Editor: **Diomidis Spinellis**
Athens University of Economics and Business, dds@aueb.gr

Differential Debugging

Diomidis Spinellis

IF ESTIMATING THE time needed for implementing some software is difficult, coming up with a figure for the time required to debug it is nigh on impossible. Bugs can lurk in the most obscure corners of the system, or even in the crevices of third-party libraries and components. Ask some developers for a time estimate, and don't be surprised if an experienced one snaps back, "I'll find the bug when I find the bug." Thankfully, there are some tools that allow methodical debugging, thereby giving you a sense of progress and a visible target. A method I've come to appreciate over the past few months is differential debugging. Under it, you compare a known good system with the buggy one, working toward the problem source.

Finding yourself in a situation with both a working and a buggy system is quite common. It might happen after you implement some new functionality, when you upgrade your tools or infrastructure, or when you deploy your system on a new platform. In all these cases, you might find yourself facing a system that should have been working but is behaving erratically for some unknown reason.

> Be ready to dig deeper, for this is where the bugs often lurk.

Differential debugging works because, despite what are our everyday experience suggests, deep down, computers are designed to work deterministically: the same inputs produce identical results. Probe sufficiently deep within a failing system, and, sooner or later, you'll discover the bug that causes it to behave differently from the working one.

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools

Observing Behavior

It's surprising how many times a system's failure reason stares us right in the eye, if only we would take the time to open its log file: "clients.conf: syntax error in line 92". In other cases, the reason is hidden deeper, so we need to increase the system's log verbosity in order to expose it. Many systems can adjust the amount of information they log through a command-line option, a configuration option, or even at runtime by sending them a suitable signal. So, increase the logging levels on both the known-good and the failing systems, take a snapshot of each system's log, and compare the two.

If the system doesn't offer a sufficiently detailed logging mechanism, you have to tease out its runtime behavior with an external tool. Besides general-purpose tools such as DTrace and SystemTap, some specialized tools I've found useful are those that trace calls to the operating system (strace, truss, ProcMon), those that trace calls to the dynamically linked libraries (ltrace, ProcMon), those that trace network packets (tcpdump, Wireshark), and those that allow the tracing of SQL database calls. Many Unix applications, such as the R Project for Statistical Computing, start their operation through complex shell scripts, which can misbehave in wonderfully obscure ways. You can trace their operation by passing the `-x` option to the corresponding shell. In most

0740-7459/13/$31.00 © 2013 IEEE

cases, the trace you obtain will be huge. Thankfully, modern systems have both the storage capacity to save the two logs and the CPU oomph to process and compare them.

Probing the Environment

When it comes to the environments in which your systems operate, your goal is to make the two environments as similar as possible. This will make your logs and traces easy to compare, or, if you're lucky, lead you directly to the cause behind the bug. Start with the obvious things, such as the program's inputs and command-line arguments. Verify, don't assume. Actually compare the input files of the two systems against each other, or, if they're big and far away, compare their MD5 sums.

> The log files differ in trivial ways, thus hiding the changes that matter.

Then focus on the code. Start by comparing the source code, but be ready to delve deeper, for this is where the bugs often lurk. Examine the dynamic libraries associated with each executable by using a command such as **ldd** (on Unix), or **dumpbin /dependents** (when using Visual Studio). See the defined and used symbols using **nm** (on Unix), **dumpbin /exports / imports** (Visual Studio), or **javap** (when developing Java code). If you're sure the problem lies in the code but can't see any difference, be prepared to dig even deeper, comparing the assembly code that the compiler generates.

But before you go to such an extreme, consider other elements that influence the setup of a program's execution. An underappreciated one is environmental variables, which even an unprivileged user can set in ways that can wreak havoc on a program's execution. Another is the operating system. Your application might be failing on an operating system that's a dozen years newer or older than the one where it's working okay. Also consider

the compiler, the development framework, third-party linked libraries, the browser (ah, the joy), the application server, the database system, and other middleware. How to locate the culprit in this maze is what we'll tackle next.

Techniques

Given that in most cases you'll be searching for a needle in a haystack, it makes sense to trim down the haystack's size. Invest the time to find the simplest possible test case in which the bug appears. (Making the needle—the buggy output—larger is rarely productive.) A svelte test case will make your job easier through shorter logs, traces, and processing times. Therefore, trim down the test case by gradually removing either elements from it or configuration options from the system until you arrive at the leanest possible setting that still exhibits the bug.

If the difference between the working and failing system lies in their source code, a useful method is to conduct a binary search through all the changes performed in between the two versions so as to pinpoint the culprit. Thus, if the working system is at version 100 and the failing one is at version 132, you'll first test version 116, and then, depending on the outcome, versions 108 or 124, and so on. The ability to perform such searches is one reason why you should always commit each change separately into your version control system. Thankfully, some version control systems offer a command that performs this search automatically; on Git, it's the `git bissect` command.

Another highly productive option is to process the two log files with Unix tools to find the difference related to the bug. The workhorse in this scenario is the `diff` command, which will display the differences between the two files. However, more often than not, the log files differ in trivial ways, thus hiding the changes that matter. There are many ways to filter these out. If the leading fields of each line contain varying elements, such as timestamps and process IDs, eliminate them with `cut` or `awk`. Select only the events that interest you—for instance, files that were opened—using a command like `grep 'open('`. Or eliminate noise lines (such as those thousands of annoying calls to get the system's time in Java programs) with a command such as `grep –v gettimeofday`. You can also eliminate parts of a line that don't interest you by specifying the appropriate regular expression in a `sed` command.

Finally, a more advanced technique that I've found particularly useful if the two files aren't ordered in a way in which `diff` can be productive is to extract the fields that interest you, sort them, and then find the elements that aren't common in the two sets. Consider the task of finding which files were only opened in only one of two trace files t1 and t2. In the Unix bash shell, the corresponding incantation for comparing the second field (the file name) in lines containing the string `open(` in would be

```
comm -3 <(awk '/open\(/{print $2}' t1 | sort) \
        <(awk '/open\(/{print $2}' t2 | sort)
```

Now brush up your Unix skills by reading the November/December 2005 installment of this column ("Working with Unix Tools," pp. 11–12), and then go catch some bugs!

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the books *Code Reading* and *Code Quality: The Open Source Perspective* (Addison-Wesley, 2003 and 2006). Contact him at dds@aueb.gr.

See www.computer.org/software-multimedia for multimedia content related to this article.

IEEE Software

HOW TO REACH US

WRITERS

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

LETTERS TO THE EDITOR

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

ON THE WEB

www.computer.org/software

SUBSCRIBE

www.computer.org/software/subscribe

SUBSCRIPTION CHANGE OF ADDRESS

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software.*

MEMBERSHIP CHANGE OF ADDRESS

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

MISSING OR DAMAGED COPIES

If you are missing an issue or you received a damaged copy, contact help@computer.org.

REPRINTS OF ARTICLES

For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

REPRINT PERMISSION

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.