        sem_wait(&quitingSemaphore);

    //clean up the connection created.
```

```
        close(socketIn);
}

/*Establishes an incoming connection on the specified port and an outgoing
         connection on port and ipaddress specified and ports messages between them.*/
void RunBridgeMode(int incomingPort, char* outgoingPath, int outgoingPort)
{
        //Create an incoming connection
        int socketIn = CreateIncomingConnection(incomingPort);
        if(socketIn == FALSE)
                return;

        //Create an outgoing connection
        int socketOut = CreateOutgoingConnection(outgoingPath,outgoingPort);
        if(socketOut == FALSE)
        {
                close(socketIn);
                return;
        }

    pthread_t clientThreadId, serverThreadId;

    //Start 2 threads to pipe incoming data from either direction to the other connection
    pthread_create(&clientThreadId, NULL, (void *)PipeMessages, CreatePipePair(socketIn,
        socketOut));
    pthread_create(&serverThreadId, NULL, (void *)PipeMessages, CreatePipePair(socketOut,
        socketIn));

    sem_init(&quitingSemaphore, 0, 0);

    //Wait until both piping threads releases to return
    sem_wait(&quitingSemaphore);

    //Clean up the connections created.
    close(socketOut);
    close(socketIn);
}

//Creates a structure pairing an input fd with an output fd
pipePair* CreatePipePair(int in, int out)
{
    //Create the pair
    pipePair* pair = (pipePair*)malloc(sizeof(pipePair));

    //Assign their values
    pair->In = in;
    pair->Out = out;

    //Return it
    return pair;
}

//Takes in a pipePair and cycles through reading from the input
//side and writing that data to its output side.
void PipeMessages(void *value)
{
    //The input value is expedted to be a pipe pair
    pipePair *pair = (pipePair *)value;

    //Create a buffer to read data into
    int bufferSize = 8192;
    char inputBuffer[bufferSize];

    int bytesRead = 0;
    while(1)
    {
      //Read in from the input side
      bytesRead = read(pair->In, inputBuffer, bufferSize);

      if(bytesRead < 0)
        break;

        //Write to the output side
        if(write (pair->Out, inputBuffer, bytesRead) < 0)
          break;
    }
```

94

```
    //Notify the sempahore that we are no longer piping
    sem_post(&quitingSemaphore);

    //Clean up the value passed in.
    free(pair);
}
```

Appendix B

The Runtime

The contributions that The Runtime makes in order to provide the user with useful debugging information is detailed in Section 3.3. This appendix is provided to give the user some insight into select sections of The Runtime's code as well as to provide a reference to the instructions on how to integrate The Runtime into a user's code.

B.1   mpidebug.h

The `mpiddebug.h` header file contains the signatures of the methods that The Runtime uses when redirecting the user's MPI code. It is included in the `mpi.h` file that the user includes in step 2 of the steps to compiling The Runtime detailed in Appendix B.3. The contents of `mpidebug.c` are omitted from this Appendix, but Listing 14 shows the names of the methods that the user's MPI methods will be replaced with.

Listing 14: `mpidebug.h` source code.

```
#ifndef _MPIDEBUG_
#define _MPIDEBUG_
int _MPI_Init(char pname[100], int line, int *argc, char ***argv);
int _MPI_Finalize(char pname[100], int line);
int _MPI_Comm_rank(char pname[100], int line, MPI_Comm comm, int *rank);
int _MPI_Comm_size(char pname[100], int line, MPI_Comm comm, int *size);
int _MPI_Send(char pname[100], int line, void *buf, int count,
             MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) ;
int _MPI_Recv(char pname[100], int line, void *buf, int count,
             MPI_Datatype datatype,  int src, int tag, MPI_Comm comm,
             MPI_Status *status);

int _MPI_ISend(char pname[100], int line, void *buf, int count,
             MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *
                 request);

int _MPI_IRecv(char pname[100], int line, void *buf, int count,
             MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Request *request
                 );

int _MPI_Wait(char pname[100], int line, MPI_Request *request, MPI_Status *status);

int _MPI_Barrier(char pname[100], int line, MPI_Comm comm);

int _MPI_Probe(char pname[100], int line, int src, int tag, MPI_Comm comm, MPI_Status *
    status);

int _MPI_Probe(char pname[100], int line, int src, int tag, MPI_Comm comm, MPI_Status *
    status);

int _MPI_IProbe(char pname[100], int line, int src, int tag, MPI_Comm comm, int *flag,
    MPI_Status *status);

void* StdOutRedirectThread(void* value);

#endif
```

97

B.2 Redirecting Stdout

The Runtime redirect's its `stdout` file descriptor in order to be able to send what the user meant to print to the screen back to The Client. First, it calls the method RedirectStdOut which pipes whatever is written to `stdout` to a different file descriptor. The system then spawns off a dedicated thread to listen to the new file descriptor and send back messages to The Client which let it know what has been written to `stdout`. Listing 15 contains the two methods used to first redirect `stdout` on the main thread, and the worker thread tasked with sending back messages to The Client.

Listing 15: The code used to redirect The Runtime's `stdout` back to The Client.

```
void RedirectStdOut()
{
  int stdOutDupResult = dup(STDOUT_FILENO);

  //Exit out of the original one
  if( pipe(_out_pipe) != 0 ) {
      exit(1);
    }

  //Redirect the write side
  dup2(_out_pipe[1], STDOUT_FILENO);

  close(_out_pipe[1]);
  //Set the stdout buffer to autoflush
  setvbuf(stdout, NULL, _IONBF, 0);
}


//Listens to std out and sends its contents out as a serialized message
void* StdOutRedirectThread(void *value)
{
  //setup structures to peek at the read queue from
  fd_set rfds;
  struct timeval tv;

  int retval;
  int bytesRead;
  int bufferSize = 8192;
  char readBuffer[bufferSize];

  while(1)
  {
    FD_ZERO(&rfds);
    FD_SET(_out_pipe[0], &rfds);

    tv.tv_sec = 2;
    tv.tv_usec = 0;
    //wait up to 2 seconds to read
    retval = select(_out_pipe[0] + 1, &rfds, NULL, NULL, &tv);

    //Check if there was anything in the output buffer
    if(retval > 0)
    {
      bytesRead = read(_out_pipe[0], readBuffer, bufferSize);
      if(bytesRead > 0)//It better be greater than 1!
      {
        //Write out a console message to the users.
        writeToClient(serializeConsole(readBuffer, bytesRead, _rank,
                _sohReplace, _partitionReplace, _eotReplace));
      }
    }
    //Nothing read, check if we finalized yet
    else if( _finalized == TRUE)
    {
      FD_ZERO(&rfds);
```

```
        FD_SET( _out_pipe [0] , &rfds );
        tv.tv_sec = 2;
        tv.tv_usec = 0;

        retval = select ( _out_pipe [0] + 1, &rfds , NULL, NULL, &tv );
        if ( retval > 0)
          continue;
        else
          //Nothing in the stdout buff and we finalized , time to finish this thread
          break;
    }
  }

  sem_post(&_finalizedNotification);
  return 0;
}
```

B.3 Compiling The MPI Runtime

This section deals with the four step process needed to compile an MPI project with the correct libraries in order to allow for The Distributed Application Debugger's Client and Call Center to be able to interact with it.

1. **Copy the MpiFiles directory to your root directory.**

   The *MpiFiles* directory contains all of the files needed in order to compile your project. This directory just needs to be copied to your root directory one time and then, in order to inject The Distributed Application Debugger Runtime component, you just need to put your project files in this folder. Inside of this folder is a *bin* directory which will be the destination of all the binaries complied using the *Makefile* in step 4.

2. **Create .distributedApplicationDebugger.conf.**

   There is a hidden file called `.distributedApplicationDebugger.conf` which The Call Center looks for in order to know where to find the *MpiFiles/bin* directory created in Step 1. This file needs to just be created once and is expected to be placed in the user's root directory. For example, for user mjones the file would be placed in */home/mjones* and contain the following line:

   `/home/mjones/MpiFiles/bin/`

3. **Include `mpi.h`.**

   As detailed in section 3.3.1 the user needs to include the local `mpi.h` in their mpi files instead of the installed real framework mpi.h. This file will inject The Runtime into the code and then do all of the pre and post processing used to utilize the tool. In order to organize the header files, the MpiFiles includes the needed `mpi.h` in a *Headers* subdirectory, so if the files are included in the root MpiFiles directory, the included file would be `"Headers/mpi.h"`. The

file `text.c` included in the MpiFiles directory can be used as a sample for reference.

4. **Run the Makefile from the command line.**

   After including the `mpi.h` file included in the MpiFiles directory, the user needs to compile their application with the Distributed Application Debugger's assemblies. A Makefile to compile with is included within this directory too and its contents are shown in Listing 16. In order indicate the file to compile with the MPI runtime, the user must place the file in the MpiFiles directory and then run the *make* command with the name of the file to include. Assuming that the user is compiling from their root directory, the MpiFiles directory is located there and the file to debug is called `testParDev.c` the command line to make the file would be:

   ```
   make -C MpiFiles/ File=testParDev
   ```

   Note that the *.c* extension is NOT included in the command line, just the file name. The Makefile compiles all of the needed assemblies, including the XML, collection, parsing, and validation libraries and then compiles the user's code. The important line of the Makefile is line 21 which reads:

   ```
   mpicc -g -s -Wall -O0 -c -o $(FILE).o -DMPIDEBUG $(FILE).c
   ```

   The `-DMPIDEBUG` token means that when the header information from the `mpi.h` file illustrated in Figure 3.28 is compiled, that the `debug.h` and `mpidebug.h` files will be included as well. These files redirect the calls to the MPI library to intermediary libraries prefaced with underscores as illustrated in Figure 3.30. These methods wrap the standard four step debugging process around each call that is described in Section 3.3.4 which allows the entire system to work.

Listing 16: The Makefile included with the MPI runtime files.

```
1   # usage:  make clean
2   #          make FILE=test−file−name
3   #          make run FILE=test−file−name DATA=data−file−name
4
5   all:
6          gcc −Wall −O3 −Wno−unused −c −o XML/obj/xmlDoc.o XML/xmlDoc.c
7          gcc −Wall −O3 −Wno−unused −c −o XML/obj/xmlReader.o XML/xmlReader.c
8          gcc −Wall −O3 −Wno−unused −c −o XML/obj/xmlWriter.o XML/xmlWriter.c
9          gcc −Wall −O3 −Wno−unused −c −o obj/dictionary.o dictionary.c
10         gcc −Wall −O3 −Wno−unused −c −o obj/DADParser.o DADParser.c
11         gcc −Wall −O3 −Wno−unused −c −o obj/communication.o communication.c
12         gcc −Wall −O3 −Wno−unused −c −o obj/collections.o collections.c
13         mpicc −Wall −O3 −Wno−unused −c −o obj/mpiUtils.o mpiUtils.c
14         mpicc −Wall −O3 −Wno−unused −c −o obj/mpiXML.o mpiXML.c
15         mpicc −Wall −O0 −Wno−unused −c −o obj/mpiValidate.o mpiValidate.c
16         mpicc −Wall −O0 −Wno−unused −c −o obj/mpiSerialize.o mpiSerialize.c
17         mpicc −Wall −O0 −Wno−unused −c −o obj/gdbAttach.o gdbAttach.c
18         mpicc −Wall −O0 −Wno−unused −c −o obj/mpidebug.o mpidebug.c
19         strip XML/obj/*.o −S
20         strip obj/*.o −S
21         mpicc −g −s −Wall −O0 −c −o $(FILE).o −DMPIDEBUG $(FILE).c
22         mpicc −o bin/$(FILE) $(FILE).o XML/obj/*.o obj/*.o −lpthread
23  run:
24         ./$(FILE) $(DATA)
25
26  clean:
27         rm −f *.o
28         rm −f *~
29         rm −f obj/*.o
30         rm −f obj/*.~
31         rm −f XML/obj/*.o
32         rm −f XML/obj/*.~
33         rm −f Headers/*.*~
```

Appendix C

MPI Serializing

This appendix contains the examples illustrating the schemas of each of the twelve MPI commands supported by the Distributed Application Debugger. Listings 17-28 show what the XML serialized versions of each of the commands looks likes when saved during a Record session.

Listing 17: A serialized *MPI_Init()* command.

```
<MPI_INIT rank="0" commandId="1" dateTime="Mon Mar 04 08:55:15 2013 ">
  <returnvalue >0</returnvalue >
</MPI_INIT>
```

Listing 18: A serialized *MPI_Comm_rank()* command.

```
<MPI_RANK rank="0" commandId="2" dateTime="Mon Mar 04 08:55:15 2013 ">
  <parameters >
    <comm>1140850688</comm>
  </parameters >
  <returnvalue >0</returnvalue >
</MPI_RANK>
```

Listing 19: A serialized *MPI_Comm_size()* command.

```
<MPI_SIZE rank="0" commandId="3" dateTime="Mon Mar 04 08:55:15 2013 ">
  <parameters >
    <comm>1140850688</comm>
  </parameters >
  <returnvalue >0</returnvalue >
</MPI_SIZE>
```

Listing 20: A serialized *MPI_Send()* command.

```
<MPI_SEND rank="0" commandId="4" dateTime="Mon Mar 04 08:55:15 2013 ">
  <parameters>
    <buf>
      <value>0</value>
      <value>-2147483648</value>
      <value>2147483647</value>
      <value>356456</value>
      <value>765</value>
      <value>68378376</value>
      <value>67787</value>
      <value>17636</value>
      <value>585356</value>
      <value>253636</value>
    </buf>
    <count>10</count>
    <datatype>MPI_INT</datatype>
    <dest>1</dest>
    <tag>10</tag>
    <comm>1140850688</comm>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_SEND>
```

Listing 21: A serialized *MPI_Recv()* command.

```
<MPI_RECV rank="0" commandId="6" dateTime="Sun Aug 26 21:33:10 2012 ">
  <parameters>
    <buf>
      <value>H</value>
      <value>e</value>
      <value>l</value>
      <value>l</value>
      <value>o</value>
      <value> </value>
      <value>W</value>
      <value>o</value>
      <value>r</value>
      <value>l</value>
      <value>d</value>
      <value></value>
    </buf>
    <count>12</count>
    <datatype>MPI_CHAR</datatype>
    <src>1</src>
    <tag>10</tag>
    <comm>1140850688</comm>
    <status>
      <MPI_SOURCE>1</MPI_SOURCE>
      <MPI_TAG>10</MPI_TAG>
      <MPI_ERROR>0</MPI_ERROR>
    </status>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_RECV>
```

Listing 22: A serialized *MPI_Isend()* command.

```
<MPI_ISEND rank="0" commandId="42" dateTime="Sun Aug 26 21:33:10 2012 ">
  <parameters>
    <buf>
      <value>H</value>
      <value>e</value>
      <value>l</value>
      <value>l</value>
      <value>o</value>
      <value> </value>
      <value>W</value>
      <value>o</value>
      <value>r</value>
      <value>l</value>
      <value>d</value>
      <value></value>
    </buf>
    <count>12</count>
    <datatype>MPI_CHAR</datatype>
    <dest>1</dest>
    <tag>10</tag>
    <comm>1140850688</comm>
    <request>0</request>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_ISEND>
```

Listing 23: A serialized *MPI_Irecv()* command.

```
<MPI_IRECV rank="0" commandId="43" dateTime="Sun Aug 26 21:33:10 2012 ">
  <parameters>
    <count>12</count>
    <datatype>MPI_CHAR</datatype>
    <src>1</src>
    <tag>10</tag>
    <comm>1140850688</comm>
    <request>1</request>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_IRECV>
```

Listing 24: A serialized *MPI_Probe()* command.

```
<MPI_Probe rank="0" commandId="5" dateTime="Sun Aug 26 21:33:10 2012 ">
  <parameters>
    <src>1</src>
    <tag>10</tag>
    <comm>1140850688</comm>
    <status>
      <MPI_SOURCE>1</MPI_SOURCE>
      <MPI_TAG>10</MPI_TAG>
      <MPI_ERROR>0</MPI_ERROR>
    </status>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_Probe>
```

Listing 25: A serialized *MPI_Iprobe()* command.

```
<MPI_IProbe rank="0" commandId="41" dateTime="Sun Aug 26 21:33:10 2012 "
    >
  <parameters>
    <src>1</src>
    <tag>10</tag>
    <comm>1140850688</comm>
    <flag>0</flag>
    <status>
      <MPI_SOURCE>2</MPI_SOURCE>
      <MPI_TAG>0</MPI_TAG>
      <MPI_ERROR>0</MPI_ERROR>
    </status>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_IProbe>
```

Listing 26: A serialized *MPI_Wait()* command.

```
<MPI_WAIT rank="0" commandId="44" dateTime="Sun Aug 26 21:33:10 2012 ">
  <parameters>
    <request>0</request>
    <status>
      <MPI_SOURCE>2</MPI_SOURCE>
      <MPI_TAG>0</MPI_TAG>
      <MPI_ERROR>0</MPI_ERROR>
    </status>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_WAIT>
```

Listing 27: A serialized *MPI_Barrier()* command.

```
<MPI_Barrier rank="0" commandId="7" dateTime="Sun Aug 26 21:33:10 2012 "
    >
  <parameters>
    <comm>1140850688</comm>
  </parameters>
  <returnvalue>0</returnvalue>
</MPI_Barrier>
```

Listing 28: A serialized *MPI_Finalize()* command.

```
<MPI_FINALIZE rank="0" commandId="102" dateTime="Sun Aug 26 21:33:10
    2012 ">
  <returnvalue>0</returnvalue>
</MPI_FINALIZE>
```

Bibliography

[ANLD13] Mathematics Argonne National Laboratory and Computer Science Division. Web pages for MPI Routines. *http://www.mcs.anl.gov/research/projects/mpi/www/www3*, 2013.

[Bal13] Balsamiq Mockups Rapid Wireframing Tool. *http://www.balsamiq.com/*, 2013.

[BH04] Susanne M. Balle and Robert T. Hood. Global Grid Forum User Program Development Tools Survey. *Global Grid Forum*, 2004.

[Boe91] Barry W. Boehm. Software Risk Management: Principles and Practices. *IEEE Software*, January 1991.

[cMP94] cherri M. Pancake. What Users Need in Parallel tool Supoort: Survey Results and Analysis. *Technical report CSTR 94-80-3, Oregon State University*, June 1994.

[DDD13] DDD - Data Display Debugger. *http://www.gnu.org/software/ddd/*, 2013.

[Don94] Jack Dongarra. MPI: A Message Passing Interface Standard. *The International Journal of Supercomputers and High Performance Computing*, 1994.

[Ecl13] The Eclipse Foundation. *http://www.eclipse.org/*, 2013.

[Eis97] Marc Eisenstadt. My hairiest bug war stories. In *Debugging Scandal and What to Do About it — Communication of the ACM*. ACM Press, April 1997.

[Fos95] Ian Foster. Designing and Building Parallel Programs: Concepts and tools for parallel software engineering. *Addison Wesley*, 1995.

[GCC13] GCC, the GNU Compiler Collection. *http://gcc.gnu.org/*, 2013.

[GDB13] GDB - The GNU Debugger. *http://www.gnu.org/directory/gdb.html*, 2013.

[Gib94] W. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*, September 1994.

[Hel00] Gilbert Held. *Understanding Data Communications: From Fundamentals to Networking*. Wiley, third edition, December 2000.

[HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, first edition, October 1999.

[Lar07] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, third edition, October 2007.

[LMC87]   Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, April 1987.

[Mon13]   Mono - Cross platform, open source .NET development framework. *http://www.mono-project.com*, 2013.

[MS08]   Norman Matloff and Peter Jay Salzman. *The Art of Debugging with GDB, DDD, and Eclipse.* No Starch Press, first edition, September 2008.

[MSN13]   Microsoft Visual Studio. *http://msdn.microsoft.com/vstudio/*, 2013.

[Pan93]   Cherri M. Pancake. Why Is There Such a Mis-Match between User Need and Parallel Tool Production? *Keynote address, 1993 Parallel Computing System: A Dialog between Users and Developers.*, April 1993.

[Ped03]   Jan B. Pedersen. Multilevel Debugging of Parallel Message Passing Systems. *PhD Thesis, University of British Columbia, Vancouver, British Columbia, Canada*, June 2003.

[Ped06]   Jan B. Pedersen. Classification of Programming Errors in Parallel Message Passing Systems. *In* Proceedings of Communicating Process Architectures 2006 (CPA'06) *IOS Press*, September 2006.

[PJ12]   Jan B. Pedersen and Michael Q. Jones. Error Classifications for Parallel message Passing Programs: A Case Study. *Proceedings of Parallel and Distributed Processing techniques and Applications (PDPTA'12)*, July 2012.

[Pro13]   The Open MPI Project. FAQ: Debugging applications in parallel. *http://www.open-mpi.org/faq/?category=debugging*, 2013.

[SEI13]   Software Engineering Institute. *http://www.sei.cmu.edu/*, 2013.

[Sha13]   SharpSSH - A Secure Shell library for .NET. *http://www.tamirgal.com/blog/page/SharpSSH.aspx*, 2013.

[Sof13a]   Allinea Software. DDT product page. *http://www.allinea.com/products/ddt*, 2013.

[Sof13b]   Rogue Wave Software. TotalView product page. *http://www.roguewave.com/products/totalview.aspx*, 2013.

[Sub13]   Apache Subversion. *http://subversion.apache.org*, 2013.

[UC09]   Russ Unger and Carolyn Chandler. *A Project Guide to UX Design: For User Experience Designers in the Field or in the Making.* New Riders, first edition, March 2009.

[War09]   Todd Zaki Warfel. *Prototyping: A Practitioner's Guide.* Rosenfeld Media, first edition, November 2009.

[XML13]   XML Technology. *http://www.w3.org/standards/xml/*, 2013.

Vita

Graduate College
University of Nevada, Las Vegas

Michael Q. Jones

Degrees:

Bachelor of Science in Computer Engineering 2003
University of Wisconsin-Madison

Thesis Title: The Distributed Application Debugger

Thesis Examination Committee:

Chairperson, Dr. Jan B. Pedersen, Ph.D.
Committee Member, Dr. Ju-Yeon Jo, Ph.D.
Committee Member, Dr. Evangelos Yfantis, Ph.D.
Graduate Faculty Representative, Dr. Aly Said, Ph.D.