

Article development led by **acmqueue**
queue.acm.org

Many modern dynamic languages lack tools for understanding complex failures.

BY DAVID PACHECO

Postmortem Debugging in Dynamic Environments

DESPITE THE BEST efforts of software engineers to produce high-quality software, inevitably some bugs escape even the most rigorous testing process and are first encountered by end users. When this happens, such failures must be understood quickly, the underlying bugs fixed, and deployments patched to avoid another user (or the same one) running into the same problem again. As far back as 1951, the dawn of modern computing, Stanley Gill⁶ wrote that “some attention has, therefore, been given to the problem of dealing with mistakes after the program has been tried and found to fail.” Gill went on to describe the first use of “the post-mortem technique” in software, whereby the running program was modified to record important system state as it ran so that the programmer could later understand what happened and why the software failed.

Since then, postmortem debugging technology has been developed and used in many different systems, including all major consumer and enterprise operating systems, as well as the native execution environments on those systems. These environments make up much of today’s core infrastructure, from the operating systems that underlie every application to core services such as Domain Name System (DNS), and thus form the building blocks of nearly all larger systems. To achieve the high levels of reliability expected from such software, these systems are designed to restore service quickly after each failure while preserving enough information that the failure itself can later be completely understood.

While such software was historically written in C and other native environments, core infrastructure is increasingly being developed in dynamic languages, from Java over the past two decades to server-side JavaScript over the past 18 months. Dynamic languages are attractive for many reasons, not least of which is that they often accelerate the development of complex software.

Conspicuously absent from many of these environments, however, are facilities for even basic postmortem debugging, which makes understanding production failures extremely difficult. Dynamic languages must bridge this gap and provide rich tools for understanding failures in deployed systems in order to match the reliability demanded from their growing role in the bedrock of software systems.

To understand the real potential for sophisticated postmortem analysis tools, we first review the state of debugging today and the role of postmortem analysis tools in other environments. We then examine the unique challenges around building such tools for dynamic environments and the state of such tools today.

Debugging in the Large

To understand the unique value of postmortem debugging, it is worth



ILLUSTRATION BY GARY NEILL

examining the alternative. Both native and dynamic environments today provide facilities for *in situ* debugging, or debugging faulty programs while they're still running. This typically involves attaching a separate debugger program to the faulty program and then directing execution of the faulty program interactively, instruction by instruction or using breakpoints. The user thus stops the program at various points to inspect program state in or-

der to figure out where it goes wrong. Often this process is repeated to test successive theories about the problem.

This technique can be very effective for bugs that are readily reproducible, but it has several drawbacks. First, the act of stopping a program often changes its behavior. Bugs resulting from unexpected interactions between parallel operations (such as race conditions) can be especially challenging to analyze this way because the timing of

various events is significantly affected by the debugger itself. More importantly, *in situ* debugging is often untenable on production systems: many debuggers rely on an unoptimized debug build that's too slow to run in production; engineers often do not have access to the systems where the program is running (as in the case of most mobile and desktop applications and many enterprise systems); and the requisite debugging tools are often not

available on those systems anyway.

Even for cases where engineers can access the buggy software with the tools they need, pausing the program in the debugger usually represents an unacceptable disruption of production service and an unacceptable risk that a fat-fingered debugger command might cause the program to crash. Administrators often cannot take the risk of downtime in order to understand a failure that caused a previous outage. More importantly, they should not have to. Even in 1951 Gill cited the “extravagant waste of machine time involved” in concluding that “single-[step] operation is a useful facility for the maintenance engineer, but the programmer can only regard it as a last resort.”

The most crippling problem with *in situ* debugging is it can only be used to understand reproducible problems. Many production issues are either very rare or involve complex interactions of many systems, which are often very difficult to replicate in a development environment. The rarity of such issues does not make them unimportant: quite the contrary, an operating system crash that happens only once a week can be extremely costly in terms of downtime, but any bug that can be made to occur only once a week is very difficult to debug live. Similarly, a fatal error that occurs once a week in an application used by thousands of people may result in many users hitting the bug each day, but engineers cannot attach a debugger on every user’s system.

So-called `printf` debugging is a

common technique for dealing with the reproducibility issue. In this approach, engineers modify the software to log bits of relevant program state at key points in the code. This causes data to be collected without human intervention so it can be examined after a problem occurs to understand what happened. By automating the data collection, this technique usually results in significantly less impact to production service because when the program crashes, the system can immediately restart it without waiting for an engineer to log in and debug the problem interactively.

Extracting enough information about fatal failures from a log file is often very difficult, however, and frequently it is necessary to run through several iterations of inserting additional logging, deploying the modified program, and examining the output. This, too, is untenable for production systems since ad hoc code changes are often impractical (in the case of desktop and mobile applications) or prohibited by change control policies (and common sense).

The solution is to build a facility that captures *all* program state when the program crashes. In 1980 Douglas R. McGregor and Jon R. Malone⁹ of the University of Strathclyde in Glasgow observed that with this approach “there is virtually no runtime overhead in either space or speed” and “no extra trace routines are necessary,” but the facility “remains effective when a program has passed into production

use.” Most importantly, after the system saves all the program state, it can restart the program immediately to restore service quickly. With such systems in place, even rare bugs can often be root-caused and fixed based on the first occurrence, whether in development, test, or production. This enables software vendors to fix bugs before too many users encounter them.

To summarize, in order to root-cause failures that occur anywhere from development to production, a postmortem debugging facility must satisfy several constraints:

- ▶ Application software must not require modifications that cannot be used in production in order to support postmortem debugging, such as unoptimized code or additional debug data that would significantly impact performance (or affect correctness at all).

- ▶ The facility must be always on: It must not require an administrator to attach a debugger or otherwise enable postmortem support before the problem occurs.

- ▶ The facility must be fully automatic: It should detect the crash, save program state, and then immediately allow the system to restart the failed component to restore service as quickly as possible.

- ▶ The dump (saved state) must be comprehensive: a stack trace, while probably the single most valuable piece of information, very often does not provide sufficient information to root-cause a problem from a single occurrence. Usually engineers want both global state and each thread’s state (including stack trace and each stack frame’s arguments and variables). Of course, there’s a wide range of possible results in this dimension; the “constraint” (such as it is) is that the facility must provide enough information to be useful for nontrivial problems. The more information that can be included in the dump, the more likely engineers will be able to identify the root cause based on just one occurrence.

- ▶ The dump must be transferable to other systems for analysis. This allows engineers to analyze the data using whatever tools they need in a familiar environment and obviates the need for engineers to access production systems in many cases.

Figure 1. A simple MDB example.

```
$ mdb core
Loading modules: [ ld.so.1 ]
> ::status
debugging core file of example1 (32-bit) from solaron
file: /export/home/dap/tmp/example1
initial argv: ./example1
threading model: native threads
status: process terminated by SIGSEGV (Segmentation Fault), addr=10

> ::walk thread | ::findstack -v
stack pointer for thread 1: 8047b98
[ 08047b98 func+0x20() ]
 08047bbc main+0x21(1, 8047bdc, 8047be4)
 08047bd0 _start+0x80(1, 8047cc4, 0, 8047ccf, 8047cdc, 8047ced)

> func+0x20::dis
...
func+0x20:          movl    $0x0, (%eax)
...
```

Postmortem Debugging in Native Environments

To understand the potential value of postmortem debugging in dynamic languages, it is also helpful to examine those areas where postmortem analysis techniques are well developed and widely used. The best examples are operating systems and their native execution environments. Historically this software has comprised core infrastructure; failures at this level of the stack are often very costly either because the systems themselves are necessary for business-critical functions (as in the case of operating systems on which business-critical software is running) or because they are relied upon by business-critical systems upstack (as in the case of infrastructure services such as DNS).

Most modern operating systems can be configured so that when they crash, they immediately save a “crash dump” of all of their state and then reboot. Likewise, these systems can be configured so that when a user application crashes, the operating system saves a “core dump” of all program state to a file and then restarts the application. In most cases, these mechanisms allow the operating system or user application to return to service quickly while preserving enough information to root-cause the failure later.

As an example, let’s look at core dumps under Illumos, an open source Solaris-based system. Take the following broken program:

```

1 int
2 main(int argc, char *argv[])
3 {
4     func();
5     return (0);
6 }
7
8 int
9 func(void)
10 {
11     int ii;
12     int *ptrs[100];
13
14     for (ii = -1; ii < 100; ii++)
15         *(ptrs[ii]) = 0;
16
17     return (0);
18 }

```

This simple program has a fatal flaw:

Figure 2. Analyzing thread stacks.

```

> ::stacks -m zfs
THREAD      STATE      SOBJ      COUNT
ffffff0007c0fc60 SLEEP      CV        2
    swtch+0x147
    cv_wait+0x61
    txg_thread_wait+0x5f
    txg_quiesce_thread+0x94
    thread_start+8

ffffff0007f51c60 FREE      <NONE>    1
    cpu_decay+0x2f
    bitset_atomic_del+0x38
    apic_setspl+0x5c
    do_splx+0x50
    disp_lock_exit+0x55
    cv_signal+0x96
    taskq_dispatch+0x351
    zio_taskq_dispatch+0x6b
    zio_interrupt+0x1a
    vdev_disk_io_intr+0x6b
    biodone+0x84
    dadk_iodone+0xe7
    dadk_pktcb+0xc6
    ata_disk_complete+0x119
    ata_hba_complete+0x38
    ghd_doneq_process+0xb3
    0x16
    dispatch_softint+0x3f

ffffff0007b25c60 SLEEP      CV        1
    swtch+0x147
    cv_timedwait+0xba
    arc_reclaim_thread+0x17b
    thread_start+8

ffffff0007b2bc60 SLEEP      CV        1
    swtch+0x147
    cv_timedwait+0xba
    l2arc_feed_thread+0xa5
    thread_start+8

ffffff0009b95c60 SLEEP      CV        1
    swtch+0x147
    cv_timedwait+0xba
    txg_thread_wait+0x7b
    txg_sync_thread+0x114
    thread_start+8

ffffff01e26d08e0 SLEEP      CV        1
    swtch+0x147
    cv_wait+0x61
    txg_wait_synced+0x7f
    spa_sync_allpools+0x76
    zfs_sync+0xce
    vfs_sync+0x9c
    syssync+0xb
    sys_syscall32+0x101

ffffff0007c15c60 SLEEP      CV        1
    swtch+0x147
    cv_wait+0x61
    zio_wait+0x5d
    dsl_pool_sync+0xe1
    spa_sync+0x32a
    txg_sync_thread+0x265
    thread_start+8

```

while trying to clear each item in the `ptrs` array at lines 14–15, it clears an extra element before the array (where `ii = -1`). When running this program, you see:

```
$ gcc -o example1 example1.c
$ ./example1
Segmentation Fault (core dumped)
```

and the system generates a file called `core`. The Illumos modular debugger (MDB) shown in Figure 1 can help in examining this file.

MDB's syntax may seem arcane to new users, but this example is rather basic. First the `::status` command produces a summary of what happened: the process was terminated as a result of a segmentation fault attempting to access memory address `0x10`. Next the `::walk thread | ::findstack -v` command is used to examine thread stacks (in this case, just one), and it shows that the program died in function `func` at offset `0x20` in the program text. Then the file dumps out this instruction to see that the process died on the store of 0 into the address contained in register `%eax`.

While this example is admittedly contrived, it illustrates the basic method of postmortem debugging. Note that unlike *in situ* debugging, this method scales well with the complexity of the program being debugged. If instead of one thread in one process there were thousands of threads across dozens of components (as in the case of an operating system), a comprehensive dump would include information about all of them. The next challenge would be making sense of so much information, but root-causing the bug is at least tractable because all the information is available.

In such situations, the next step is to build custom tools for extracting, analyzing, and summarizing specific component state. A comprehensive postmortem facility enables engineers to build such tools. For example, `gdb` supports user-defined macros. These macros can be distributed with the source code so that all developers can use them both *in situ* (by attaching `gdb` to a running process) and postmortem (by opening a core file with `gdb`). The Python interpreter, for example, provides such macros, allowing both inter-

preter and native module developers to pick apart the C representations of Python-level objects.

MDB takes this idea to the next level: it was designed specifically around building custom tools for understanding specific components of the system both *in situ* and postmortem. On Illumos systems, the kernel ships with MDB modules that provide more than 1,000 commands to iterate and inspect various components of the kernel. Among the most frequently used is the `::stacks` command, which iterates all kernel threads, optionally filters them based on the presence of a particular kernel module or function in the stack trace, and then dumps out a list of unique thread stacks sorted by frequency. Figure 2 offers an example from a system doing some light I/O.

This invocation collapsed the complexity of more than 600 threads on this system to only about seven unique thread stacks that are related to the ZFS file system. You can quickly see the state of the threads in each group (e.g., sleeping on a condition variable) and examine a representative thread for more information. Dozens of other operating-system components deliver their own MDB commands for inspecting specific component state, including the networking stack, the NFS server, `DTrace`, and ZFS.

Some of these higher-level analysis tools are quite sophisticated. For example, the `::typegraph` command³ analyzes an entire production crash dump (without debug data) and constructs a graph of object references and their types. With this graph, users can query the type of an arbitrary memory object. This is useful for understanding memory corruption issues, where the main problem is identifying which component overwrote a particular block of memory. Knowing the type of the corrupting object narrows the investigation from the entire kernel to the component responsible for that type.

Such tools are by no means limited to production environments. On most systems, it is possible to generate a core dump from *running* processes too, which make core-dump analysis attractive during development as well. When testers or other engineers file bugs on application crashes, it is often easier to have them include a

core dump than to try to guess what steps they took that led to the crash and then reproduce the problem from those steps. Examining the core dump is also the only way to be sure the problem you found is the same one the bug reporter encountered.

Higher-level dump analysis tools can be built explicitly for development as well. `Libumem`, a drop-in replacement for `malloc(3c)` and friends, provides (among other features) an MDB module for iterating and inspecting objects related to the allocator. Combined with an optional feature to record stack traces for each allocator operation, the `::findleaks` MDB command can be used to identify various types of memory leaks very quickly without having added any explicit support for this in the application itself. The `::findleaks` command literally prints out a list of leaked objects and the stack trace from which each one was allocated—pointing directly to the location of each leak. `Libumem` is based on the kernel memory allocator, which provides many of the same facilities for the kernel.²

Postmortem Debugging in Dynamic Environments


While operating-system and native environments have highly developed facilities for handling crashes, saving dumps, and analyzing them postmortem, the problem of postmortem analysis (and software observability more generally) is far from solved in the realm of dynamic environments such as Java, Python, and JavaScript. In the past postmortem analysis was arguably less critical for these languages because crashes in these environments are less significant: most end-user applications save work frequently anyway, and the operating system or browser will often restart the application after a crash. These crashes still represent disruptions to the user experience, however, and postmortem debugging is the only hope of understanding such failures.

More importantly, dynamic languages such as Node.js are exploding in popularity as building blocks for larger distributed systems, where what might seem like a minor crash can cause cascading failures up the stack. As a result, just as with operating systems and core services, fully understanding each failure is essential to achieving the levels


of reliability expected of such foundational software.

Providing a postmortem facility for dynamic environments, however, is not easy. While native programs can leverage operating-system support for core dumps, dynamic languages must present postmortem state using the same higher-level abstractions with which their developers are familiar. A postmortem environment for C programs can simply present a list of global symbols, pointers to thread stacks, and all of a process's virtual memory (all of which the operating system has to maintain anyway), but a similar facility for Java must augment (or replace) these with analogous Java abstractions. When Java programs crash, Java developers want to look at Java thread stacks, local variables, and objects, not (necessarily) the threads, variables, and raw memory used by the Java virtual machine (JVM) implementation. Also, because programs in dynamic languages run inside an interpreter or VM, when the user program "crashes," the interpreter or VM itself does not crash. For example, when a Python program uses an undefined variable (the C equivalent of a NULL pointer), the interpreter detects this condition and gracefully exits. Therefore, to support postmortem debugging, the interpreter would need to trigger the core-dump facility explicitly, not rely on the operating system to detect the crash.

In some cases, presenting useful postmortem state requires formalizing abstractions that do not exist explicitly in the language at all. JavaScript presents a particularly interesting challenge in this regard. In addition to the usual global state and stack details, JavaScript maintains a pending event queue, as well as a collection of events that may happen later—both of which exist only as functions with associated context that will be invoked at some later time by the runtime. For example, a Web browser might have many outstanding asynchronous HTTP requests. For each one, there is a function with associated context that may not be reachable from the global scope, and so would not be included in a simple dump of all global state and thread state. Nevertheless, understanding which of these requests are outstanding and what state is associ-



The most crippling problem with *in situ* debugging is it can only be used to understand reproducible problems.



ated with them may very well be critical to understanding a fatal failure.

This problem is even more acute with Node.js on the server, which is frequently used to manage thousands of concurrent connections to many different types of components. A single Node program might have hundreds of outstanding HTTP requests, each one waiting on a database query to complete. The program may crash while processing one of the database query results because it encountered an invalid database state resulting from one of the other outstanding queries. Such problems beg for postmortem debugging because each instance is seen relatively rarely; they are essentially impossible to understand from just a stack trace, but they can often be identified from the first occurrence, given enough information from the time of the crash. The challenge is presenting information about outstanding asynchronous events (that is, callbacks that will be invoked at some future time) in a meaningful way to JavaScript developers, who generally do not have direct access to the event queue or the collection of outstanding events; these abstractions are implicit in the underlying APIs, so exposing this requires first figuring out how to express these abstractions.

Finally, user-facing applications have the additional problem of transferring postmortem state from the user's computer to developers who can root-cause the bug (while preserving user privacy). As Eric Schrock¹¹ details, this problem remains largely unsolved for one of the most significant dynamic environments today: the JavaScript Web application. There is no browser-based facility for automatically uploading postmortem program state back to the server.

Despite these difficulties, some dynamic environments do provide postmortem facilities. For example, the Oracle Java HotSpot VM supports extracting Java-level state from JVM native core dumps. When the JVM crashes, or when a core file is manually created using operating system tools such as `gcore(1)`, you can use the `jdb(1)` tool to examine the state of the Java program (rather than the JVM itself) when the core file was generated. The core file can also be processed by a tool called `jmap(1)` to create a Java heap dump that can in turn be analyzed us-

ing several different programs.

Such facilities are only a start, however: setting up an application to trigger a core dump on crash in the first place is nontrivial. Additionally, these facilities are very specific to the HotSpot VM. There's an active Java Community Specification proposal for a common API to access debugging information, but at the time of this writing this project is stalled pending clarity about Oracle's commitment to the project.⁸


While the Java facility has several important limitations, many other dynamic environments do not appear to have postmortem facilities at all—at least not any that meet the constraints just described.

Python¹⁰ and Ruby⁴ each has a facility called a postmortem debugger, but these refer to starting a program under a debugger and having the program break into an interactive debugger session when the program crashes. This is not suitable for production for several reasons, not least of which is that it is not fully automatic. As described earlier, it is not tenable to interrupt production service while an engineer logs in to diagnose a problem interactively.


Erlang⁵ provides a rich crash-dump facility for the Erlang runtime itself. It works much like a native crash dump in that on failure it saves a comprehensive state dump to a file and then exits, allowing the operating system to see the program has exited and restart it immediately. The crash dump file can then be analyzed later.

The bash shell¹ is interesting because its deployment model is so different even from other dynamic environments. Bash provides a mechanism called `xtrace` for producing a comprehensive trace file describing nearly every expression that the shell evaluates as part of execution. This is very useful for understanding shell script failures but can produce a lot of output even for simple scripts. The output grows unbounded as the program runs, which would normally make it untenable for production use in servers or applications, but since most bash scripts have very finite lifetimes, this mechanism is an effective postmortem facility as long as the output can be reasonably stored and managed (that is, automatically deleted after successful runs).

JavaScript, unlike many of the



There is no browser-based facility for automatically uploading postmortem program state back to the server.



above languages, is widely deployed under several completely different runtime environments such as Mozilla's SpiderMonkey, Google's V8 (used in both Chrome and Node.js) and the WebKit JavaScript engine. Although JavaScript *in situ* debugging facilities have improved substantially in recent years in the form of improved browser support for runtime program inspection, there remains no widely used postmortem facility for JavaScript.

A Primitive Postmortem Facility for Node.js

Despite the lack of JavaScript language support, we have developed a crude but effective postmortem debugging facility for use in Joyent's Node.js production deployments. Recall that Node typically runs on a server rather than a Web browser and is commonly used to implement services that scale to hundreds or thousands of network connections. We use the following primitives provided by Node and the underlying V8 virtual machine to construct a simple implementation:

- ▶ An `uncaughtException` event, which allows a program to register a function to be invoked when the program throws an exception that bubbles all the way to the top level (that would normally cause the program to crash).
- ▶ Built-in mechanisms for serializing/deserializing simple JavaScript objects as a text string (`JSON.stringify()` and `JSON.parse()`).
- ▶ Synchronous functions for writing to files.

The first challenge is actually identifying which state to dump. JavaScript provides a way to introspect global state, but Node.js programs that declare variables do not use global state *per se*. What looks like the top-level scope is actually contained inside a function scope, and function scopes cannot be introspected. To work around this, programs using our postmortem facility must explicitly register debugging state ahead of time. While this solution is deeply unsatisfying because it is always difficult to know ahead of time what information would be useful to have when debugging, it has proved effective in practice because each of our programs essentially just instantiates a singleton object representing the program itself and then registers that with the post-

mortem facility. Most relevant program state is referenced by this pseudo-global object in one way or another.

The next challenge is serializing circular objects. `JSON.stringify()` does not support this for obvious reasons, so our implementation avoids this issue by pruning all circular references before serializing the debug object. While this makes it harder to find information in the dump, we know that at least one copy of every object will be present somewhere in the dump.

Given all this, the implementation is straightforward: on the uncaughtException event, we prune circular references from the debug state, serialize it using the built-in `JSON.stringify()` routine, and save the result to disk in a file called `core`. To analyze the core file, we use a tool that reads `core` using `JSON.parse()` and presents the serialized state for engineers to examine. The implementation is open source and available on GitHub.⁷

In addition to the implementation challenges just described, this approach has several significant limitations. First, it can save only state that programmers can register ahead of time, but as already discussed, there is a great deal of other important state inside a JavaScript program such as function arguments in the call stack and the contexts associated with pending and future events, none of which is reachable from the global scope.

Second, since the entire point of this system is to capture program state in the event of a crash, it must be highly reliable. This implementation is robust to most runtime failures, but it still requires additional memory first to execute the dump code and to serialize the program state. The additional memory could easily be as large as the whole heap, which makes it untenable for failures resulting from memory pressure—a common cause of failures in dynamic environments.

Third, because the implementation removes circular references before serializing the program data, the resulting dump is more difficult to browse, and the facility cannot support dumps that are not intended for postmortem analysis (such as live dumps).

Despite these deficiencies, this implementation has proved quite effective because it meets the require-

ments set forth earlier: it is always-on in production, fully automatic, the result is transferable to other systems for analysis, and it is comprehensive enough to solve complex problems. To address many of the scope, robustness, and richness problems described here, however, and to provide such a facility for all users of a language, the postmortem facility must be incorporated into the VM itself. Such an implementation would work similarly in principle, but it could include absolutely all program state, be made to work reliably in the face of failure of the program itself, stream the output to avoid using much additional memory, and use a format that preserves the underlying memory structures to ease understanding of the dump. Most importantly, including tools for postmortem analysis out of the box would go a long way toward the adoption of postmortem techniques in these environments.

Conclusion

Postmortem debugging facilities have long enabled operating-system engineers and native-application developers to understand complex software failures from the first occurrence in deployed systems. Such facilities form the backbone of the support process for enterprise systems and are essential for software components at the core of a complex software environment. Even simple platforms for recording postmortem state enable engineers to develop sophisticated analysis tools that help them to quickly root-cause many types of problems.

Meanwhile, modern dynamic languages are growing in popularity because they so effectively facilitate rapid development. Environments such as Node.js also promote programming models that scale well, particularly in the face of latency bubbles. This is becoming increasingly important in today's real-time systems.

Postmortem debugging for dynamic environments is still in its infancy. Most such environments, even those considered mature, do not provide any facility for recording postmortem state, let alone tools for higher-level analysis of such failures. Those tools that do exist are not first-class tools in their respective environments and so are not widely used. As dynamic languages grow in

popularity for building critical software components, this gap is becoming increasingly important. Languages that ignore the problems associated with debugging production systems will increasingly be relegated to solving simpler, well-confined, well-understood problems, while those that provide rich tools for understanding failure postmortem will form the basis of the next generation of software bedrock.

Acknowledgments

Many thanks to Bryan Cantrill, Peter Memishian, and Theo Schlossnagle for reviewing earlier drafts of this article and to Adam Cath, Ryan Dahl, Robert Mustacchi, and many others for helpful discussions on this topic. ☐

Related articles on queue.acm.org

Erlang for Concurrent Programming

Jim Larson

<http://queue.acm.org/detail.cfm?id=1454463>

Orchestrating an Automated Test Lab

Michael Donat

<http://queue.acm.org/detail.cfm?id=1046946>

Scripting Web Services Prototypes

Christopher Vincent

<http://queue.acm.org/detail.cfm?id=640158>

References

1. Bash Reference Manual (2009); <https://www.gnu.org/s/bash/manual/bash.html>.
2. Bonwick, J. The slab allocator: An object-caching kernel memory allocator. *Usenix Summer 1994 Technical Conference*.
3. Cantrill, B.M. Postmortem object type identification. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*. (2003)
4. Debugging with ruby-debug. 2011; http://bashdb.sourceforge.net/ruby-debug.html#Post_002dMortem-Debugging.
5. Erlang Runtime System Application User's Guide, version 5.8.4. 2011. How to interpret the Erlang crash dumps; http://www.erlang.org/doc/apps/erts/crash_dump.html.
6. Gill, S. The diagnosis of mistakes in programmes on the EDSAC. In *Proceedings of the Royal Society A 206* (1951), 538–554.
7. GitHub Project. 2011; <https://github.com/joyent/node-panic>
8. Incubator Wiki. March 2011 Board reports. Kato Project; <http://wiki.apache.org/incubator/March2011>.
9. McGregor, D.R., Malone, J.R. Stabdump—A dump interpreter program to assist debugging. *Software Practice and Experience* 10, 4 (1980), 329–332.
10. Python Standard Library. Python v2.7.2 documentation pdb—the Python debugger; 2011; <http://docs.python.org/library/pdb.html>.
11. Schrock, E. Debugging AJAX in production. *ACM Queue* 7, 1 (2009); <http://queue.acm.org/detail.cfm?id=1515745>.

David Pacheco is an engineer at Joyent where he leads the design and implementation of Cloud Analytics, a real-time Node.js/DTrace-based system for visualizing server and application performance in the cloud. Previously a member of the Sun Microsystems Fishworks team, he worked on several features of the Sun Storage 7000 appliances.

© 2011 ACM 0001-0782/11/12 \$10.00