RogueWave SOFTWARE

Accelerating Great Code

DETERMINISTICALLY TROUBLESHOOTING NETWORK DISTRIBUTED APPLICATIONS

Debugging is all about understanding what the software 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: "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's difficult to be objective. You build mental models of the program's execution 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's easier to be objective when reading when other people's code is one of the main reasons for code reviews.

In addition to the fundamental challenge of objectivity, 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 explains three different ways to approach debugging a client-server application: tracing, interactive source code debugging, and interactive replay debugging. The application referenced is a simple multithreaded, multimachine, memory status monitoring application written in C using the UNIX socket interface. The purpose is to 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're 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. It runs on a server application with the command:

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

or if your server is already running you can attach strace to it with a command line

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

where 3216 is the process id of the 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, "listen filedescriptor3\n,27) = 21
setsockopt(3,SOL_SOCKET,SO_REUSEADDR,[49},4)=0
bind(3,[sa_family=AF_INET,sin_ort=htons(54661),
Sin_addr=inet_addr("192.168.88.102")],16)=0
futex(0xb7ddcc2c, FUTEX_WAKE, 2147483647)=0
brk(0)         =0x804b000
brk(0x806c000)       =0x806x000
write(2, "bound to port 54661 on 192.168.8"…, 38)=38
listen(3,3)
write(2, "listening for 3 on port 54661 on"…, 48)=48
```

Read the man 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 that 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's 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.

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 identify the specific conditions under which output is generated.

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:

```
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're interested in. The script runs the program to these points and each time they are outputting 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 CodeDynamics product by Rogue Wave Software.

For example, say that you wanted to know about the sequence of your program in terms of two specific functions. The following command would create a trace log that tells you about each time the 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 creates a logfile in the current directory with a series of output blocks, one for each time the program executes one of the designated functions. Each output block gives the timestamp and a detailed backtrace for what happened 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:
!   06-25-2015 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=0xbfdac7z8 [/lib/tls/i686/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:

```
-create_actionpoint “#85=>print foreign_addr”
```

The output includes blocks like the following:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!Print
!
!Process:
!   ./server(Debugger Process ID: 1,System ID: 12110)
!Thread:
!   Debugger ID: 1.1, System ID: 3083946656
!Time Stamp:
!   06-26-2015 14:04:09
!Triggered from event:
!   actionpoint
!Results:
!   foreign_addr={
!         sin_family=0x0002(2)
!         sin_port=0x1fb6(8118)
!         sin_addr={
!               s_addr=0x6658a8c0*1717086400)
!         }
!         sin_zero=""
!   }
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Similar commands get other kinds of information from the process, such as local variable and register information.

The advantage to using tvscript to get this information from your program is that you can refine and refocus the information that you're looking for without having to recompile the program.

One disadvantage of using tvscript to get information from a program is that it incurs significant overhead. There is some startup time while the debugger 'reads' your program to understand how it's structured and plants the actionpoints. Each time the process reaches one of the actionpoints, the process is paused and the debugger has to take some action. Depending on how your network application is constructed, these pauses may mean that the behavior of the program might change (connections that aren't getting accepted at the usual rate may start to queue up). The interruptions are typically short so it is unlikely that connections will actually time out.

INTERACTIVE SOURCE CODE DEBUGGING

A very different approach to debugging a networked application is to use an interactive source code debugger, which offers a different view into the program than tracing does. With tracing you get to see a sequence of program events, but you have to decide beforehand which events you want to see. With an interactive source code debugger, you have the ability to pause the application and examine any variable or data structure in the program at the point where you paused it. This enables you to explore the interconnections between different data structures and follow any unexpected patterns you might find.

Using an interactive source code debugger on the server side of network applications is a bit harder than debugging the client because the server process typically runs in a machine room somewhere. You want to use remote desktop solutions like VNC, tunneled through ssh, to have a fast and secure way to get a graphical display for something like a debugger.

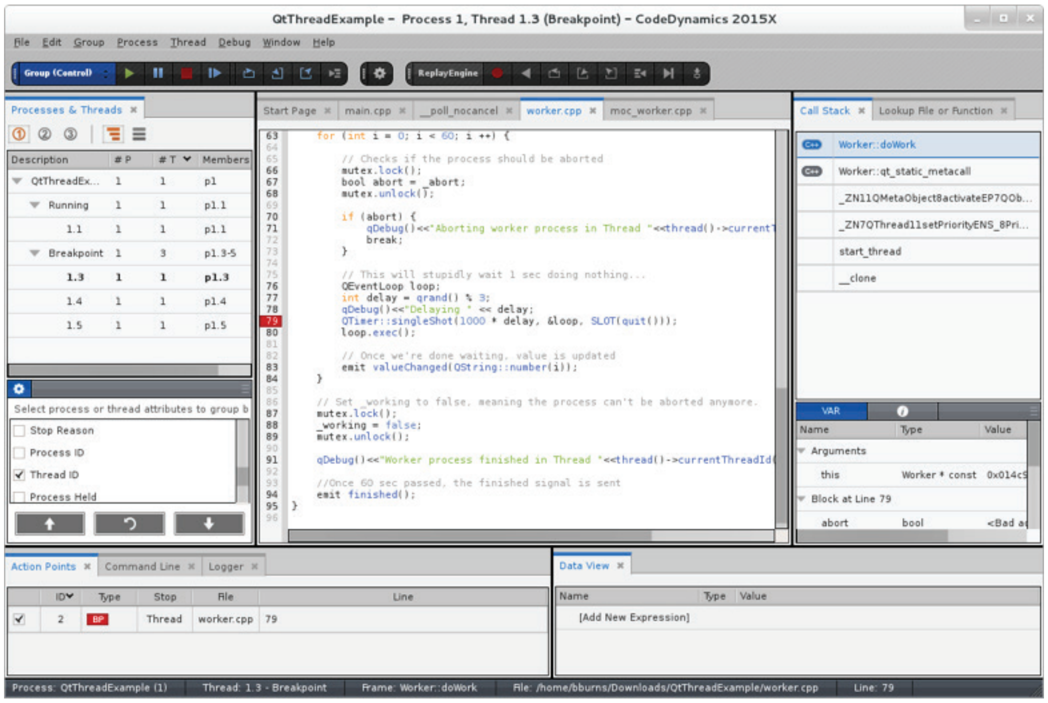
An interactive source code debugger gives the developer very precise ways to control the application. Two important basic examples of program control are single stepping and breakpoints.

Single stepping is an excellent way for you to follow the execution of a complex or unfamiliar routine. The debugger 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 the paths 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 to $stop, the debugger halts the application so you can examine it.


QtThreadExample – Process 1, Thread 1.3 (Breakpoint) – CodeDynamics 2015X

Description	# P	# T	Members
QtThreadEx...	1	1	p1
Running	1	1	p1.1
1.1	1	1	p1.1
Breakpoint	1	3	p1.3-5
1.3	**1**	**1**	**p1.3**
1.4	1	1	p1.4
1.5	1	1	p1.5

Select process or thread attributes to group b

- [ ] Stop Reason
- [ ] Process ID
- [x] Thread ID
- [ ] Process Held

```
    for (int i = 0; i < 60; i ++) {

        // Checks if the process should be aborted
        mutex.lock();
        bool abort = _abort;
        mutex.unlock();

        if (abort) {
            qDebug()<<"Aborting worker process in Thread "<<thread()->currentT
            break;
        }

        // This will stupidly wait 1 sec doing nothing...
        QEventLoop loop;
        int delay = qrand() % 3;
        qDebug()<<"Delaying " << delay;
        QTimer::singleShot(1000 * delay, &loop, SLOT(quit()));
        loop.exec();

        // Once we're done waiting, value is updated
        emit valueChanged(QString::number(i));
    }

    // Set _working to false, meaning the process can't be aborted anymore.
    mutex.lock();
    _working = false;
    mutex.unlock();

    qDebug()<<"Worker process finished in Thread "<<thread()->currentThreadId(

    //Once 60 sec passed, the finished signal is sent
    emit finished();
}
```

Call Stack:
Worker::doWork
Worker::qt_static_metacall
_ZN11QMetaObject8activateEP7QOb...
_ZN7QThread11setPriorityENS_8Pri...
start_thread
__clone

Name	Type	Value
Arguments		
this	Worker * const	0x014c9
Block at Line 79		
abort	bool	<Bad a

	ID	Type	Stop	File	Line
✓	2	BP	Thread	worker.cpp	79

Name	Type	Value
[Add New Expression]		

Process: QtThreadExample (1) Thread: 1.3 - Breakpoint Frame: Worker::doWork File: /home/bburns/Downloads/QtThreadExample/worker.cpp Line: 79

Figure 1: Breakpoints and focusing on specific threads in CodeDynamics

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 makes 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're using an interactive debugger on a network application that aggressively times out inactive connections, 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 interactive replay debugger to the point where it has done whatever you're interested in seeing, then go back and examine how that happened. Usually this means that you run the program until it crashes, hangs, or gives invalid data. Using an interactive replay debugger, you can examine the crashed or hung state, looking for clues about what might be wrong and examine recorded program history by searching backwards for the cause of the error.

This strategy has some extremely compelling advantages. First, because you aren't attempting to examine the process with the debugger while it's running, you only need to 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.

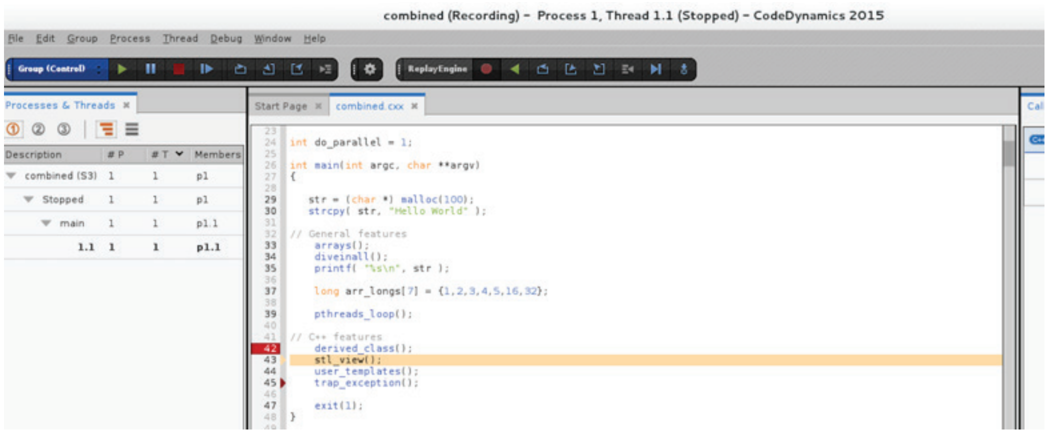
Another benefit, compared with traditional interactive debugging, is that you can use specific details gleaned from the crash or hang to guide your thinking while you're working. Looking through the execution history for the cause of an array bounds violation at a specific address is a much more narrowly-focused 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 reduces the pressure of the troubleshooting process.

The CodeDynamics ReplayEngine provides developers with just such a recording and replay mechanism. You load the ReplayEngine and debug server-dbg with the following command line:

```
CodeDynamics -replay ./server-dbg
```

Once the server program is running under the ReplayEngine, you can let it run forward to the point that it's 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 and data as they were one line earlier in execution history each time they're 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.


combined (Recording) – Process 1, Thread 1.1 (Stopped) – CodeDynamics 2015
File Edit Group Process Thread Debug Window Help
Group (Control)
ReplayEngine
Processes & Threads
Start Page
combined.cxx

Description	# P	# T	Members
combined (S3)	1	1	p1
Stopped	1	1	p1
main	1	1	p1.1
1.1	1	1	p1.1

```
int do_parallel = 1;

int main(int argc, char **argv)
{

   str = (char *) malloc(100);
   strcpy( str, "Hello World" );

// General features
    arrays();
    diveinall();
    printf( "%s\n", str );

    long arr_longs[7] = {1,2,3,4,5,16,32};

    pthreads_loop();

// C++ features
    derived_class();
    stl_view();
    user_templates();
    trap_exception();

    exit(1);
}
```

Figure 2: This example shows the ReplayEngine on and recording.

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 until it gets just where you want it.

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

CONCLUSIONS

We have 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.

Interactive 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 CodeDynamics 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." We have tried to highlight some practical advice for techniques that you'll find much more productive than blind probing.

RogueWave
SOFTWARE
Accelerating Great Code