Extracted from:

Release It!
Design and Deploy Production-Ready Software

The Pragmatic Programmers

Release It!

Design and Deploy Production-Ready Software

Michael T. Nygard

Printed in the United States of America.

ISBN-10: 0-9787392-1-3

ISBN-13: 978-0-9787392-1-8

Printed on acid-free paper.

P3.0 printing, February 2009

Version: 2009-3-27

5.4 Steady State

Roget's Thesaurus (3rd ed.) offers the following definition for the word *fiddling*: "To handle something idly, ignorantly, or destructively." It offers helpful synonyms such as *fool*, *meddle*, *tamper*, *tinker*, and *monkey*. Fiddling is often followed by the "ohnosecond": that very short moment in time during which you realize that you have pressed the wrong key and brought down a server, deleted vital data, or otherwise damaged the peace and harmony of stable operations. Every single time a human touches a server is an opportunity for unforced errors.[6]

It's best to keep people off of production systems to the greatest extent possible. If the system needs a lot of crank-turning and hand-holding to keep running, then administrators develop the habit of staying logged in all the time. This inevitably leads to fiddling. To that end, the system should be able to run indefinitely without human intervention.

Don't encourage fiddling. Systems should run indefinitely without intervention.

Unless the system is crashing every day (in which case, look for the presence of the stability antipatterns), the most common reason for logging in will probably be cleaning up log files or purging data.

Any mechanism that accumulates resources (whether it is log files in the filesystem, rows in the database, or caches in memory) is like the bucket from those high-school calculus problems. The bucket fills up at a certain rate, based on the accumulation of data. It must be drained at the same rate, or greater, or it will eventually overflow. When this bucket overflows, bad things happen: servers go down, databases get slow or throw errors, response times head for the stars. The Steady State pattern says, for every mechanism that accumulates a resource, some other mechanism must recycle that resource. You'll look at several types of sludge that can accumulate and how to avoid the need for fiddling.

Data Purging

It certainly seems like a simple enough principle. Computing resources are always finite; therefore, you cannot continually increase consumption without limit. Still, in the rush of excitement about rolling out a

6. I know of one incident in which an engineer, attempting to be helpful, observed that a server's root disk mirror was out of sync. He executed a command to "resilver" the mirror, bringing them back into synchronization. Unfortunately, he made a typo and synced the good root disk from the new, totally empty drive that had just been swapped in to replace a bad disk, thereby instantly annihilating the operating system on that server.

new killer application, the next great mission-critical, bet-the-company whatever, data purging always gets the short end of the stick. It certainly doesn't demo as well as...well, anything else in the world demos better than purging, really. It sometimes seems that you'll be lucky if the system ever runs at all in the real world. The notion that it will run long enough to accumulate too much data to handle seems like a "high-class problem"—the kind of problem you'd love to have.

Nevertheless, someday your little database will grow up. When it hits the teenage years—about two in human years—it will get moody, sullen, and resentful. In the worst case, it will start undermining the whole system (and it will probably complain that nobody understands it, too).

Data purging never makes it into the first release, but it should.

The most obvious symptom of data growth will be steadily increasing I/O rates on the database servers. You may also see increasing latency at constant loads.

Data purging is nasty, detail-oriented work. Referential integrity constraints in the database are half the battle. It can be very difficult to cleanly remove obsolete data without leaving orphaned rows. The other half of the battle is ensuring that applications still work once the data is gone.

For example, will the applications work if items are missing from the middle of collections? (Hint: under Hibernate, they won't!) As a consequence, data purging always gets left until after the first release is out the door. The thin rationale is, "We've got six months after launch to implement purging." (Somehow, they always say "six months." It's kind of like a programmer's estimate of "two weeks.")

Of course, after launch, there are always emergency releases to fix critical defects or add "must-have" features from marketers tired of waiting for the software to be done. The first six months can slip away pretty quickly, but when that first release launches, a fuse is lit.

Purging in Practice

I gave a talk at OTUG[7] that eventually led to this book. I was thrilled to see most of my project's team in attendance, including the sponsor. When I was presenting this very issue about the importance of data purging and

7. The Object Technology Users' Group in the Twin Cities of Minneapolis and St. Paul, Minnesota. See http://www.otug.org/.

its usual neglect, I could see everyone from my project nodding along (with their eyes open!). So you can imagine my chagrin when we launched our first release without data purging!

We eventually implemented a very thorough purge process, based on measuring our shortest fuse to see how long we had. It ended up being a very close thing when we rolled out the first iteration of purging, which took care of the highest-volume data items. That bought us time. Subsequent releases rolled out more rigorous routines for lower-volume accumulations.

Another type of sludge you will commonly encounter is old log files.

Log Files

Last week's log files are about as interesting as a book full of actuarial tables. A few rare, special people would be delighted to pore through them. The rest of us regard them as warmly as the dumpster behind a sushi restaurant. Last month's log files are even worse. The main thing these old log files do is take up valuable disk space.

Left unchecked, however, they become more than just a meaningless pile of uninterpreted bytes. When log files grow without bound, they will eventually fill up their containing filesystem. Whether that's a volume set aside for logs, the root disk, or the application installation directory (I hope not), it means trouble. When log files fill up the filesystem, they jeopardize stability. That's because of the different negative effects that can occur when the filesystem is full. On a UNIX system, the last 5% to 10% percent (depending on the configuration of the filesystem) of space is reserved for root. That means an application will start getting I/O errors when the filesystem is 90% or 95% full. Of course, if the application is running as root, then it can consume the very last byte of space. On a Windows system, an application can always use the very last byte. In either case, the operating system will report errors back to the application. For a Java-based system, that means java.io.IOException. For .NET, it's a System.IO.IOException. For C, it's an errno value of ENOSPC. (Show of hands, please: Who checks errno for ENOSPC after *every* call to write()?) In almost every case, logging libraries do not handle the I/O exception themselves. Instead, they wrap it or translate it and then throw a new exception at the application code.[8]

8. Log4J is a pleasant exception in this regard. It uses a pluggable ErrorHandler policy to dispose of exceptions in any of the "appenders."

What happens next is anyone's guess. In the best-case scenario, the logging filesystem is separate from any critical data storage (such as transactions), and the application code protects itself well enough that users never realize anything is amiss. Significantly less pleasant, but still tolerable, is a nicely worded error message asking the users to have patience with us and please come back when we've got our act together. Several rungs down the ladder is serving a stack trace to the user.

Worse yet, I saw one system where the developers had added a "universal exception handler" to the servlet pipeline. This handler would log any kind of exception. It was reentrant, so if an exception occurred while logging an exception, it would log *both* the original and the new exception. As soon as the filesystem got full, this poor exception handler went nuts, trying to log an ever-increasing stack of exceptions. Because there were multiple threads, each trying to log its own Sisyphean exception, this application server was able to consume eight entire UltraSPARC III CPUs—for a little while, anyway. The exceptions, multiplying like Leonardo da Pisa's rabbits, rapidly consumed all available memory. This was followed shortly by a JVM crash.

Don't leave log files on production systems. Copy them to a staging area for analysis.

A less dramatic problem with large log files is their poor signal-to-noise ratio. Consider access logs from a web server. Other than WebTrends-type analysis, it's very unlikely that you will find value in last month's access logs. With eight million requests, which corresponds to 800,000 to 4,000,000 page views, depending on the number of assets per page, Apache's common log format produces more than a 1GB a day in access logs. No human being can find an event of interest in that volume of data. And by the way, there's no reason to leave those log files on production systems. Copy them off to a staging area for analysis.

Of course, it's always better to avoid filling up the filesystem in the first place. Log file rotation requires just a few minutes of configuration.

The various translations of Log4J, including Log4R (Ruby) and Log4Net (any .NET language), all support a RollingFileAppender, which can be configured to rotate log files based on size. You should always use RollingFileAppender in place of the default FileAppender. In java.util.logging, the default FileHandler can also be configured to rotate logs based on size by setting its limit property to the maximum number of bytes to write to the current file. The count variable controls how many old files to keep. The

Joe Asks...

What About Sarbannes-Oxley? Don't We Have to Keep All Our Log Files Forever?

You will sometimes hear people talking about logging in terms of Sarbannes-Oxley (SOX) requirements. SOX makes many heavy demands on IT infrastructure and operations. One of these demands is that the company must be able to demonstrate adequate controls on any system that produces financially significant information. In other words, if a billing system feeds into the company's financial reports, the company must be able to demonstrate that nobody can monkey with the billing system's data.

For most customer-facing websites, this is irrelevant in reality but often perceived as necessary. Financials come from order management systems or credit card settlement systems, not from web and application servers. The website cannot possibly retain web server logs for the years required by SOX, not even on tape or DVD. Further, could web server access logs actually prove anything about the integrity of the financial controls? Not likely. That comes from tracking administrator login sessions.

Unfortunately, legal issues are not always decided based on rational probability analysis, particularly in an area as fuzzy and ill-defined as SOX compliance. Your best bet is to work with your company's CIO or compliance staff. (Many companies have dedicated SOX consultants.) They will help define how your system can stay in compliance. Start these discussions early. They involve legal, IT, and finance departments, so you should not expect speedy resolution.

product of limit and count obviously determines how much space the log files can possibly consume.

In the case of legacy code, third-party code, or code that doesn't use one of the excellent logging frameworks available, the logrotate utility is ubiquitous on UNIX. For Windows, you can try building logrotate under Cygwin, or you can hand roll a .vbs or .bat script to do the job. Logging can be a wonderful aid to transparency. Make sure that all log files will get rotated out and eventually purged, though, or you will eventually spend time fixing the tool that's supposed to help you fix the system.

Between data in the database and log files on the disk, there are plenty of ways for persistent data to clog up your system. Like jingles from old commercials, sludge stuck in memory can clog up your application.

In-Memory Caching

Pattern 10.2, *Use Caching Carefully*, on page 210 has much more to say on the subject of caching. To a long-running server, memory is like oxygen. Cache, left untended, will suck up all the oxygen. Low memory conditions are a threat to both stability and capacity. Therefore, when building any sort of cache, it's vital to ask two questions:

- Is the space of possible keys finite or infinite?
- Do the cached items ever change?

If there is no upper bound on the number of possible keys, then cache size limits must be enforced. Unless the key space is finite and the items are static, then the cache needs some form of cache invalidation. The simplest mechanism is a time-based cache flush. You can also investigate least recently used (LRU) or working-set algorithms, but nine times out of ten, a periodic flush will do.

Improper use of caching is the major cause of memory leaks, which in turn lead to horrors like daily server restarts. Nothing gets administrators in the habit of being logged on to production like daily (or nightly) chores.

Sludge buildup is a major cause of slow responses, so steady state helps avoid that antipattern. Steady state also encourages better operational discipline by limiting system administrators' need to log on to the production servers.

Remember This

Avoid fiddling

Human intervention leads to problems. Eliminate the need for recurring human intervention. Your system should run at least for a typical deployment cycle without manual disk cleanups or nightly restarts.

Purge data with application logic

DBAs can create scripts to purge data, but they don't always know how the application behaves when data is removed. Maintaining logical integrity, especially if you use an ORM tool, requires the application to purge its own data.

Limit caching

In-memory caching speeds up applications, until it slows them down. Limit the amount of memory a cache can consume.

Roll the logs

Don't keep an unlimited amount of log files. Configure log file rotation based on size. If you need to retain them for compliance, do it on a nonproduction server.

5.5 Fail Fast

If slow responses are worse than no response, the worst must surely be a slow *failure* response. It's like waiting through the interminable line at the DMV, only to be told you need to fill out a different form and go back to the end of the line. Can there be any bigger waste of system resources than burning cycles and clock time only to throw away the result?

If the system can determine in advance that it will fail at an operation, it's always better to fail fast. That way, the caller doesn't have to tie up any of its capacity waiting; it can get on with other work.

How can the system tell whether it will fail? What kind of secret heuristics am I about to reveal? Is this the application-level equivalent of Intel's branch-prediction algorithms?

It's actually much more mundane than that. There is a large class of "resource unavailable" failures. For example, when a load balancer gets a connection request but not one of the servers in its service pool is functioning, it should immediately refuse the connection. Some configurations have the load balancer queue the connection request for a while, in the hopes that a server will become available in a short period of time. This violates the Fail Fast pattern.

Check resource availability at the start of a transaction.

In any service-oriented architecture, the application can tell from the service requested roughly what database connections and external integration points will be needed. The service can very quickly check out the connections it will need and verify the state of the circuit breakers around the integration points. It can tell the transaction manager to start a transaction. This is sort of the software equivalent of the cook's *mise en place*—gathering all the ingredients it will need to service the request before it begins. If any of the resources are not available, it can fail immediately, rather than getting partway through the work.

Black

One of my more interesting projects was for a studio photography company. Part of the project involved working on the software that rendered images for high-resolution printing. The previous generation of this software exhibited a problem that generated more work for humans downstream: if any color profiles, images, backgrounds, or alpha masks were not available, it "rendered" a black image—full of zero-valued pixels.

This black image went into the printing pipeline and was printed, wasting paper, chemicals, and time. Quality checkers would pull the black image and send it back to the people at the beginning of the process for diagnosis, debugging, and correction. Ultimately, they would fix the problem (usually by calling developers to the printing facility) and remake the bad print. Since the order was already late getting out the door, they would expedite the remake—meaning that it interrupted the pipeline of work and went to the head of the line.

When my team started on the rendering software, we applied the Fail Fast pattern. As soon as the print job arrived, the renderer would check for the presence of every font (missing fonts caused a similar remake, but not because of black images), image, background, and alpha mask. It preallocated memory, so it couldn't fail an allocation later. The renderer reported any such failure to the job control system immediately, before it wasted several minutes of compute time. Best of all, "broken" orders would be pulled from the pipeline, avoiding the case of having partial orders waiting at the end of the process. Once we launched the new renderer, software-induced remake rate[9] dropped to zero.

The only thing we didn't preallocate was disk space for the final image. We violated "steady state" under the direction of the customer, who indicated that they had their own rock-solid purging process. Turns out the "purging process" was one guy who occasionally deleted a bunch of files. A little less than one year after we launched, the drives filled up. Sure enough, the one place we broke the Fail Fast principle was the one place our renderer failed to report errors before wasting effort. It would render images—several minutes of compute time—and then throw an IOException in the log file.

Another way to fail fast in a web application is to perform basic parameter-checking in the servlet or controller that receives the request, before loading EJBs or domain objects. Be cautious, however, that you do not violate encapsulation of the domain objects. If you are checking for more than null/not-null or number formatting, you should move those validity checks into the domain objects or an application facade.

Even when failing fast, be sure to report a system failure (resources not available) differently than an application failure (parameter violations or invalid state). Reporting a generic "error" message may cause an upstream system to trip a circuit breaker just because some user entered bad data and hit Reload three or four times.

9. Orders could still be remade because of other quality problems: dust in the camera, poor exposure, bad cropping, and so on.

The Fail Fast pattern improves overall system stability by avoiding slow responses. Together with timeouts, failing fast can help avert impending cascading failures. It also helps maintain capacity when the system is under stress because of partial failures.

Remember This

Avoid Slow Responses and Fail Fast

If your system cannot meet its SLA, inform callers quickly. Don't make them wait for an error message, and don't make them wait until they time out. That just makes your problem into their problem.

Reserve resources, verify Integration Points early

In the theme of "don't do useless work," make sure you will be able to complete the transaction before you start. If critical resources aren't available—for example, a popped Circuit Breaker on a required call out—then don't waste work by getting to that point. The odds of it changing between the beginning and the middle of the transaction are slim.

Use for input validation

Do basic user input validation even before you reserve resources. Don't bother checking out a database connection, fetching domain objects, populating them, and calling `validate()` just to find out that a required parameter wasn't entered.

5.6 Handshaking

Handshaking refers to signaling between devices that regulate communication between them. Serial protocols such as RS-232 (now EIA-232C) rely on the receiver to indicate when it is ready to receive data. Analog modems used a form of handshaking to negotiate a speed and a signal encoding that both devices would agree upon. And, as illustrated earlier, TCP uses a three-phase handshake to establish a socket connection. TCP handshaking also allows the receiver to signal the sender to stop sending data until the receiver is ready. Handshaking is ubiquitous in low-level communications protocols but is almost nonexistent at the application level.

The sad truth is that HTTP doesn't handshake well. HTTP-based protocols, such as XML-RPC or WS-I Basic, have few options available for handshaking. HTTP provides a response code of "503 Service Unavailable," which is defined to indicate a temporary condition.[10] Most clients, however, will not distinguish between different response codes. If the code is not a "200 OK,"[11] "403 Authentication Required," or "302 Found (redirect)," the client probably treats the response as a fatal error.

Similarly, the protocols underneath CORBA, DCOM, and Java RMI are equally bad at signaling their readiness to do business.

Handshaking is all about letting the server protect itself by throttling its own workload. Instead of being victim to whatever demands are made upon it, the server should have a way to reject incoming work. The closest approximation I've been able to achieve with HTTP-based servers relies on partnership between a load balancer and the web or application servers. The web server notifies the load balancer—which is pinging a "health check" page on the web server periodically—that it is busy by returning either an error page (HTTP response code 503 "Not Available" works) or an HTML page with an error message. The load balancer then knows not to send any additional work to that particular web server. Of course, this helps only for web services and still breaks down if all the web servers are too busy to serve another page.

In a service-oriented architecture, the server can provide a "health check" query for use by clients. The client would then check the health of the server before making a request. This provides good handshaking at the expense of doubling the number of connections and requests

10. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.
11. Many clients even treat other 200 series codes as errors!

the server must process. On the downside, most of the time for a typical web service call is spent just setting up and tearing down the TCP connection, so making a health check call before the actual call just doubles that connection overhead.

Handshaking can be most valuable when unbalanced capacities are leading to slow responses. If the server can detect that it will not be able to meet its SLAs, then it should have some means to ask the caller to back off. If the servers are sitting behind a load balancer, then they have the binary on/off control of stopping responses to the load balancer, which would in turn take the unresponsive server out of the pool. This is a crude mechanism, though. Your best bet is to build handshaking into any custom protocols that you implement.

Circuit breakers are a stopgap you can use when calling services that cannot handshake. In that case, instead of asking politely whether the server can handle the request, you just make the call and track whether it works.

Overall, handshaking is an underused technique that could be applied to great advantage in application-layer protocols. It is an effective way to stop cracks from jumping layers, as in the case of a cascading failure.

Remember This

Create cooperative demand control

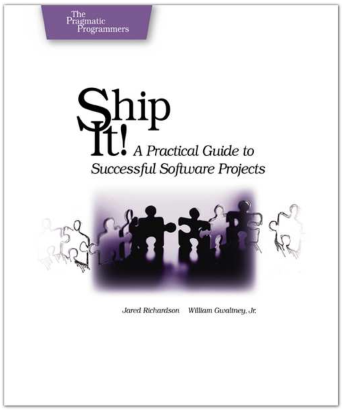
Handshaking between client and server permits demand throttling to serviceable levels. Both client and server must be built to perform Handshaking. Most common application-level protocols—such as HTTP, JRMP, IIOP, and DCOM—do not perform Handshaking.

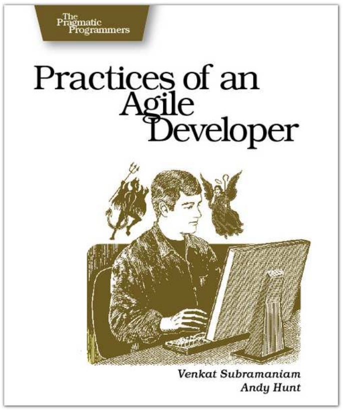
Consider health checks

Health-check requests are an application-level workaround for the lack of Handshaking in the protocols. Consider using them when the cost of the added call is much less than the cost of calling and failing.

Build Handshaking into your own low-level protocols

If you create your own socket-based protocol, build Handshaking into it, so the endpoints can each inform the other when they are not ready to accept work.


The Pragmatic Programmers
Ship It! A Practical Guide to Successful Software Projects
Jared Richardson William Gwaltney, Jr.


The Pragmatic Programmers
Practices of an Agile Developer
Venkat Subramaniam
Andy Hunt