Extracted from:

Release It! Second Edition

Design and Deploy Production-Ready Software

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Release It!

Second Edition

Design and Deploy
Production-Ready Software

Michael T. Nygard
Edited by Katharine Dvorak

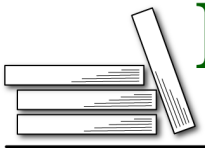
Release It! Second Edition

Design and Deploy Production-Ready Software

Michael T. Nygard

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-239-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

CHAPTER 4

Stability Antipatterns

Delegates to the first NATO Software Engineering Conference coined the term *software crisis* in 1968. They meant that demand for new software outstripped the capacity of all existing programmers worldwide. If that truly was the start of the software crisis, then it has never ended! (Interestingly, that conference also appears to be the origin of the term *software engineering*. Some reports say it was named that way so certain attendees would be able to get their travel expenses approved. I guess that problem hasn't changed much either.) Our machines have gotten better by orders of magnitude. So have the languages and libraries. The enormous leverage of open source multiplies our abilities. And of course, something like a million times more programmers are in the world now than there were in 1968. So overall, our ability to create software has had its own kind of Moore's law exponential curve at work. So why are we still in a software crisis? Because we've steadily taken on bigger and bigger challenges.

In those hazy days of the client/server system, we used to think of a hundred active users as a large system; now we think about millions. (And that's up from the first edition of this book, when ten thousand active users was a lot.) We've just seen our first billion-user site. In 2016, Facebook announced that it has 1.13 *billion* daily active users.[1] An "application" now consists of dozens or hundreds of services, each running continuously while being redeployed continuously. Five nines of reliability for the overall application is nowhere near enough. It would result in thousands of disappointed users every day. Six Sigma quality on Facebook would create 768,000 angry users per day. (200 requests per page, 1.13 billion daily active users, 3.4 defects per million opportunities.)

The breadth of our applications' reach has exploded, too. Everything within the enterprise is interconnected, and then again as we integrate across

1. http://venturebeat.com/2016/07/27/facebook-passes-1-billion-mobile-daily-active-users

enterprises. Even the boundaries of our applications have become fuzzy as more features are delegated to SaaS services.

Of course, this also means bigger challenges. As we integrate the world, tightly coupled systems are the rule rather than the exception. Big systems serve more users by commanding more resources; but in many failure modes big systems fail faster than small systems. The size and the complexity of these systems push us to what author James R. Chiles calls in *Inviting Disaster [Chi01]* the "technology frontier," where the twin specters of high interactive complexity and tight coupling conspire to turn rapidly moving cracks into full-blown failures.

High interactive complexity arises when systems have enough moving parts and hidden, internal dependencies that most operators' mental models are either incomplete or just plain wrong. In a system exhibiting high interactive complexity, the operator's instinctive actions will have results ranging from ineffective to actively harmful. With the best of intentions, the operator can take an action based on his or her own mental model of how the system functions that triggers a completely unexpected linkage. Such linkages contribute to "problem inflation," turning a minor fault into a major failure. For example, hidden linkages in cooling monitoring and control systems are partly to blame for the Three Mile Island reactor incident, as Chiles outlines in his book. These hidden linkages often appear obvious during the postmortem analysis, but are in fact devilishly difficult to anticipate.

Tight coupling allows cracks in one part of the system to propagate themselves—or multiply themselves—across layer or system boundaries. A failure in one component causes load to be redistributed to its peers and introduces delays and stress to its callers. This increased stress makes it extremely likely that another component in the system will fail. That in turn makes the next failure more likely, eventually resulting in total collapse. In your systems, tight coupling can appear within application code, in calls between systems, or any place a resource has multiple consumers.

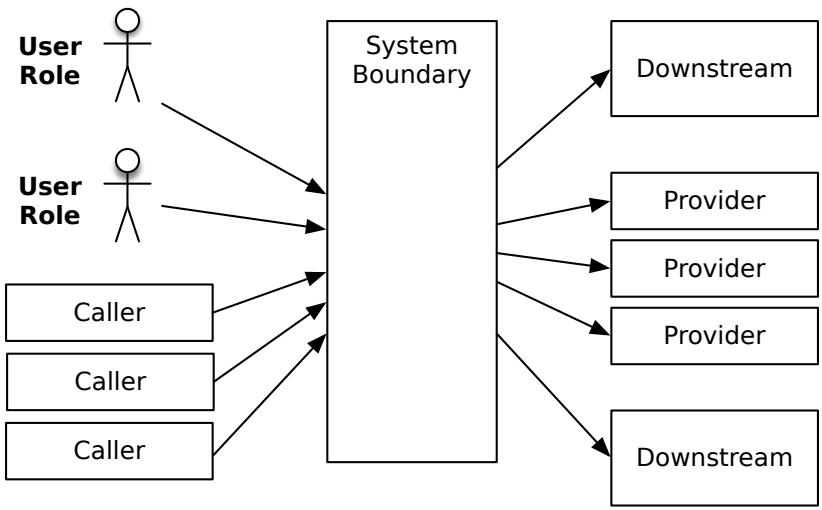
In the next chapter, we'll look at some patterns that can alleviate or prevent the antipatterns from harming your system. Before we can get to that good news, though, we need to understand what we're up against.

In this chapter, we'll look at antipatterns that can wreck your system. These are common forces that have contributed to more than one system failure. Each of these antipatterns will create, accelerate, or multiply cracks in the system. These bad behaviors are to be avoided.

Simply avoiding these antipatterns isn't sufficient, though. Everything breaks. Faults are unavoidable. Don't pretend you can eliminate every possible source of them, because either nature or nurture will create bigger disasters to wreck your systems. Assume the worst. Faults will happen. We need to examine what happens *after* the fault creeps in.

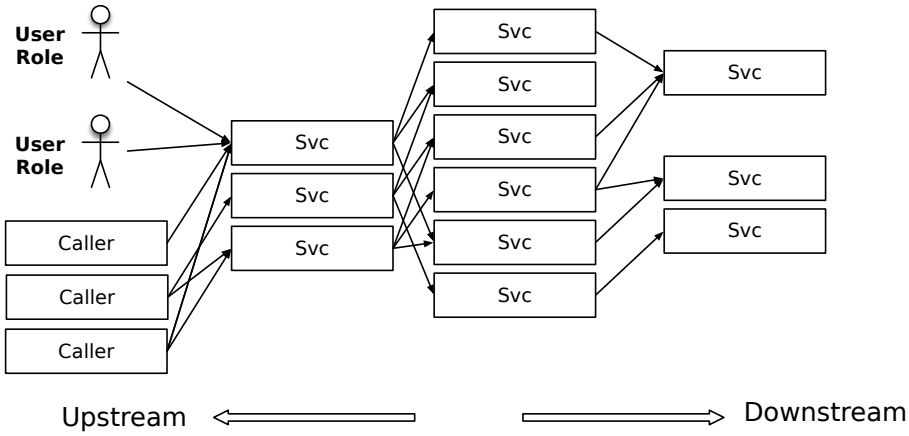
Integration Points

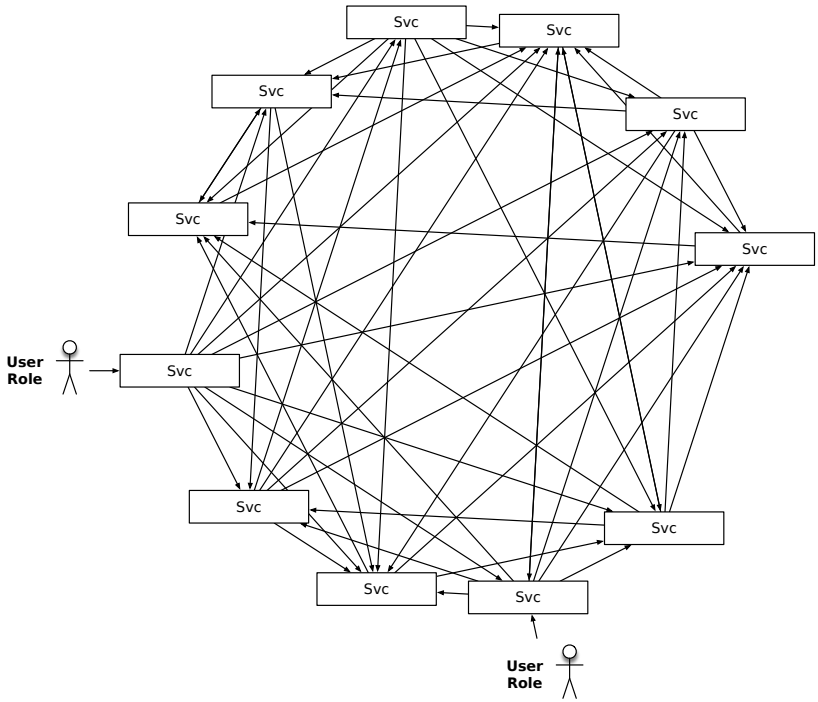
I haven't seen a straight-up "website" project since about 1996. Everything is an integration project with some combination of HTML veneer, front-end app, API, mobile app, or all of the above. The context diagram for these projects will fall into one of two patterns: the butterfly or the spider. A butterfly has a central system with a lot of feeds and connections fanning into it on one side and a large fan out on the other side, as shown in the figure that follows.



Some people would call this a monolith, but that has negative connotations. It might be a nicely factored system that just has a lot of responsibility.

The other style is the spiderweb, with many boxes and dependencies. If you've been diligent (and maybe a bit lucky), the boxes fall into ranks with calls through tiers, as shown in the first figure on page 8. If not, then the web will be chaotic like that of the black widow, shown in the second figure on page 8. The feature common to all of these is that the connections outnumber the services. A butterfly style has $2N$ connections, a spiderweb might have up to 2^N, and yours falls somewhere in between.





All these connections are integration points, and every single one of them is out to destroy your system. In fact, the more we move toward a large number of smaller services, the more we integrate with SaaS providers, and the more we go API first, the worse this is going to get.

You Have How Many Feeds?

I was helping launch a replatform/rearchitecture project for a huge retailer. It came time to identify all the production firewall rules so we could open holes in the firewall to allow authorized connections to the production system. We had already gone through the usual suspects: the web servers' connections to the application server, the application server to the database server, the cluster manager to the cluster nodes, and so on.

When it came time to add rules for the feeds in and out of the production environment, we were pointed toward the project manager for enterprise integration. That's right, the site rebuild project had its own project manager dedicated just to integration. That was our second clue that this was not going to be a simple task. (The first clue was that nobody else could tell us what all the feeds were.) The project manager understood exactly what we needed. He pulled up his database of integrations and ran a custom report to give us the connection specifics.

Feeds came in from inventory, pricing, content management, CRM, ERP, MRP, SAP, WAP, BAP, BPO, R2D2, and C3P0. Data extracts flew off toward CRM, fulfillment, booking, authorization, fraud checking, address normalization, scheduling, shipping, and so on.

On the one hand, I was impressed that the project manager had a fully populated database to keep track of the various feeds (synchronous/asynchronous, batch or trickle feed, source system, frequency, volume, cross-reference numbers, business stakeholder, and so on). On the other hand, I was dismayed that he *needed* a database to keep track of it!

It probably comes as no surprise, then, that the site was plagued with stability problems when it launched. It was like having a newborn baby in the house; I was awakened every night at 3 a.m. for the latest crash or crisis. We kept documenting the spots where the app crashed and feeding them back to the maintenance team for correction. I never kept a tally, but I'm sure that every single synchronous integration point caused at least one outage.

Integration points are the number-one killer of systems. Every single one of those feeds presents a stability risk. Every socket, process, pipe, or remote procedure call can and will hang. Even database calls can hang, in ways obvious and subtle. Every feed into the system can hang it, crash it, or generate other impulses at the worst possible time. We'll look at some of the specific ways these integration points can go bad and what you can do about them.

Socket-Based Protocols

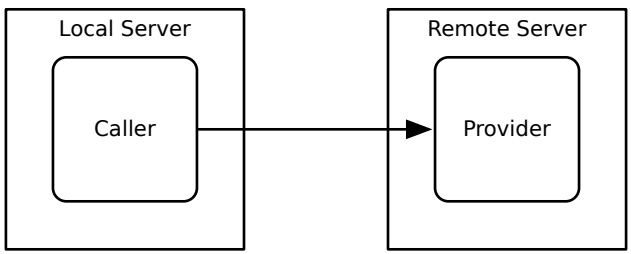
Many higher-level integration protocols run over sockets. In fact, pretty much everything except named pipes and shared-memory IPC is socket-based. The

higher protocols introduce their own failure modes, but they're all susceptible to failures at the socket layer.

The simplest failure mode occurs when the remote system refuses connections. The calling system must deal with connection failures. Usually, this isn't much of a problem, since everything from C to Java to Elm has clear ways to indicate a connection failure—either an exception in languages that have them or a magic return value in ones that don't. Because the API makes it clear that connections don't always work, programmers deal with that case.

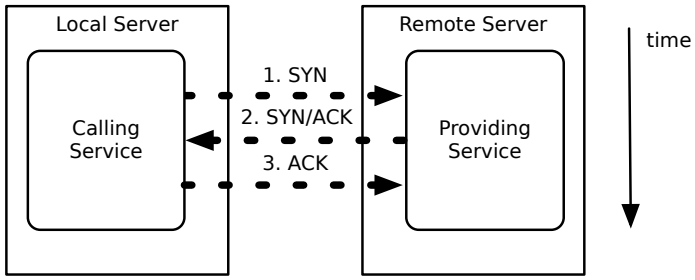
One wrinkle to watch out for, though, is that it can take a *long* time to discover that you can't connect. Hang on for a quick dip into the details of TCP/IP networking.

Every architecture diagram ever drawn has boxes and arrows, similar to the ones in the following figure. (A new architect will focus on the boxes; an experienced one is more interested in the arrows.)



Like a lot of other things we work with, this arrow is an abstraction for a network connection. Really, though, that means it's an abstraction for an abstraction. A network "connection" is a logical construct—an abstraction—in its own right. All you will ever see on the network itself are packets. (Of course, a "packet" is an abstraction, too. On the wire, it's just electrons or photons. Between electrons and a TCP connection are many layers of abstraction. Fortunately, we get to choose whichever level of abstraction is useful at any given point in time.) These packets are the Internet Protocol (IP) part of TCP/IP. Transmission Control Protocol (TCP) is an agreement about how to make something that looks like a continuous connection out of discrete packets. The figure on page 11 shows the "three-way handshake" that TCP defines to open a connection.

The connection starts when the caller (the client in this scenario, even though it is itself a server for other applications) sends a SYN packet to a port on the remote server. If nobody is listening to that port, the remote server immediately sends back a TCP "reset" packet to indicate that nobody's home. The calling application then gets an exception or a bad return value. All this



happens very quickly, in less than ten milliseconds if both machines are plugged into the same switch.

If an application is listening to the destination port, then the remote server sends back a SYN/ACK packet indicating its willingness to accept the connection. The caller gets the SYN/ACK and sends back its own ACK. These three packets have now established the "connection," and the applications can send data back and forth. (For what it's worth, TCP also defines the "simultaneous open" handshake, in which both machines send SYN packets to each other before a SYN/ACK. This is relatively rare in systems that are based on client/server interactions.)

Suppose, though, that the remote application is listening to the port but is absolutely hammered with connection requests, until it can no longer service the incoming connections. The port itself has a "listen queue" that defines how many pending connections (SYN sent, but no SYN/ACK replied) are allowed by the network stack. Once that listen queue is full, further connection attempts are refused quickly. The listen queue is the worst place to be. While the socket is in that partially formed state, whichever thread called `open()` is blocked inside the OS kernel until the remote application finally gets around to accepting the connection or until the connection attempt times out. Connection timeouts vary from one operating system to another, but they're usually measured in *minutes*! The calling application's thread could be blocked waiting for the remote server to respond for ten minutes!

Nearly the same thing happens when the caller can connect and send its request but the server takes a long time to read the request and send a response. The `read()` call will just block until the server gets around to responding. Often, the default is to block forever. You have to set the socket timeout if you want to break out of the blocking call. In that case, be prepared for an exception when the timeout occurs.

Network failures can hit you in two ways: fast or slow. Fast network failures cause immediate exceptions in the calling code. "Connection refused" is a very

fast failure; it takes a few milliseconds to come back to the caller. Slow failures, such as a dropped ACK, let threads block for minutes before throwing exceptions. The blocked thread can't process other transactions, so overall capacity is reduced. If all threads end up getting blocked, then for all practical purposes, the server is down. Clearly, a slow response is a lot worse than no response.