TSS Editors Note:
Manning Publications and TheServerSide.com have partnered to bring you this chapter of Java Doctor, a new book, in which you can contribute to and get published! Manning is providing you free early access to these tips and in the spirit of cooperation we hope you will also contribute tips back to the authors. To proceed directly to the tips, scroll down to page 5 (section 9.3), or read on to learn how you can contribute and also have your name published.

Reality: action and pitfall tips

- In this Chapter

- How it happened
- Reality
- Submission guidelines
- Action tips
- Pitfall tips

It happened one Thursday night. The night, like most weekday evenings, was filled with keyboard clicking sounds as we cranked away at the chapters of this book. Collaborating via instant messenger, we were engrossed in edits and re-edits, adding and moving content, and checking for grammar and thought flow. We knew we had a lot of work to do to get the book out on schedule. Everything took a backseat, even our personal lives. But this was no ordinary Thursday night: "The Apprentice" would be on TV at 9PM, and that's when all work ceased and we became glued to the glowing boxes in our living rooms.

If you're not familiar with "The Apprentice," it's a reality TV show on which contestants in business attire compete in teams over several weeks to get a job with Donald Trump. The day-to-day trials and tribulations, politics, and drama between the individuals and teams come to a head each week when one contestant is fired. Reality TV had a grip on us and a lot of other people. The camera was there to capture things as they were, without a lot of scripting. Hmm.

That's when it struck us.

9.1 Reality

Why not have a "reality" chapter? After all, we're dealing with troubleshooting, something we're sure many people have war stories about. Like many other books, much of the material in this book is based on our factual knowledge. But where are the real-life, real-world elements? Where's the drama? The trials? The tribulations? Why not have a chapter that lets people contribute their own troubleshooting tips and techniques to share with the world? It sounded like a great idea, and we were excited.

We took our idea to the people at Manning and asked them to consider doing something with the folks at TheServerSide.com. If we could tap into the TSS community and solicit troubleshooting tips from the community, we believed we'd get an enormous response. As members of the community, we also wanted to give something back. So we decided that when we receive and accept a submission, we'll not only use it in this chapter but also give full credit to the submitter. That's right—submitters' names (and contact information) will appear in print. Try that on your resume!

Tip

If you're interested in sending us a troubleshooting tip for possible publication, email it to javadoctor@manning.com. We highly suggest that you read the tip submission guidelines discussed in the rest of this chapter.

The idea evidently went over well with Manning and TheServerSide.com. However, they asked us to take care of some formalities. The most important is making sure submissions are on target with what we're looking for. This chapter provides a clear explanation of our expectations, so contributions will have a high chance of being accepted. We want to include a large number of tips, and having guidelines in place will increase the hit ratio. So before you rush off an email with a story from the trenches, please look at the submission guidelines that follow.

Note

We'll do our best to include as many tips as possible. However, we may not be able to include certain tips that are similar to the ones already submitted or that don't conform to the guidelines, or if we've run out paper to print them on.

9.2 Submission categories and guidelines

As it turned out, we couldn't finish watching that episode of "The Apprentice" because we got so wrapped up with the idea of including real-world troubleshooting tips. While Donald Trump was busy firing someone in the boardroom, we were coming up with ways to categorize tips and some simple guidelines.

9.2.1 Tip categories

We knew a hodgepodge of tips would be messy, so we came up with ways to categorize the content we received:

- *Action tips*—These tips provide information about tool usage or techniques and how a tool can be ingeniously used to identify or solve a production issue. Such a tip could be about a little-known switch to a command-line tool or the JVM or a script you've written and found very useful. It could be a special technique for using a common tool such as `vmstat` (see chapter 2 for information about `vmstat`). A good criterion for an action tip is that it involves a degree of activity from the reader to be able benefit from it. That is, the reader must perform some actions to use the tip.

- *Pitfall tips*—In chapter 6, we looked at common coding pitfalls. In that chapter, we examined code that we've seen in many cases at many organizations. The code had a negative impact on scalability, performance, or manageability. We're looking for similar advice in the pitfall tips category, but in a broader sense than just code. We're interested in hearing about the kinds of architectural, programmatic, or design flaws you've seen in applications. Such a tip could describe how incorrectly configured firewalls can affect a cluster. Or, it could tell how an EJB is commonly written in a not-so-ideal fashion.

A good differentiator between a pitfall tip and an action tip is that pitfall tips usually provide information that can be absorbed by the reader, who then benefits from it the knowledge. No action is required—unless the reader realizes that the pitfall is something they have in their own production system! On the other hand, action tips arm the reader with techniques or tools to diagnose certain problems that may arise in production systems.

Within these two categories, there are additional classifications:

- *Infrastructure*—Primarily focused on hardware, such as memory and the network
- *Middleware*—Applications, identity, LDAP, portals, and other middle-tier products
- *JVM*—Common pitfalls seen in any version of any vendor's JVM
- *Database*—Items that cover Java and interactions with databases, such as transactions

It's possible that any of the tips you submit may fit in more than one classification. After all, technology boundaries aren't clear cut. We urge you to pick the classification you feel best covers your tip. If you feel it can't be appropriately classified, we're open to your suggestions. We may reclassify all the tips as we get a wider grasp on the total content submitted over time.

9.2.1 Submission guidelines

In addition to identifying the category and classification you feel your tip belongs to, be sure your tips adhere to the five basic guidelines set forth here. As we mentioned earlier, these guidelines are in place to increase the likelihood that your tip will be published. The goal of the guidelines is to keep the tips aligned to the overall intent of this book.

Production-related

We all know that when stuff hits the fan, it's in production. Therefore, the first guideline is that your tip should be applicable in a production environment. This means certain characteristics can be attributed to your tip. For example, its intrusion on a running application should be minimal. Using a Java debugger technique, although useful, isn't applicable in a production environment, because the load of the debugger itself can destabilize the production environment.

Related to systemic problems

We'd like to hear how you tracked down issues that affected scalability, availability, performance, and manageability. How were you able to determine a bottleneck, undo it, and increase performance? How did you detect a connection pool issue? How were you able to monitor an application in ways that were extremely insightful? What technique did you use to identify scalability hurdles?

Unique and creative

The most interesting tips are always the ones that are unique and creative. This book has covered many tips and techniques; however, we're looking for ingenious ways of doing interesting troubleshooting.

Widely useful

We prefer your tips to be applicable to a wide body of administrators, developers, and architects in a wide range of technologies (within the Java/J2EE universe), as opposed to being useful only in a unique, proprietary system. This may mean using some of the tools that come out of the box with many operating systems, JVMs, and best-of-breed vendor tools.

The more people can immediately use your tip, the more likely it will be published. This doesn't mean your tip should be technically simplistic; it just means that someone reading the tip should be able to say "Hmm, I think I'll try that now!" and be able to do so.

Consise and in the correct context

When you're writing, make sure you present the proper contextual information. You've troubleshot an environment, and as a result, you're familiar with it. Writing a tip may involve making certain assumptions about what the reader knows—assumptions you take for granted due to the intimacy of the environment you worked with. Providing a proper context allows the reader to make fewer assumptions, understand the motivations and forces behind your troubleshooting effort, and thus be more appreciative.

We'd also appreciate your keeping tips to fewer than 250 words, but we'll leave room for exceptions.

9.2.3 Sending your submissions

If you're interested in sending tips, please do so at javadoctor@manning.com. Be sure you include your name and, optionally, the email address you'd like printed next to your tip. The deadline for tip submission is *January 30th, 2005,* so hurry!

We'll kick off the process by providing several tips of our own in the following sections. If you plan to send a submission, you may want to read our tips to get a feel what we're looking for. As always, if you have any suggestions about the material, let us know.

9.3 Action tips

A dog may be a man's best friend, but a good troubleshooting technique is ours. Following are some of the best tips from TheServerSide.com. Try them out!

> Tip
>
> If you're interested in sending us a troubleshooting tip for possible publication, email it to javadoctor@manning.com. We highly suggest that you read the tip submission guidelines discussed earlier in this chapter.

9.3.1 Action tip #1: Java and the CPUs

By the authors, The Java Doctor

We were assigned to an architecture assessment project at a large bank for an application that had been having tremendous performance problems for several months. Running through the usual checks described in this book, it became evident that the application was CPU starved. On further investigation, we determined that 4 CPUs were allocated to the application in a large Sun E10K box consisting of 32 CPUs. We also found that certain CPUs were completely idle while the remaining CPUs were assigned to other applications. When we asked why these CPUs were idle and whether they could be allocated to the ailing application, the administrative team responded that the CPUs were made available only when the need arose or a specific request was made and that the idle CPU would not be released. This seemed to be a political stance more than anything else, and in our opinion the need to maintain orderly distribution of resources shouldn't hinder the critical need for the resources. Suffice to say, our architecture assessment report suggested a review of this policy and the allocation of one of the idle CPUs as the quickest remedy for the application.

In another, very similar, situation, the client was fuming about how slowly a Java application ran on a four-CPU Sun system. Starting on the usual path, which was to get a feel for the problem and other externalities involved, we raised an important question: how many other applications ran on the server. The answer was that the server was entirely dedicated to the application. However, looking at the CPU utilizations, it was clear that one CPU always sat idle. Taking a `psrset`, we confirmed that the application wasn't using all four CPUs, but just three. After the client escalated the matter within the organization, it turned out that the infrastructure group wanted to preserve the extra CPU for future work. (We still haven't figured out the rationale behind this.)

You need to be aware that in organizations, multiple groups are involved in the workings of an application. Not all groups are on the same page, making it the troubleshooter's responsibility to check everything that would otherwise seem like a safe assumption. In this case, everyone on the development team assumed that the application was using all the CPUs. In the end, our discovery didn't completely resolve the problem, but it helped improve application throughput.

9.3.2 Action tip #2: Sniffing network communication

By Kevin Tung, Software Architect, kev_tung@earthlink.net

The first step in troubleshooting any type of application defect is to isolate the problem. Distributed applications generally complicate the process of problem isolation. One of the best tools available to developers is the Ethereal network protocol analyzer, an open-source network sniffer.

Ethereal is lightweight and can be used to capture network traffic to and from your development and production machines. By seeing the actual TCP/UDP payload on the wire, you can easily determine the scope of the problem; that allows you to zero in on the particular node within which the problem resides.

Due to the prominence of web services / SOAP as a means of remote invocation, I'm often faced with SOAP-related problems that are hard to trace back to a specific origin. Vendor-generated stubs and proxy classes add another layer of indirection that makes troubleshooting web services difficult. In these circumstances, I often use Ethereal to validate the actual SOAP messages being sent and received. Sometimes I uncover subtle nuances in vendor implementation, and at other times I uncover errors of my own doing. Here's a case in point:

- *Context*—Building a SOAP web service client for a large pharmaceutical firm based in

Denmark.

- *The requirement*—Two disparate web sites need to share a common authentication mechanism. One of the sites is running on IIS/ASP.Net, and the other is running on Apache/Resin.
- *The approach*—Generated stub classes wouldn't work well because the WSDL could change and recompilation of the client side code is impossible. Instead, the SOAP calls must be made programmatically with parameters and types defined in deployment descriptors.
- *The problem*—Without the convenience of WSDL binding, we had to figure out all the right options to set for the proper interoperation between Axis SOAP client and the Microsoft web service. Our first attempt using Axis's default configuration didn't work and returned little useful information.
- *The solution*—Using Ethereal's packet-capture capability, we were able to trap and compare the Axis-generated SOAP messages against Microsoft's expected message. It turned out that `SOAPActionUri` needs to be set properly. Once we changed that parameter, the two web servers happily interchanged authentication info. Ethereal allowed us to quickly identify the problem and easily resolve it instead of reading pages and pages of documentation or relying on a Good Samaritan to respond to our newsgroup postings.

To get Ethereal, go to www.ethereal.com/.

9.3.3 Action tip #3: Isolating JDBC misuse

By Derek C. Ashmore, dashmore@dvt.com, author of The J2EE Handbook

- *Tip*—Use a JDBC profiler to detect JDBC resource leaks in applications prior to production.
- *Background*—After a lengthy testing period in testing environments, my client applied a service pack to their EJB container in production. Shortly after that, one of their most high-volume applications was erring out on all requests because it had allocated all available database connections allowed by its database connection pools. On a gradual basis, other applications affected by the same upgrade began experiencing the same symptoms.
- *Observations*—We suspected that connections were being leaked (opened, but not closed), but we were puzzled as to why the same code worked properly prior to applying the service pack.
- *Actions*—In the short term, we assigned an administrator to recycle all containers periodically to avoid outages due to the connection leaks. This bought us time to analyze the issue on test equipment.

Visual inspection of the code didn't reveal the source of the connection leaks. Many applications hadn't centralized connection management, so I needed a general solution for detecting connection leaks.

Fortunately, I had used an open-source JDBC profiler called P6Spy (www.p6spy.com). It works as a proxy for other JDBC drivers, so it can log SQL execution times for performance-tuning purposes. Because it was open source, I altered it slightly to report connection leaks. Since then, I've further enhanced it to report leaks for other types of JDBC objects, such as `Statement`s, `PreparedStatement`s, `CallableStatement`s, and `ResultSet`s.

Since then, before deploying builds to production, I run a test sequence using P6Spy to detect JDBC leaks.

9.3.4 Action tip #4: JMX to the rescue

By John Musser, jrmusser@hotmail.com

The command line is your friend.

In these days of ubiquitous GUI admin tools like those in front of everything from Tomcat to WebLogic to LDAP servers, it can be easy to forget just how useful good ol' command-line tools can be. As any decent system administrator can tell you, a GUI can be handy at times, but trying to perform the same operation repeatedly or performing the same action across a cluster of servers can be slow, error-prone, and downright tedious through a GUI. There's an old admin adage that says "If you do it more than once, script it." Even to a Java developer, this can be sage advice.

We encountered a situation in which a customer was running multiple applications on a WebLogic server cluster; they wanted to know how many user sessions were active at any given time. Time was short, and we didn't have the luxury of developing a custom application to dig into the MBeans themselves; nor was there time (or authority) to instrument the application code itself. Using the WebLogic console also wasn't an option because using that UI required repeatedly clicking across applications and clusters in order to get sum totals. In this case, it was WebLogic's `weblogic.Admin` command-line interface to the rescue.

This handy tool, often used for WebLogic administration and configuration, served our monitoring needs nicely. Because all WebLogic services are exposed as JMX MBeans (just like those from most of the major application servers), we were able to get the count of active sessions right from the command line. In our script, we summed the totals across all the applications and servers and presented the customer with a clean tabular output showing exactly how many users were currently in each application and in the system as a whole (yes, some sessions may have been stale and awaiting to expire, but with a 30-minute timeout, these were limited and could be factored into the calculation).

As an example of how the `weblogic.Admin` tool works, here's a basic administrative command to shut down a server:

```
java weblogic.Admin -url <hostname:port> -username <user> -password
<mypass> SHUTDOWN
```

In our case, we needed a command that looked like the following query (note that this is very implementation- and deployment-specific):

```
java weblogic.Admin -url <hostname:port> -username <user> -password
<mypass> 1 -GET -pretty -mbean
mydomain:ApplicationRuntime=myserver_explodeweb,Location=myserver,Name=
myserver_myserver_explodeweb_exploded_mywebapp,ServerRuntime=myserver,T
ype=WebAppComponentRuntime -property SessionsOpenedTotalCount
```

We put this in a script that repeated it for each application and cluster we were interested in, and then summed the result.

Note that in order for this technique to work, you first need to set Enable Session Monitoring on the WebLogic server. You can do this in the console or, of course, via the command line. You can verify this setting via the following command:

```
java weblogic.Admin -url <hostname:port> -username <user> -password
<mypass>
GET -type WebAppComponent -property SessionMonitoringEnabled
```

Remember, everything that can be done in the console can be done via command line. You can also do some things via command line that *can't* be done in the console, such as interact not just with the Administration Server but also with Managed Servers—something quite handy when the admin server has crashed.

9.4 Pitfall tips

We all know the saying that a wise person is one who learns from his mistakes, but a wiser person is one who learns from the mistakes of others. In that spirit, this section will highlight some common problems that occur in enterprise systems and discuss the symptoms they display. It covers problems that we've encountered numerous times in production environments. We expect to see them again in the future without fail, even after the virulent, stampeding success of this book! Well, anyway, the idea is to help you home in on a similar problem quickly and comfortably once you've acquainted yourself with what's described here. Better yet, these tips will make you aware of issues that will lead you to take more preventive measures, and an ounce of prevention is always better than a pound of cure.

Tip

If you're interested in sending us a pitfall tip for possible publication, email it to javadoctor@manning.com. We highly suggest that you read the tip submission guidelines discussed earlier in this chapter.

9.4.1 Pitfall tip #1: Check the firewall

By the authors, The Java Doctor

We'll describe a problem we faced while trying to figure out why a certain application kept failing, especially during the wee hours of the morning. The client had a web-based application that catered to a large user base. The application was required to have high availability, and because of that the customer required that any unplanned outages should be minimal. For each outage, a report needed to be filed and tracked to ensure the problem wasn't repeated in the future.

Unfortunately, the application had at least one unplanned outage per day. At the point we were called in, the application development team as well as the operations team had run out of ideas and theories as to the source of the problem. The striking part was the time at which the application suffered the outage: the early hours of the morning, just as the first few customers signed in. The users weren't happy that the application repeatedly failed on them, day after day.

After a close look at the application logs, it became clear that the application failure occurred around the time that database connections were refreshed. This in itself was an enigma: Why did

the application attempt to refresh the database connections? And this possibly had something to do with the application server not being able to create a new connection pool. We first started looking into the source of this problem, hoping to catch a lucky break by solving why the connections couldn't be reestablished. However, the source of this problem turned out to be more obvious and also less relevant to the real issue.

As the last few users signed off at the end of the previous business day, the database connections in the pool sat idle. After a few hours, some idle connections started getting timed-out by the database server. Therefore, the application database connection pool began dropping connections but held on to the stale connection objects.

As the first users logged in during the early hours of the morning, the database connection threw a `SQLException`. The exception was thrown because an attempt was made to execute a SQL query using a stale connection. When the application connection pool received the exception, it attempted to refresh the whole pool.

We were finally able to explain the anomaly of why the database connections were being refreshed. But we still had to deal with the main problem: why the application was unable to create a new connection pool. The inability of the application to create a new connection pool meant that the end users could no longer use the system, because access to the database was severed. We tried a number of tools and techniques, including an attempt to reproduce the problem in the test environment by waiting overnight for the database connections to become stale. The test environment was able to easily re-create the connections to the database, much to our dismay. Why was the problem only occurring in production?

Then it was time to request the deployment diagram for the production environment. We wanted to see how the database server and the application server communicated. The presence of a firewall between the application server and the database in production immediately caught everyone's attention. The test environment didn't have that, and so we excitedly started viewing the configurations of the firewall. The firewall turned out to be the culprit, because it was configured to keep connections alive for a longer duration of time.

This might not have been much of a problem if the database wasn't running near capacity with a limit on maximum number of open connections, but here's how the two interplayed. The application would establish connections to the firewall, which would then open separate connections to the database. When connections to the database became stale, the application attempted to drop its connection pool and re-create a new pool of connections. However, because the firewall was configured to retain connections for a long span of time, it retained the older connections going to the database server. The refresh couldn't happen because the number of connections being attempted was greater than the database's maximum limit. As a result, the refresh of the connection pool failed.

The solution was simple: Change the firewall setting to reduce the time for which the connections were held alive.

9.4.2 Pitfall tip #2: Pool connections

By the authors, The Java Doctor

A client had an outage problem with their web-based application and couldn't determine why they occurred during peak hours. They had decided not to use database connection pools and instead were creating a new connection for each request to the database. As a result, during peak load periods, a large number of connections were instantiated and opened to the database. Their rationale for giving up on database connection pooling was that an average programmer writes

code with lots of bugs, and they couldn't force a programmer to use the connection pool properly. If their programmers didn't properly code or didn't return the connection back to the pool, then the total available connections would be exhausted. This does happen in many development shops, so their rationale may have been justified—but their workaround wasn't.

Digging in deeper, we found that the application stopped responding due to very high lock contention at the `java.sql.DriverManager` level. The client couldn't understand why a simple lock contention caused the application to stall. After all, they argued, slight performance degradation was to be expected during high loads.

It's important to understand that when lock contentions exist, as they did with the synchronizations in `DriverManager`, performance degradation won't be linear, and a spike in load can cause the system to stall. The fear of bad programming led the client to bad application design. The solution was to use connection pooling, which is a necessity especially when a large number of connections need to be made to the database. Improper programming shouldn't be acceptable when you're writing enterprise-level applications; instead, a rigorous process should be in place to enforce and encourage good coding techniques.

With the connection pool in place, the application outages were reduced by 70%. The remaining outages were due to buggy JNI code. This code was later identified and eventually resolved. Ah yes, another satisfied customer.

9.4.3 Pitfall tip #3: Data latency in clusters

By the authors, The Java Doctor

We were called in by a client whose Java application stopped displaying data correctly when the directory servers failed-over. The application required data that was stored and updated in a directory server. To improve application availability, redundancy had been built it. The directory server architecture incorporated redundancy as well, by having one primary server and another backup server.

The two directory server IP addresses were known by the application, and when the primary failed, the application established connections with the backup. For the data to be consistent across the two directory servers, the configuration enforced the two to synch up any changes made.

However, the Java application just wouldn't recover from a failover by the directory servers. The application showed incorrect data once the failover had occurred. The values the application had reported only moments before the failover were now different. The complexity of the application required that we reduce the problem down to simpler terms and solve it. We therefore wrote a basic test application that changed data in the directory server and then requested the changed entries.

After we wrote the application, we attempted to re-create the test case in which their application had been failing. After running the application, we intentionally failed-over the master directory server. The directory server architecture replaced the failed directory server with a new master directory server, as correctly claimed by the infrastructure group. However, what the infrastructure group got wrong was the fact that the failover wasn't perfect, because the test application retrieved stale data from the new master directory server. The stale data had only moments earlier been updated before we flicked off the switch of the primary directory server. The test application showed that the directory servers were the ones not returning correct data to the application. And all along, the assumption had been that the Java application was at fault.

The client was finally convinced of the nature of the problem, which was due to synchronization delays between the directory servers. The synchronization configuration of the directory servers was three seconds. The solution was simple in this case: The configuration for synchronization was reduced, because three seconds was too long. Within this time frame, crucial data written to one directory server didn't synchronize with the backup in case of a failover. We reduced the synchronization time to a 100 milliseconds. Problem solved, case closed.

9.4.4 Pitfall tip #4: IO buffering

By the authors, The Java Doctor

We received an emergency call requesting assistance and fast! A client that was known for good software-engineering processes and excellent architects had run into a problem. They had spent six months developing a large enterprise application for B2B transactions, but during its first load test it had failed to achieve the expected SLA. The application couldn't scale over a few hundred users, and the response times were unacceptable.

We flew in and were immediately taken to a conference room, which was referred to as the "war room." The application's architecture was drawn on a white board, showing the class, deployment, and sequence diagrams. The team knew what software engineering was about. They were fairly confident about their skills—so much so that they had openly begun wondering whether Java was up to the enterprise level.

The client took us through all the tests and analysis they had done, from running profilers to writing micro benchmarks, but they couldn't isolate the source of the problem. They even had detailed performance numbers taken by each subsystem of their application through the use of performance logs. They had definitely given performance some thought in the design process, by ensuring that their application logged the time taken by each transaction and, for each transaction, the time taken in each component. After reviewing what had already been done, we decided to take the driver's seat.

It was time to get down and dirty. We started looking at the operating system during the load runs. It was initially puzzling to see that the CPUs were on the idle side when they shouldn't be in an application with significant business logic. We used `sar -d 2 1`, which shows the average queue size of jobs waiting to be written to disk as well as the average service time of each request before it's written to disk (details in chapter 2). Although the application wasn't logging large amounts of data, including the performance log, the disk was showing excessively large average disk queue lengths—over 30 in our case. The CPUs were idling, and the disk queues were large. Very strange, we thought. It must be related to logging, but we had seen log files in other applications being written to at a much faster pace without an issue.

So we moved to the next step of pulling up code where the logging was occurring, and to our surprise, the output file streams weren't using any buffering (such as the `BufferedOuputStream` wrapper). We modified the code, the disk queues disappeared, and the application worked like a breeze.

9.4.5 Pitfall tip #5: Throttle when required

By the authors, The Java Doctor

One client we worked with admitted that their hardware wasn't the best. In fact, the servers were slow. The client was aware of this problem but couldn't replace them in the very near future. The problem they wanted solved was that their application was unable to respond to users

during peak work times. They wondered if there was a simple way to throttle the application server so as to accept all incoming user requests but still be able to respond to the users.

To enable all incoming requests to be accepted by the application, we first ensured that the web server's incoming request queues were large enough to hold user requests. However, our strategy revolved around configuring the worker threads in the application server. Setting the number of worker threads in a J2EE application server is a tricky issue. It's a common practice to have a large number of worker threads in application servers, to avoid making clients wait. However, this becomes a problem, because a very large number of worker threads is detrimental to the application's overall throughput. If the CPUs are already maxed out, having more active threads than a certain threshold (specific to the server) reduces the throughput of the server and, at a certain stage, can cripple it.

All servers have a certain threshold of work that they can do, after which any additional work starts queuing up. With more threads, the server has to do the additional work of context-switching between threads. For a server already overloaded with work, context-switching takes priority. This can lead to a server doing no useful work, because its gets bogged down in context-switching between threads.

In our case, we wanted the number of worker threads in the application server to be sufficient. This would mean having a balanced number of worker threads to enable optimal performance for the given server. We determined the numbers of ideal worker threads through load testing.

When J2EE application servers run out of worker threads, the web server holds onto the requests in its queue until they time out. However, if the application server can service the requests in a sufficient time, then the worst that the end users notice is a long pause. But they do get their response, and the operations folks don't have to bounce the application servers to get the servers out of a state of doing very little except context-switching threads.

The place that needs attention when going with an approach of throttling an application based on its worker threads is the callback logic. A *callback* is when the application uses certain logic or when the configuration is such that a request made by a worker thread requires that the request be sent out of the container and back in. A callback in an application server with a drained thread pool results in a deadlock.

One source of a callback in an application is when an object opens a `URLConnection` to a resource within the same container. Another is deployment configuration related where J2EE application servers enforce a callback when invoking an EJB, as a different thread executes the EJB then the one calling it. If you're using the throttling approach and run into a deadlock, get a thread dump to determine where the callback is being made from (chapter 4 has details on figuring deadlocks based on thread dumps).

In our case, our solution worked to the extent that the application became slow during peak hours, but it continued to service the clients. This was an acceptable solution to the client, and in the absence of better hardware, we felt it was the best alternative.

9.4.6 Pitfall tip #6: Use JVM as prescribed

By the authors, The Java Doctor

A potential client was experiencing constant down times with their application, which consisted of an applet on the front end and servlets in the back end. They complained that they had to constantly kill the JVM process that housed the servlet container, thus dropping their applet

clients and losing any state information. This occurred with more frequency as the load reached a mere 100 concurrent users.

A conference call was set up for one of us, the client's tech lead, and several of their developers. After the initial onslaught of blaming Java for being slow and unreliable, the true discovery process started. The client's perspective was that since the JVM was the cause of the problem, the creators of the JVM needed to address the issue.

The conference call wasn't meant to solve their problems on the spot but was meant to advise them on possible remedies. The first goal was to identify the architecture so we could build a base on which to make recommendations. Figure 9.1 shows a rough view of their architecture, based on the information collected during the call

[Picture of Applet→Servlet→JVM→Scripts→Database]

Figure 9.1 Client architecture

As they described the architecture, it started to become clear that they were a very heavy UNIX shop and looked at the JVM as a kernel for the servlet container. For them, since a UNIX process could be killed externally, a thread in the virtual machine should be able to be killed just as easily (for example, the `kill` command in Solaris). Their analogy was that the OS was to a process what the JVM was to a thread. They constantly said, "A thread is the same as a process." At some level this may be true, but the differences are huge. Wiping out a process releases the entire address space allocated to that process. Killing a thread externally can have unpredictable or uncontrollable consequences. The client disagreed and insisted the two were identical. With these types of concepts held fervently, we knew there were darker and murkier areas in the architecture yet to appear. Yet we hadn't gotten to the JVM misuse!

As the architecture became clearer (settling into that shown in figure 9.1), the question that begged to be asked was, why did they want to kill threads? Assuming it could be done safely externally (from outside the JVM), what was the reasoning? The answer was that they were doing `Runtime.exec()`s from within their servlets and calling dozens of Solaris shell scripts. The scripts were themselves opening connections to the database, querying it, writing the results to a file, and then closing the connection. Occasionally, they said, the scripts would hang. So they wanted to kill the thread that was hung by the script. Since they couldn't kill the thread, they killed the whole JVM process. They wanted to know why the JVM couldn't have a parameter passed to it to indicate that a thread reaching a certain timeout period should automatically be terminated. A bad design was being blamed on the JVM, and we couldn't let that happen!

Here we had an example of misuse of the JVM. Not only were they not using database connection pools, but they were bound to have heavy I/O latency issues due to large report queries being put into UFS files. Their solution to the I/O issue was to set up a SAN, and now, according to them, their only problem was the misbehaving JVM.

Since time was limited and the call was a freebie for the client to get them going in the right direction, we focused on the JVM issues. We put the database, I/O, and other yet-to-be discovered issues on the back burner. We explained that making calls to scripts from the JVM wasn't a good practice. It was a well-known way to destabilize the VM, because the VM had no control over what happened once the script was executed. Scripts couldn't report back exceptions to the JVM, and the state of the script couldn't then be determined. The client finally understood and began looking for possible solutions to fix the problem.