

TOTALVIEW TECHNOLOGIES WHITE PAPER

**Deterministically
Troubleshooting Network
Distributed Applications**
Chris Gottbrath,
TotalView Technologies





W H I T E P A P E R

Abstract

Debugging is all about understanding what the software you are looking at is really doing. Computers are unforgiving readers; they never pay attention to what you mean, and always insist on doing what the code says.

Debugging happens naturally when actively developing code and troubleshooting a problem. The same kind of investigation is also a great way to learn about programs that are working just fine. It pays to look closely at what programs are really doing when you re-introduce yourself to code that you wrote a long time ago, or when you try to understand a new bit of code that you encounter for the first time.

Debugging is hard. Almost every programmer can relate to Brian Kernighan's quote in "The Elements of Programming Style," which states that "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

There are several reasons why debugging is hard. The main reason is simply that it is difficult to be objective. You build mental models of the program executing when you read code. If you wrote the code, how do you keep those models from being influenced by knowing how the code was intended to work? The fact that it is easier to be objective reading other people's code is one of the main reasons for code reviews.

In addition to the fundamental challenge of objectivity, there are a number of mundane facts that make debugging hard. First, it can be difficult to reproduce whatever triggers bugs, especially bugs that occur on shipping applications or systems in production. The challenge is knowing what to pay attention to and deciding how much detail about the system environment is relevant for a given problem. Finally, once you have the right environment, you need to gather and display detailed information about the execution of the program.

This paper will look at three different ways to approach debugging a client-server application: tracing, interactive debugging, and replay debugging. The application I am looking at is a simple multi-threaded, multi-machine, memory status monitoring application written in C using the UNIX socket interface. The purpose is to share some techniques that may give you new ideas on how to tackle bugs you may encounter with network programs.

Tracing

Tracing the execution of one or two of the processes that make up the status-monitoring program enables us to understand the sequence of the program. In effect, you are building a list of things that happen in the program and their order. This can be very useful when troubleshooting, and also when examining or learning an unfamiliar program.

Strace

There are several different ways to approach tracing a program. The first is to use a system tracing tool that records all the times that the program makes system calls. A well-known system tracing program on Linux is *strace* and similar tracing calls are available on other operating systems. *Strace* is open source and is probably part of your favorite Linux's package repository. I can run it on my server application with the command

```
strace -o trace-log.txt ./server
```

or if my server is already running I can attach *strace* to it with a command line

```
strace -o trace-log.txt -p 3216
```

W H I T E P A P E R

where 3216 is the process id of my server. In either case, *strace* creates an output file called trace-log.txt. The log file is plain text with each system call, complete with arguments, followed by the return code for the system call.

```
write(1, "this is the server\n", 19) = 19
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
write(2, "listening filedescriptor 3\n", 27) = 27
write(2, "constructed my addr \n", 21) = 21
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [49], 4) = 0
bind(3, {sa_family=AF_INET, sin_port=htons(54661), sin_addr=inet_addr("192.168.88.102")}, 16) = 0
futexp(0xb7ddcc2c, FUTEX_WAKE, 2147483647) = 0
brk(0) = 0x804b000
brk(0x806c000) = 0x806c000
write(2, "bound to port 54661 on 192.168.8...", 38) = 38
listen(3, 3) = 0
write(2, "listening for 3 on port 54661 on...", 48) = 48
```

Read the main page for all of the options. Four worth noting here are:

- t which gives timestamp information (note that there are a number of timestamp options to choose from)
- T which gives elapsed time in system calls
- f which requests *strace* to follow fork calls
- v which prints full details on long system calls

Strace will tell you about all the system calls in your program, not just the network-related calls, which makes it handy when you need to figure out issues such as where a poorly documented program is looking for configuration files.

Print Statements

The major limitation of *strace* is that it only tells you about system calls and not about the internal behavior of the program itself. If you want to follow the logic of the program itself you need a way to instrument the program. The first way to do this is to use print statements. A principal advantage of using print statements, especially when you are trying to understand code as you write it, is that it can be done using the tools you use to write the code in the first place.

Print statements inserted into a program can be useful for understanding the sequence of operations and for following specific data within the program. It is easy to place a few too many and get too much information. A good practice is to use as few print statements as possible, by adding and removing them throughout the debugging process. Since any change requires a recompile, this can be tedious in practice.

Print statements are good when you want to look for unusual behavior in routines that are executed extremely frequently. In this case, either the print statement can be used to output some data that can be analyzed after the fact, or the print statement can be combined with conditional test statements that help identify that the conditions are such that you want to know them.

tvscript

If you don't want to take the time to recompile your program over and over again just to move, add or change the print statements, there is a way to have the best of both worlds.

You can get the instant gratification of being able to easily change the functions and variables from which you print out data, together with the benefits of seeing everything listed out in a logfile that you can study to get a picture of the sequence of actions your program takes. You do need to start with a version of your program compiled with debug symbols as shown

W H I T E P A P E R

```
gcc -g -o server-dbg server.c
```

Then you can use a script that uses a debugger to get information from the application and writes it out to a logfile. You want a script that makes it easy to define a set of points within the program that you are interested in. The script will run the program to these points and each time they are reached it will output information to a logfile.

It would be possible to write a script to drive GDB or any other command line debugger in this fashion. A script called `tvscript` that does this is included as part of the TotalView debugger.

For example, say that I wanted to know about the sequence of my program in terms of two specific functions. The following command would create a trace log that tells me about each time that my server executes `my_create_port()`, `server_listener()`, or line 187 in the file that contains `handle_connections_threads()`.

```
tvscript \  
-create_actionpoint "my_create_port=>display_backtrace -show_arguments" \  
-create_actionpoint "server_listener=>display_backtrace" \  
-create_actionpoint "handle_connections_threads#187=>display_backtrace -show_locals" \  
-maxruntime "00:00:30" \  
./server
```

This will create a logfile in my current directory with a series of output blocks, one for each time the program executes one of the designated functions. Each output block will give the timestamp and a detailed backtrace for what was happening in the program when the event occurred.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
! Backtrace  
!  
! Process:  
! ./server (Debugger Process ID: 1, System ID: 17031)  
! Thread:  
! Debugger ID: 1.1, System ID: 3084126880  
! Time Stamp:  
! 06-25-2008 19:50:06  
! Triggered from event:  
! actionpoint  
! Results:  
! > 0 my_create_port PC=0x080488f3, FP=0xbfdac718 [/tvscript/server.c#40] [/tvscript/server]Function "my_create_port" arguments:  
! 0.1 port_number = 0x0000d585 (54661)  
! 0.2 ipv4_addr_string = 0x0804947a -> '1' (0x31, or 49)  
!  
! 1 main PC=0x08049106, FP=0xbfdac748 [/tvscript/server.c#216] [/tvscript/server]Function "main" arguments:  
! 1.1 argc = 0x00000001 (1)  
! 1.2 argv = 0xbfdac7d4 -> 0xbfdada76 -> "./server"  
!  
! 2 __libc_start_main PC=0xb7d55e9f, FP=0xbfdac7a8 [/lib/lsb686/cmov/libc.so.6]  
!  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

You can easily add a command to print out specific variables at any point in the program. The following command flag tells `tvscript` to report the contents of the `foreign_addr` structure each time the program gets to line 85:

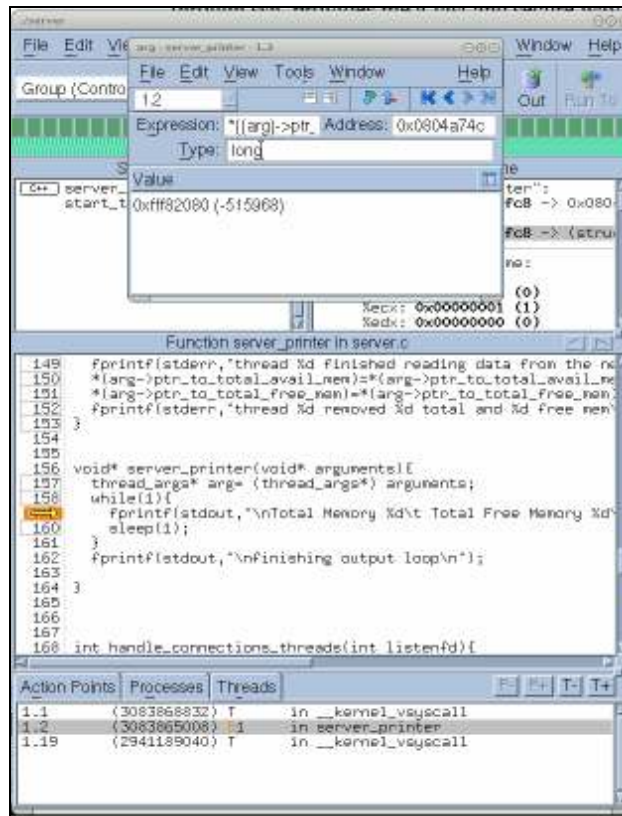
W H I T E P A P E R

An interactive source code debugger gives the developer very precise ways to control the application. Two important basic examples of program control are single stepping and breakpoints. Single stepping is an excellent way for you to follow the execution of a complex or unfamiliar routine. The debugger very carefully runs the program just far enough to get to the next line of code, allowing you to watch where the program goes (following the instruction pointer) and how data is used (watching variables, registers, and expressions). A debugger should highlight variables that change during any given step operation. When the program includes non-trivial conditional statements, lining up interesting input and then stepping through the code allows you to see very clearly what paths are being taken.

The second basic tool that a debugger provides for controlling program execution is the breakpoint. You should be able to set a breakpoint on any executable line of code in the program and have it stop at that location. The debugger should also make it easy to evaluate the conditions when the program gets to that point and make a decision about stopping or continuing on. You may want, for example, to stop a process when some expression looks clearly wrong

```
if (*(ptr->memTotal) < 0) { $stop; }
```

This lets the debugger do the grunt work of watching to see when the value goes out of bound. The expression above is evaluated each time the associated breakpoint is reached. If it ever evaluates the \$stop the debugger halts the application so you can examine it.



The expression stops the process when it detects an out of bound condition (a negative total memory in this case). The breakpoint associated with the expression was triggered in thread 1.2.



W H I T E P A P E R

You should expect your debugger to provide you with more control than this. With an application that has multiple threads, there are a very large number of execution sequences that the program might possibly take. A debugger should provide you with the ability to explore any conceivable execution sequence. For example, a region executed freely by more than one thread that operates on any kind of global data structure may be the site of a race condition. The debugger should give you the ability to control thread execution in such a way that several different threads enter that section at about the same time. Sequences of execution with a high degree of overlap may bring the race condition out into the open.

One issue that can make interactive debugging problematic for network programs is that it frequently takes more than a few seconds to absorb a screen full of information about your program. When you are using an interactive debugger on a network application that aggressively times out inactive connections, then just pausing the application to examine its behavior can trigger timeouts.

If this is the case, the easiest thing to do is simply to raise the timeout value. Another possibility is to use a debugger on both ends of the connection. If you pause both processes in just the right place, you may be able to either pause the remote process at a point when its timeout timer isn't running, or prevent the connection from being closed by steering the application away from the connection closing sequence.

Interactive replay debugging

A better approach would be to let your program continue running so that it can handle all those incoming connections, but still allow you to explore the behavior of your program at whatever level of detail you need to ultimately find the problem. Your third general debugging strategy is to run the program under a record and replay debugger to the point where it has done whatever you are interested in seeing, then go back and examine how that happened. Usually this means that you run the program till it crashes, hangs, or gives invalid data. Using the debugger, you can examine the crashed or hung state, looking for clues about what might be wrong. You can then examine recorded program history – searching backwards for the cause of the error.

This strategy has some extremely compelling advantages. First, because you are not attempting to examine the process with the debugger while it is running, you only pause it after the interesting behavior has happened. At that point any network connections that remain open can be closed or allowed to hang or time out. There is certainly overhead involved with recording the program execution history, but that overhead is more evenly distributed and the program remains responsive to incoming network traffic.

Another benefit compared with traditional interactive debugging is that you can use specific details gleaned from the crash or hang to guide your thinking while you are working. Looking through the execution history for the cause of an array bounds violation at a specific address is a much more narrowly focused process than trying to spot a bounds violation that might be happening anywhere in a currently running process.

Finally, you can look forwards and backwards in the execution history, making comparisons, and checking things more than once if you need to. This fact that you can look at something more than once if you need to reduces the pressure of the troubleshooting process.

The TotalView Tech ReplayEngine provides developers with just such a recording and replay mechanism. Tell TotalView to load the ReplayEngine and debug server-dbg with the following command line:

```
totalview –replay ./server-dbg
```

Once the server program is running under the ReplayEngine, you can let it run forward to the point that it is misbehaving. Once the program crashes, you examine the state of the crash and work your way backwards towards the cause. The interface provides backwards-stepping buttons that show the program as it was one line earlier in execution history each time

W H I T E P A P E R

they are clicked. If you want to jump farther back, you can select a line anywhere in the program and use the “go back” operation to examine the program at the point in history when it most recently executed that line.

The screenshot displays the TotalView debugger interface. At the top, the menu bar includes File, Edit, View, Group, Process, Thread, Action Point, Instrumentation, Tools, Window, and Help. Below the menu is a toolbar with icons for Go, Halt, Kill, Restart, Next Step, Out, Run To, Prev, UnStep, Caller, Back To, and Live. The status bar shows 'Process 1 (15425): server (Stopped)' and 'Thread 1 (15425) (Stopped) <Stop Signal>'. The main window is divided into several panes. The 'Stack Trace' pane on the left shows the call stack: my_create_port (FP=bfee9a88), main (FP=bfee9ab8), and __libc_start_main (FP=bfee9b18). The 'Stack Frame' pane on the right shows the details for the 'my_create_port' function, including port_number (0x0000d585 / 54661), ipv4_addr_string (0x0804947a), and local variables: listenfd (0x00000004 / 4), queuedepth (0x08049489 / 134517), my_addr (struct sockaddr_in), and yes (0x00000031 / 49). The 'Registers for the frame' pane is empty. The main code editor shows the source code for 'Function my_create_port in server.c'. Line 42 is highlighted in yellow and has a blue arrow pointing to it, indicating the current execution point. The code includes a socket call: `listenfd=socket(PF_INET,SOCK_STREAM,0);` followed by `fprintf(stderr,"listening filedescriptor %d\n",listenfd);`. The 'Action Points' pane at the bottom is empty.

The program after completing a socket call. Notice that the socket returned is filedescriptor 4.

W H I T E P A P E R

The screenshot displays the TotalView debugger interface. At the top, the menu bar includes File, Edit, View, Group, Process, Thread, Action Point, Instrumentation, Tools, Window, and Help. Below the menu is a toolbar with icons for Go, Halt, Kill, Restart, Next Step, Out, Run To, Prev, UnStep, Caller, Back To, and Live. The status bar shows 'Process 1 (15425): server (Stopped)' and 'Thread 1 (15425) (Stopped) <Stop Signal>'. The main window is divided into several panes:

- Stack Trace:** Shows the call stack with frames for `my_create_port` (FP=bfee9a88), `main` (FP=bfee9ab8), and `__libc_start_main` (FP=bfee9b18).
- Stack Frame:** Shows the local variables for the `my_create_port` function:
 - `port_number`: 0x0000d585 (54661)
 - `ipv4_addr_string`: 0x0804947a ->
 - Block "\$b1":**
 - `listenfd`: 0xb7e2f0e0 (-12098)
 - `queuedepth`: 0x08049489 (134517)
 - `my_addr`: (struct sockaddr_in)
 - `yes`: 0xb7e2eadc (-12098)
- Registers for the frame:** (Empty)
- Code Editor:** Shows the source code for `my_create_port` in `server.c`. The function signature is `int my_create_port(int port_number, char* ipv4_addr_string){`. The code includes variable declarations for `listenfd`, `queuedepth`, and `my_addr`. A blue triangle points to line 40, where `listenfd` is assigned the value of `yes`. The code continues with a `socket` call, a `fprintf` statement, an `if` statement checking for `listenfd == -1`, and an `exit(1)` call. Finally, `my_addr.sin_family` is set to `AF_INET` and `my_addr.sin_port` is set to `htons(port_number)`.

Here you can see what the value of the variable "listenfd" was before the socket call and the assignment. The debugger remembers where the "live" program is (shown with a blue triangle).

If you want to track down a particularly hard-to-find point in history you can "overshoot" your desired location (going into execution history before the program got where you want to go) and then construct a conditional expression (as you did in the previous section) using variables within the program to define the point you want to see. Then you let the program run forward through deterministic replay till it gets just where you want it.

This way you can make detailed observations of things going wrong in your program, after the fact.



W H I T E P A P E R

Conclusions

This article has looked at three different approaches to finding out what is going on inside of a program. Tracing program execution provides you with a way of looking at the behavior of your program over time. *Strace* provides easy access to limited information; print statements give you more detail about what your program is doing, but require recompilation; and *tvscript* provides flexibility along with detailed information but at the cost of some overhead.

The best way to explore the behavior of the program interactively is to use a graphical source code debugger. Remote desktop software makes it possible to have a graphical debugging experience even when working on applications running at distant sites. The control that a full-featured debugger gives you over program execution can be vital for solving hard-to-reproduce bugs.

Record and replay debugging provides a way to examine the execution history of a program that has been running. This allows you to troubleshoot your program by looking for errors with the benefit of clues from other parts of the execution history. Stepping forwards and backwards through complex algorithms or running right to the point where a wild pointer does its damage is easy to do with the TotalView Technologies ReplayEngine.

This paper started with a quote from Brian Kernighan about debugging. He and Rob Pike devote a chapter to debugging in “The Practice of Programming” in which they provide sound advice about approaches to troubleshooting software problems. They advocate tracing a program with print statements and using interactive debuggers only sparingly — “Blind probing with a debugger is not likely to be productive.” I have tried to highlight some practical advice for techniques that you will find much more satisfying than blind probing.

To find out more

Contact [TotalView Technologies](#).