Report Number: WUCS-95-14

1995-01-01

Distributed Debugging With I/O Abstraction

Authors: Andrew S. Koransky

This thesis presents a simple, yet powerful, set of mechanisms for testing and debugging distributed applications consisting of modules that communicate through well-defined data interfaces. The tools allow default or programmer-defined functions to be attached to various communication events so that particular data values at interesting points in the program are made available for testing and debugging. The debugging status of each component of the communication interface can be controlled separately so that various debugging information can be turned on and off during program execution. By attaching breakpoints to programmer-defined fucntions in a standard debugger, fine-grained examination of each module of the applicaton can be integrated with the coarse-grained communication debugging information provided by our tools.

DISTRIBUTED DEBUGGING WITH I/O ABSTRACTION

Andrew S. Koransky

WUCS-95-14

May 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

Undergraduate Thesis Advisor: Kenneth J. Goldman

Distributed Debugging with I/O Abstraction

Andrew S. Koransky

May 12, 1995

Abstract

This thesis presents a simple, yet powerful, set of mechanisms for testing and debugging distributed applications consisting of modules that communicate through well-defined data interfaces. The tools allow default or programmer-defined functions to be attached to various communication events so that particular data values at interesting points in the program are made available for testing and debugging. The debugging status of each component of the communication interface can be controlled separately so that various debugging information can be turned on and off during program execution. By attaching breakpoints to programmer-defined functions in a standard debugger, fine-grained examination of each module of the application can be integrated with the coarse-grained communication debugging information provided by our tools.

1 Introduction

As more and more people jump into the global communication infrastructure, it is becoming increasingly important to provide user friendly tools for writing distributed applications. I/O abstraction provides all users with a simplified communication paradigm, making it easy to quickly write distributed applications. The programming environment has been designed to work in heterogeneous environments, with different programming languages running on various operating systems. I/O abstraction also allows for dynamic configuration of the communication environment. However, even with a simpler communication model, it is necessary to provide users with a debugging mechanism which will aid in finding bugs in user programs.

In this thesis, we will first introduce the I/O abstraction model being used and discuss some of the background work that was necessary for creation of the debugger. Alternate designs will be discussed, followed by a discussion

of the chosen design for our debugger. A manual for using the debugger may also be found at the end of this thesis.

2 Background

I/O abstraction is based on a formal model of distributed computing called I/O automata. [1] An I/O automaton is a state machine consisting of input actions and locally controlled actions. The locally controlled actions can be divided into output actions and internal actions. The I/O automaton may change state on an input action and as a result, might perform an output action. In addition, internal actions can change the state and cause output actions as well.

The Programmers' Playground [2] is a distributed application development environment that is based, in part, on ideas from the I/O automaton model. Specifications of I/O automaton systems can be described as "schedule modules" that characterize the allowable sequences of events that can occur at the module interface. Similarly, one can describe the "behavioral specification" of a Playground module as the allowable sequence of states that can occur at the presentation. For each module in a distributed application, a programmer declares a set of published variables available for other programs to read or write. At run-time, the user may specify (using a GUI) logical connections between similarly typed variables in other modules. Once a connection is made between variables, a change in one readable variable causes a change in the recipient writable variable. If the programmer chooses, this change can be seamless. In other words, the variable will change values without informing the programmer or interrupting the execution of the program. The programmer may also declare a reaction function in which the program can react to a change in a variable's value. The system provides asynchronous communication; however it also provides "first in first out" or FIFO communication between modules.

The current implementation of the system requires a centralized connection manager. The connection manager is a Playground module itself and is used in keeping track of connections between modules and sending requests to modules. The GUI is a separate Playground module which connects to the connection manager and receives all information about the current modules running and the connections between them. When the user uses the GUI to make a logical connection between variables, the GUI sends a connection request to the connection manager. The connection manager then sends a request to one of the modules, allowing that module to connect to the other

module. With this implementation, it is possible for each module to operate only knowing the location of the connection manager and not knowing about any other modules. To allow easy creation of graphical interfaces to Playground modules, EUPHORIA [3] has been developed. EUPHORIA allows a user to connect Playground variables to graphical items such as a box's height or a circle's center.

The system was developed with the GNU C++ compilers on Sun SPARCstations running Solaris. To create a Playground application, users simply needed to include a file in their C++ source and link their code with the library. With the previous implementation of the system, each module consisted of two processes communicating with shared memory and using semaphores to control access to the shared memory. One process would control communication between the modules. Incoming and outgoing information was stored in shared memory. Whenever the user's program accessed a Playground variable, the shared memory was checked for new information. If the user's program set the value of a variable, information was placed into the shared memory to be given to the other process. We found this implementation to be inefficient and difficult to port. We also found it more difficult to design a debugger for a system using such an implementation. Therefore, we removed the shared memory and semaphores and placed the communication functionality inside of the Playground library. As a result, we have reduced latency and round trip delays, we have made the system more portable by removing operating system dependent calls, and we have prepared the system for the introduction of a debugger.

3 Motivation

The Playground programming environment removes all the details of communication from the user and presents the user with an easy to use GUI. The GUI assists in showing the programmer how communication is occuring within the system. This greatly reduces the need of a debugger; however, we have found it necessary to provide the user with some type of mechanism that will allow one to monitor changes in the published variables.

4 Related Work

Much related work has been done in the field of distributed debuggers. However, most of this work revolves around a system where the communication is explicit in a program rather than implicit. Some of these systems require kernel

modification and daemons, hence preventing portability. For example, Smith [4] introduces a system that uses kernel modifications and daemons working with a debugger to debug a distributed program. Smith's debugger allows you to replay a process with the information stored by the daemons. IDD [5] deeply embeds its debugger into the operating system and provides state examination, stepping, tracing, and assertion monitoring. Although many of the designs mentioned above have useful properties in certain environments, they do not allow for a programming environment where the communication is implicit such as the Playground. It is also important to remember that one of the goals of the Playground Project is portability for heterogeneous environments. With such goals, we choose not to implement kernel modifications and operating system specifics as part of our debugger

Other systems have introduced some useful concepts. Bugnet [6] introduces a system with checkpoints for synchronizing two processes. Each process then records information on I/O events and tracing information. This is useful for seeing what events led up to the error. Amoeba [7] is a system where the debugger is not transparent to target programs. Each program would has knowledge of the debugger and all communication is synchronous. Amoeba provides check-pointing (synchronizing programs at a checkpoint), and roll back (reviewing the actions of programs). Some simpler systems synchronize clocks and record events such as Bates and Wileden [8]. If events and traces have been logged, Garcia-Molina [9] suggests bringing the program into an "artificial" environment and trying to create the conditions that led to the crash. With the Playground paradigm, it would be possible to create such logging and synchronization mechanisms. These mechanisms could be useful in the Playground environment, but we were unable to implement these ideas due to time constraints.

5 Design

In designing a debugger for the Playground, it is important to remember that communication in the Playground is implicit, and that connections between modules are dynamic (IE, they can be changed at run time). This means, that one module, A, might be connected to another module, B, during some point of module A's life. However, the user might quit module B and start up a different module C. C might have a different functionality than B, and the user might connect A to C in the GUI. Because these connections are determined at run-time, the operation of a Playground application as a whole is not deterministic. The ease of creating new communication graphs between

creates somewhat of a debugging environment in and of itself. As discussed above, [9] Garcia-Molina mentions an "artificial environment" for reproducing the error. The Playground model essentially allows you to create an "artificial environment" in order to reproduce some errors. A module can be built to provide the artificial environment necessary to reproduce the error.

It is also important to note that errors in distributed systems can occur in the system calls that initiate, perform, and destroy communication streams. The Playground completely removes this layer of complexity from the user. Because the communication is implicit, the programmer does not have the difficult task of ensuring that communication is occuring properly.

In keeping with the spirit of the I/O automaton model, we wanted the debugger to allow the user to determine if each module was behaving in a manner consistent with its behavioral specification. Therefore, we decided that the debugger should monitor changes to the values in the module's presentation. To monitor the data, we determined that it would be useful for a user to place a breakpoint on a logical connection between one variable to another. The user could then step through the program to find out how the error occured. For the long term, we also wished to have control of the debugger through the GUI. We generated some alternate designs before settling on a final design.

5.1 Alternate Design

One design that we generated requires use of the GNU compilers and the GDB debugger. It consists of a Playground module called the GDB extension. Each GDB extension creates an invisible logical connection into the GUI module. A user could specify that a link in the GUI was to be a debugging link. The GUI module would then communicate with the GDB extension to establish breakpoints. The GDB module would then communicate with GDB to set the breakpoints in the debugger.

This method would be very difficult to implement. Although it would be very user friendly, the design could not be ported to other systems. The interface into GDB is poorly documented, and on other systems with different compilers and debuggers, there is not necessarily an interface into the debugger at all. The complexity of implementing such a system was also a drawback.

5.2 Design of the Debugger

The Playground model currently makes use of event-driven reaction functions. One way to debug a program is to set a breakpoint on a function. To make

the debugger consistent with the Playground model, we created a mechanism in which the user can register functions for events. These events include:

- Pre-Send: before the new value is set or sent
- Post-Send: after the new value has been set and sent
- Pre-Receive: before setting the PG variable to the incoming value
- Post-Receive: after setting the PG variable to the incoming value
- Pre-React: before calling the reaction function
- Post-React: after calling the reaction function

When an event occurs, a call to the registered function for that event occurs. The user can set a breakpoint on this function and step through the module's source, or the function could simply print the value of the variable. In addition, to simplify future enhancements, events may be filtered by setting flags. These flags can be turned on and off throughout the module's execution. If a flag for a specific event has been turned off, that event will not trigger a call to the user's registered function.

This design is highly portable. Breakpoints can be set in any development environment on the registered debugging function. The user can then browse through the local and remote variables in the program upon the given event. The debugger is also extremely easy to use and contains the functionality to be fully extended with GUI support.

6 Future Work and Summary

To make the debugger easier to use, we would eventually like to support flag setting for each published variable through the GUI. We also hope to support data visualization in the GUI such that you can see the data moving from the origin of the link to the destination. Other possibilities for augmenting the debugger include a distributed snapshot algorithm and event logging.

We have shown how the debugger can be useful in the Playground environment. Now that a simple mechanism exists to monitor what events are occuring on a module's published variables, we expect debug time for distributed Playground applications to be reduced significantly.

7 Acknowledgments

Special thanks to Ken Goldman for providing me with the opportunity to work on this project as an undergraduate and for helping me out with this thesis.

A User's Guide

The Programmers' Playground provides a simple set of software tools to assist the programmer in debugging applications. To use the debugger, you can declare a set of debugging functions. Each debugging function can get called upon a specific event. These events include:

- Pre-Send: before the new value is set or sent
- Post-Send: after the new value has been set and sent
- Pre-Receive: before setting the PG variable to the incoming value
- Post-Receive: after setting the PG variable to the incoming value
- Pre-React: before calling the reaction function
- Post-React: after calling the reaction function

Debugging functions can be registered with the debugger for each event. The debugger allows registration of functions for each Playground variable. When an event occurs, the veneer will call the registered function for that event if the flag is set.

A simple debugging function would print the value of the Playground variable. If using in conjunction with a debugger such as GDB, a breakpoint can be placed on a debugging function.

The debugger also allows flag setting for specifying if an event should trigger its respective debugging function. Eventually, Playground will support run-time flag setting via the GUI.

A.1 PGdebug

PGdebug registers a Playground variable with the debugger:

```
PGint a;
PGpublish(a,"a",RW_WORLD);
PGdebug(a);
```

This would register the Playground integer a with debugger. By default, all flags are set, meaning that events will trigger the registered functions. Also by default, a function that prints the value of the variable is registered for each event. This default function is defined as follows:

```
void
PGdefaultDebug(PGobj* x)
{
  cout << "DEBUG value: ";
  x->print();
  cout << endl;
}
```

You may override the defaults in your call to PGdebug. Following the PGobj pointer, you can set flags, followed by the call-back functions as in the example below:

```
void
dummy(PGobj* x)
{
}

// ...

PGint a;
PGpublish(a,"a",RW_WORLD);
PGdebug(a,                  // This is the PGobject we wish to debug
        (cbfDEBUGPRESEND  | cbfDEBUGPOSTSEND |  // These are the flags
         cbfDEBUGPREREACT | cbfDEBUGPOSTREACT),
        &dummy,             // This is the pre-send callback function ptr
        &dummy,             // This is the post-send callback function ptr
        &PGdefaultDebug,    // This is the pre-receive callback function ptr
        &PGdefaultDebug,    // This is the post-receive callback function ptr
        &dummy,             // This is the pre-react callback function ptr
        &dummy);            // This is the post-react callback function ptr
```

The example above registers callback functions for all events, but only sets flags for the pre-send, post-send, pre-react, and post-react events. These events will trigger a call to the dummy function. If using a debugger, a breakpoint could be placed on the dummy function.

PGdebug can be called at any time after a call to PGinitialize. The variable must be published before calling PGdebug.

A.2 PGdebugSetFunction

After a call to PGdebug, you may change your registered function for a specific event with a call to PGdebugSetFunction:

```
void dummy(PGobj* x)
{
}

// ...

PGreal a;
PGpublish(a,"a",RW_WORLD);
PGdebug(a,cbfDEBUGPRERCV | cbfDEBUGPRESEND);
PGdebugSetFunction(a,fDEBUGPRERCV,&dummy);
PGdebugSetFunction(a,fDEBUGPRESEND,&dummy);
```

In the example above, the call to PGdebug would register functions for Playground variable a and set the flags for the pre-receive event and the pre-send event. The call to PGdebugSetFunction changes the call-back function from the PGdefaultDebug function to the dummy function for the specified events. The code above effectively registers the dummy function for the two events, and it sets flags for those events. This insures a call to dummy will occur upon the events, pre-receive and post-receive.

The second parameter, the event specifier, can be one of the following constants:

- `fDEBUGPRESEND`
- `fDEBUGPOSTSEND`
- `fDEBUGPRERCV`
- `fDEBUGPOSTRCV`
- `fDEBUGPREREACT`
- `fDEBUGPOSTREACT`

A.3 PGdebugSetFlags

PGdebugSetFlags allows you to set flags which specify which events will trigger calls to the registered functions:

```
PGstring str;
PGpublish(str,"str",RW_WORLD);
PGdebug(str);
PGdebugSetFlags(str,cbfDEBUGPRESEND | cbfDEBUGPOSTSEND );
```

The code above will register the Playground variable `str` with the debugger and register the default debug functions for each event. The PGdebugSetFlags function will set the flags for the events pre-send and post-send that might occur with the variable `str`.

Flags can also be set when registering your variable with PGdebug. The code above could also be written as follows to achieve the same effect:

```
PGstring str;
PGpublish(str,"str",RW_WORLD);
PGdebug(str,cbfDEBUGPRESEND | cbfDEBUGPOSTSEND );
```

Therefore, calls to PGdebugSetFlags are most useful at runtime. See the Extended Example for an example which sets flags at runtime.

More than one flag can be specified by using the | operator. Valid callback function flags are:

- `cbfDEBUGPRESEND`
- `cbfDEBUGPOSTSEND`
- `cbfDEBUGPRERCV`
- `cbfDEBUGPOSTRCV`
- `cbfDEBUGPREREACT`
- `cbfDEBUGPOSTREACT`

A.4 Extended Example

```
// consumer-debug.cc
//
// This module has a single writable integer presentation variable.
// Whenever the value changes, it is printed to standard output.
// The module quits if the value goes negative.
//
// Debug code has been added such that:
//   - the integer is registered with the debugger, default debug function is
//     registered for all events on the integer
//   - functions are registered for pre-receive and post-receive events on the
//     integer, which overrides the default debug function for those
//     events.
//   - if the PG int changes to a number greater than 100, the flags
//     change such that the pre-react and post-react events cause a call
//     to the registered functions for that event (default debug function in
//     this case)
//   - if the PG int changes to a number less than 100, the flags are changed
//     such that the pre-receive and the post-receive events trigger a call
//     to the registered functions for that event (debug_prercv and
//     debug_postrcv respectively in this case).

#include <iostream.h>
#include "PG.hh"

static int done = 0;

void
test_and_print(PGobj* v) {
  PGint& i = (PGint&) *v;

  if (i < 0)
    done = 1;
  else
    {
      if (i > 100)
        PGdebugSetFlags(i,cbfDEBUGPREREACT | cbfDEBUGPOSTREACT);
      if (i <= 100)
        PGdebugSetFlags(c, cbfDEBUGPRERCV | cbfDEBUGPOSTRCV);
```

```
      cout << i << endl;
    }
}

void
debug_postrcv(PGobj* a)
{
  cout << "DEBUG postrcv:";
  a->print();
  cout<<endl;
}

void
debug_prercv(PGobj* a)
{
  cout << "DEBUG prercv:";
  a->print();
  cout<<endl;
}

int
main()
{
  PGint c = 0;

  PGinitialize("CONSUMER");
  PGpublish(c,"c",WRITE_WORLD);
  PGdebug(c);
  PGdebugSetFunction(c,fDEBUGPRERCV,debug_prercv);
  PGdebugSetFunction(c,fDEBUGPOSTRCV,debug_postrcv);
  PGdebugSetFlags(c, cbfDEBUGPRERCV | cbfDEBUGPOSTRCV);
  PGreact(c,test_and_print);
  PGreactUntilTrue(done,0,0);
  PGterminate();
}
```

References

[1] N. A. Lynch and M. R. Tuttle, "An introduction to I/O automata", *CWI-Quarterly*, **2**, (1989).

[2] K. J. Goldman, M. D. Anderson, and B. Swaminathan, "The Programers' Playground: I/O Abstraction for Heterogeneous Distributed Systems", *Proceedings of the 27th Hawaii International Conference on System Sciences*, 1994, pp. 363-372.

[3] T. P. McCartney and K. J. Goldman, "Visual Specification of Interprocess and Intraprocess Communication", *Proceedings of the 10th International Symposium on Visual Languages*, 1994, pp. 80-87.

[4] E. T. Smith, "A debugger for message-based processes", *Software – Practice and Experience*, 15, 1073-1086 (1985).

[5] P. Harter, D. Heimbigner and R. King, "IDD: an interactive distributed debugger", *IEEE Proc. 5th Int. Conf Distributed Computing Systems*, 1982, pp. 498-506.

[6] R. Curtis and L. Wittie, "Bugnet: a debugging system for parallel programming environments", *IEEE Proc. 3rd Int. Conf. Distributed Computing Systems*, 1982, pp. 394-399.

[7] I. J. P. Elshoff, "A distributed debugger for amoeba", *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, **24**, (1), 1-10 (1989).

[8] P. Bates and J. C. Wileden, "High-level debugging of distributed systems: the behavioral abstraction approach", *Journal of Systems and Software*, **3**, (4), 255-264 (1983).

[9] H. Garcia-Molina, F. Germano and W. Kohler, "Debugging a distributed computing system", *IEEE Trans. Software Engineering*, **SE-10**, (2), 210-219 (1984).

[10] R. S. Side and G. C. Shoja, "A debugger for distributed programs", *Software: Practice and Experience*, **24**, 507-525 (1994).