IS THE GLASS HALF FULL OR HALF EMPTY? PG 4

JAN/FEB 2006 VOL:5 ISSUE:1

wldj™

WWW.WLDJ.COM

THE LEADING INDEPENDENT MAGAZINE FOR WEBLOGIC™ PROFESSIONALS

INTEGRATING APACHE POI IN WEBLOGIC SERVER ...16

PLUS...

With BEA WebLogic Platform™, global organizations are building powerful business advantages using SOA. And the ultimate value comes when these vital applications are in full production.

But keeping these complex applications up and running can be a challenge – putting you, and your business, at risk.

Intersperse Manager™ removes those risks, optimizing your investment in BEA and delivering on the promise of SOA. Using non-invasive management standards, such as JMX, Intersperse Manager enables real time visibility, context, and control at all levels.

It gives you true, proactive production application management that corrects issues – before they become problems.

Intersperse Manager helps you build business advantage – and that's an asset worth protecting.

Call Intersperse today for
a free demonstration

866.499.2202

intersperse™
Management solutions
for the evolving enterprise
www.intersperse.com

bea 8.1 VALIDATED
BEA WEBLOGIC PLATFORM

Is the Glass Half Full or Half Empty?

By Peter Holditch

THREADS, AMDAHL, AND A VERY BIG MACHINE

In this column over the years, I have spent a considerable amount of time talking about contention and locking in the database tier. At the end of the day, the endless conversations about scaling the application tier boil down to less than a bag of beans if a scaled application can't go any faster because the database has hit a limit.

Sometimes, the "database hitting a limit" involves a limit to the physical capacity of the database to service requests for data, which usually leads to purchase of a larger database server, or some kind of partitioning or tuning work. Other times, the database server is not too busy, but the data itself is the problem (or at least, the pattern of access to the data) – remembering back to ACID transactions and the rules for using them, the most golden of golden rules is that transactions should be as short in duration as possible, to ensure that the system does not grind to a halt. The reason a system with lots of long transactions in it might grind to a halt is that an ACID transaction is associated with a bunch of locks in the database; therefore, the longer the transaction is active, the higher the probability that several pieces of the application will try to obtain the same lock – resulting in the application being bogged down in contention for the same piece of data. In this situation no amount of hardware can help; while data accesses do not touch each others' locks, they can fly through the database as fast as the server can access its disks. As soon as multiple pieces of the application want to touch the same data, things are not so free and easy.

This is all well understood stuff, at the database tier. A lot of effort has gone in over the years to remove lock contention wherever possible. In particular, optimistic locking has become a well-established technique to reduce data contention. With optimistic locking, the database will give its locks out to multiple accessors simultaneously, on the assumption that they will not make conflicting updates to the data. At the time the transaction commits, if it turns out the two accessors did actually trample on each other, one of the transactions will be rolled back with an exception, since it has turned out that the database's optimism was unfounded and the access really needed to be serialized in the more traditional pessimistic way. The success of this technique in increasing throughput is not magic; clearly, it works because what is being contended for is only the lock – not the data itself (this being the key optimistic assumption). If the actual data were being contended for a majority of the time, optimistic locking would actually result in a net slowdown, brought about by the endless collisions and retrying.

So why am I rambling at high speed through this resume of transaction design and database lock management?

"You cannot change the laws of physics!"

Well, it turns out that the same principles apply in other tiers – surprise surprise, these are "laws of physics" after all… So why haven't we heard the analogue to this tale of lock-related pain and trickery in the application tier? Well, it all comes down to what there is a lot of, and what there is not much of. In the database view, there is just one database, and a desire for throughput that leads to many concurrent accessors. It is as you try to drive the concurrency up that you get more and more likely to hit locking issues in your database. So where is that pattern in the application tier? Well, one of the features of the Java language is easy threadedness so there are likely to be many threads, as anyone who has designed a Java application will know of the "singleton" design pattern because it is often used to represent something that there is only one of in a given JVM – a log singleton, a hashmap of reference data, you name it… everywhere there are apps, there are singletons!

So, you say, apps are full of singletons; Java in general and Java application servers in particular have a capacity for running lots of threads through them, so why hasn't every highly threaded java application ground to a halt long ago? Amdahl's law (the particular "law of physics" in question),

wldj THE LEADING INDEPENDENT MAGAZINE FOR WEBLOGIC™ PROFESSIONALS

EDITORIAL

Founding Editor
Peter Zadrozny

Executive Editor
Seta Papazian seta@sys-on.com

Editor
Nancy Valentine nancy@sys-con.com

Research Editor
Bahadir Karuv, PhD bahadir@sys-con.com

PRODUCTION

Production Consultant
Jim Morgan jim@sys-con.com

Lead Designer
Abraham Addo abraham@sys-con.com

Art Director
Alex Botero alex@sys-con.com

Associate Art Directors
Louis F. Cuffari louis@sys-con.com
Richard Silverberg richards@sys-con.com
Tami Beatty tami@sys-con.com
Andrea Boden andrea@sys-con.com

Video Production
Frank Moricoo frank@sys-con.com

EDITORIAL OFFICES

SYS-CON MEDIA
135 Chestnut Ridge Rd., Montvale, NJ 07645
Telephone: 201 802-3000 Fax: 201 782-9638
WLDJ (ISSN #1535-9581)
is published bimonthly (6 times a year)
by SYS-CON Publications, Inc.

Postmaster: send address changes to:
WLDJ: SYS-CON MEDIA
135 Chestnut Ridge Rd., Montvale, NJ 07645

Worldwide Newsstand Distribution
Curtis Circulation Company, New Milford, NJ
FOR LIST RENTAL INFORMATION:
Kevin Collopy: 845 731-2684,
kevin.collopy@edithroman.com
Frank Cipolla: 845 731-3832,
frank.cipolla@epostdirect.com

7
6
5
Application: Operations Monitor
Customer Transactions per Second
Debit Account
Bill Payments
Authenticate
Account Info
Cardholder Service
Loan Processing

after all, states that you can only parallelize an application to the degree that the threads can be run independently – what gives!

The answer turns out to be in the hardware that is running the software virtual machine. In a typical environment, eight CPUs constitute a pretty large machine, and it is more common to have deployed many smaller servers, with one or two CPUs each. Therefore, in this environment, if you code a Java application that spawns a number of threads that all access a single hashmap as part of their work, you will not see much difference to the throughput if you compare it with and without the hashmap as you increase the number of threads from, say 50 to 5,000. The reason for that is that even if you spawned 5,000 threads, they are still getting scheduled over the eight CPUs (or however many you have), so the level of parallelism you are achieving is much less than you expected – only eight threads can really physically run at once. This points up a fundamental scaling issue with Java on traditionally architected machines. The "thing there is not much of" – the place where contention will limit your throughput – is not the hashmap, it's the CPU itself. This is one of the reasons why Java applications typically end up deploying over a pretty large set of machines – that way you get enough CPUs to really physically run things in parallel and have a hope of meeting your throughput goals and other SLAs. It is a shame about the running costs of the resulting complex, partitioned system.

This latter partitioning problem is one of those that Azul is addressing by manufacturing 384 processor SMP boxes to allow truly big Java virtual machines to run – removing the complexity of having to artificially break the application up into small pieces. "AHA!" you cry – now that lock thing in the singleton will kill you! You now have many truly concurrent threads all contending for those singletons. It turns out that there is a solution to this problem supported in the hardware (here's an advantage of designing hardware explicitly to execute virtual machines), and it's called Optimistic Thread Concurrency or OTC for short.

As the name suggests, the system's behavior using OTC is the same as (or at least analogous to) the behavior of a database doing optimistic locking: the virtual machine allows many threads to pass through a lock on the assumption that bad things won't happen (that's the optimism part). If bad things do happen and data (rather than just the lock) is actually contended for, then the contending thread is reverted to the state it was in before it acquired the lock and it proceeds forward again – none the wiser, with the integrity of the data intact, and all with no change to the application code itself.

The trick is knowing *which* locks to be optimistic about – if a particular lock protects a piece of data that is nearly always written to, all that rolling back will hurt, not help, performance. In other cases, for data that is read very frequently but seldom written such as reference data, optimism will be the best policy. In order to tune the OTC system for these different eventualities (which will doubtless coexist in a single app) there are three types of lock that are implemented internally by Azul's virtual machine. Thick locks are the usual pessimistic locks through which access will be serialized, and thin locks are the cheapest form of lock. If a thin lock experiences contention, it must be promoted to either a thick lock or a speculative lock, which is the type of lock that sits between thin and thick and allows multiple threads through, using the hardware to check that there is no contention, and, importantly, to drive the rollbacks if contention occurs. Throughout the life of a virtual machine, data is kept about all the locks. Locks that are frequently contended will be thick, those that are never contended will be thin, and those that are "mostly" uncontended will be speculative. As usage patterns change, heuristics in the VM move individual locks between these three states, thus providing optimum parallelization, and hence throughput, for the application at run time.

Not only does this scheme just improve parallelism with no code changes, but it also allows simpler code to be used. Often, programmers in search of the ultimate throughput end up coding complex nested locking schemes, which are difficult to code, and even more difficult to debug and maintain. With OTC, the locking can be coded in a coarse-grained manner (and therefore can be simple to code and maintain) without impacting throughput – for once the developers get to have their cake and eat it too.

So, fill your glass with OTC. You may find that that old glass of yours holds more than you think, if you deploy it on Azul! For more details about OTC, you can download the whitepaper from the Azul Web site, at www.azulsystems.com/products/whitepaper_abstracts.html.

Author Bio:

Peter Holditch is a senior presales engineer in the UK for Azul Systems. Prior to joining Azul he spent nine years at BEA systems, going from being one of their first Professional Services consultants in Europe and finishing up as a principal presales engineer. He has an R&D background (originally having worked on BEA's Tuxedo product) and his technical interests are in high throughput transaction systems. "Of the pitch" Peter likes to brew beer, build furniture, and undertake other ludicrously ambitious projects – but (generally) not all at the same time!

Contact:

pholditch@azulsystems.com

PRESIDENT & CEO

Fuat Kircaali fuat@sys-con.com

VP, BUSINESS DEVELOPMENT

Grisha Davida grisha@sys-con.com

GROUP PUBLISHER

Jeremy Geelan jeremy@sys-con.com

ADVERTISING

Senior VP, Sales & Marketing
Carmen Gonzalez carmen@sys-con.com

VP, Sales & Marketing
Miles Silverman miles@sys-con.com

Advertising Director
Robyn Forma robyn@sys-con.com

Advertising Manager
Megan Mussa megan@sys-con.com

Advertising Sales Manager
Dennis Leavey dennis@sys-con.com

Associate Sales Manager
Kerry Mealia kerry@sys-con.com

SYS-CON EVENTS

President, Events
Grisha Davida grisha@sys-con.com

National Sales Manager
Jim Hanchrow jimh@sys-con.com

CUSTOMER RELATIONS

Circulation Service Coordinators
Edna Earle Russell edna@sys-con.com

Manager, JDJ Store
Brunilda Staropoli bruni@sys-con.com

SYS-CON.COM

VP, Information Systems
Robert Diamond robert@sys-con.com

Web Designers
Stephen Kilmurray stephen@sys-con.com

Online Editor
Roger Strukhoff roger@sys-con.com

ACCOUNTING

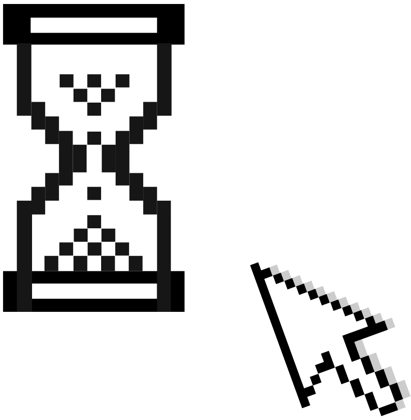
Financial Analyst
Joan LaRose joan@sys-con.com

Accounts Payable
Betty White betty@sys-con.com

Credits & Accounts Receivable
Gail Naples gailn@sys-con.com

SUBSCRIPTIONS

subscribe@sys-con.com
Call 1-888-303-5282
For subscriptions and requests for bulk orders,
please send your letters to Subscription Department
Cover Price: $8.99/issue
Domestic: $149/yr (12 issues)
Canada/Mexico: $169/yr All other countries $179/yr
(U.S. Banks or Money Orders)
Back issues: $12 U.S. (plus S&H)



Approaches to Performance Testing

A BEST-PRACTICES APPROACH TO MAXIMIZE YOUR PERFORMANCE TEST EFFORT

By Matt Maccaux

Performance testing a J2EE application can be a daunting and seemingly confusing task if you don't approach it with the proper plan in place. As with any software development process, you must gather requirements, understand the business needs, and lay out a formal schedule well in advance of the actual testing.

The requirements for the performance testing should be driven by the needs of the business and should be explained with a set of use cases. These can be based on historical data (say, what the load pattern was on the server for a week) or on approximations based on anticipated usage. Once you have an understanding of what you need to test, you need to look at how you want to test your application.

Early on in the development cycle, benchmark tests should be used to determine if any performance regressions are in the application. Benchmark tests are great for gathering repeatable results in a relatively short period of time. The best way to benchmark is to change one and only one parameter between tests. For example, if you want to see if increasing the JVM memory has any impact on the performance of your application, increment the JVM memory in stages (for example, going from 1024 MB to 1224 MB, then to 1524 MB, and finally to 2024 MB) and stop at each stage to gather the results and environment data, record this information, and then move on to the next test. This way you'll have a clear trail to follow when you are analyzing the results of the tests. In the next section I'll discuss what a benchmark test looks like and the best parameters for running these tests.

Later on in the development cycle, after the bugs have been worked out of the application and it has reached a stable point, you can run more complex types of tests to determine how the system will perform under different load patterns. These types of tests are called capacity planning, soak tests, and peak-rest tests, and are designed to test "real-world"–type scenarios by testing the reliability, robustness, and scalability of the application. The descriptions I use below should be taken in the abstract sense because every application's usage pattern will be different. For example, capacity-planning tests are generally used with slow ramp-ups (defined below), but if your application sees quick bursts of traffic during a period of the day, then certainly modify your test to reflect this. Keep in mind, though, that as you change variables in the test (such as the period of ramp-up that I talk about here or the "think-time" of the users) the outcome of the test will vary. It is always a good idea to run a series of baseline tests first to establish a known, controlled environment to compare your changes with later.

Benchmarking

The key to benchmark testing is to have consistently reproducible results. Results that are reproducible allow you to do two things: reduce the number of times you have to rerun those tests, and gain confidence in the product you are testing and the numbers you produce. The performance-testing tool you use can have a great impact on your test results. Assuming two of the metrics you are benchmarking are the response time of the server and the throughput of the server, these are affected by how much load is put onto the server. The amount of load that is put onto the server can come from two different areas: the number of connections (or virtual users) that are hitting the server simultaneously, and the amount of think-time each virtual user has between requests to the server. Obviously, the more users hitting the server, the more load will be generated. Also, the shorter the think-time between requests from each user, the greater the load will be on the server. Combine those two attributes in various ways to come up with different levels of server load. Keep in mind that as you put more load on the server, the throughput will climb, to a point.

Author Bio:
Matt Maccaux is a performance engineer on WebLogic Portal at BEA.

matt.maccaux@bea.com

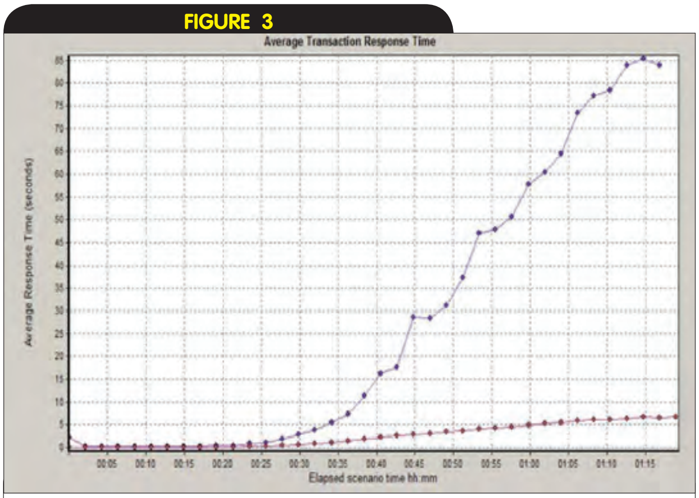
At some point, the execute queue starts growing because all of the threads on the server will be in use. The incoming requests, instead of being processed immediately, will be put into a queue and processed when threads become available.

When the system reaches the point of saturation, the throughput of the server plateaus, and you have reached the maximum for the system given those conditions. However, as server load continues to grow, the response time of the system also grows even as the throughput plateaus.

To have truly reproducible results, the system should be put under a high load with no variability. To accomplish this, the virtual users hitting the server should have 0 seconds of think-time between requests. This is because the server is immediately put under load and will start building an execute queue. If the number of requests (and virtual users) is kept consistent, the results of the benchmarking should be highly accurate and very reproducible.

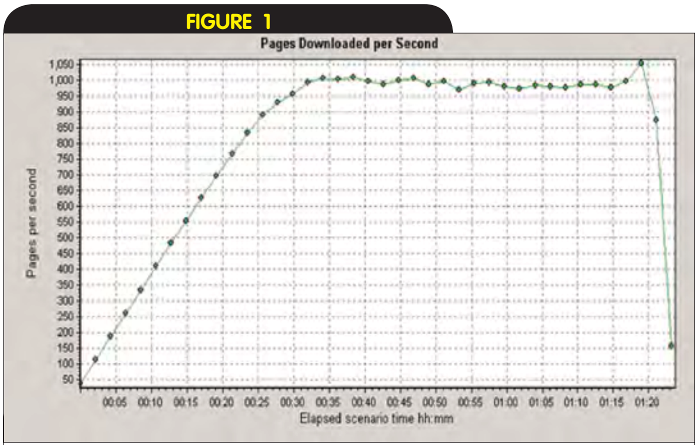
One question you should raise is, "How do you measure the results?" An average should be taken of the response time and throughput for a given test. The only way to accurately get these numbers though is to load all of the users at once, and then run them for a predetermined amount of time. This is called a "flat" run. The opposite is known as a "ramp-up" run.

FIGURE 3



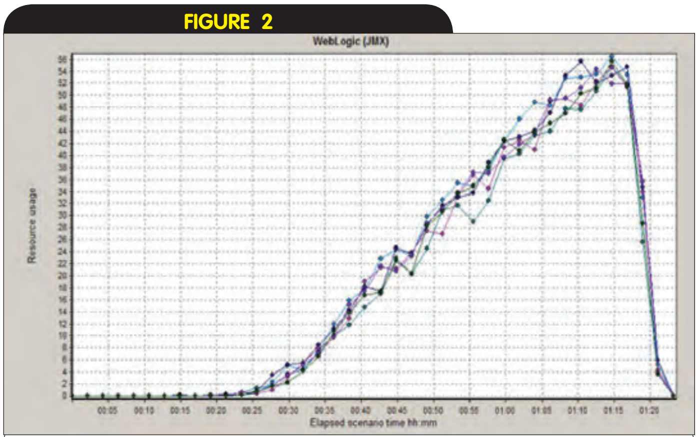
The response times of two transactions on the system as load increases over time. Note that at the same time as the execute queue (above) starts to grow, the response time also starts to grow at an increased rate. This is because the requests cannot be served immediately.

FIGURE 1



The throughput of the system in pages per second as load increases over time. Note that the throughput increases at a constant rate and then at some point levels off.

FIGURE 2



The execute queue length of the system as load increases over time. Note that the queue length is zero for a period of time, but then starts to grow at a constant rate. This is because there is a steady increase in load on the system, and although initially the system had enough free threads to cope with the additional load, eventually it became overwhelmed and had to start queuing them up.

The users in a ramp-up run are staggered (adding a few new users every x seconds). The ramp-up run does not allow for accurate and reproducible averages because the load on the system is constantly changing as the users are being added a few at a time. Therefore, the flat run is ideal for getting benchmark numbers.

This is not to discount the value in running ramp-up-style tests. In fact, ramp-up tests are valuable for finding the ballpark in which you think you later want to run flat runs. The beauty of a ramp-up test is that you can see how the measurements change as the load on the system changes. Then you can pick the range you later want to run with flat tests.

The problem with flat runs is that the system will experience "wave" effects. This is visible from all aspects of the system including the CPU utilization.

Additionally, the execute queue experiences this unstable load, and therefore you see the queue growing and shrinking as the load on the system increases and decreases over time.

Finally, the response time of the transactions on the system will also resemble this wave pattern. This occurs when all of the users are doing approximately the same thing at the same time during the test. This will produce very unreliable and inaccurate results, so something must be done to counteract this. There are two ways to gain accurate measurements from these types of results. If the test is allowed to run for a very long duration (sometimes several hours, depending on how long one user iteration takes), eventually a natural sort of randomness will set in and the throughput of the server will "flatten out." Alternatively, measurements can be taken only between two of the breaks in the waves. The drawback of this method is that the duration you are capturing data from is going to be short.

Capacity Planning

For capacity planning–type tests, your goal is to show how far a given application can scale under a specific set of circumstances. Reproducibility is not as important here as it is in benchmark testing because there will often be a randomness factor in the testing. This is introduced to try to simulate a more customer-like or real-

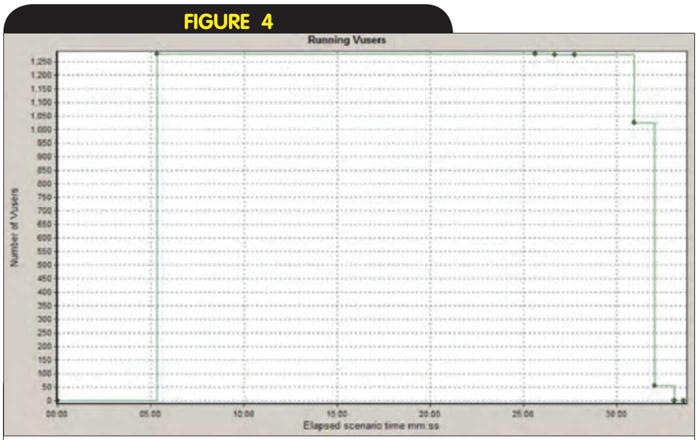
ARE YOU A WEBLOGIC EXPERT? PAGE 4
SYS-CON
NOV/DEC 2004
wldj™
THE LEADING INDEPENDENT MAGAZINE FOR WEBLOGIC™ PROFESSIONALS
WWW.WLDJ.COM
Database Controls
Best Practices
AVOIDING POTENTIAL PITFALLS DURING THE DEVELOPMENT LIFE CYCLE ...8
PLUS...
Open Source Technologies ...6
Enabling Next-Generation Portals ...14
Security Best Practices ...18
Strategies for WebLogic Domain Configuration ...20
Creating WebLogic Domains in Silent Mode ...26
A Real-World Business Process Model PART 3 ...36
web services EDGE conference
February 15-17, 2005 Hynes Convention Center Boston, MA PAGE 34
• Join Over 3,000 Developers
• App Server Shoot-Out!
• FREE Web Services
• Web Services Journal Pavilion
• Seminars and Case Studies
$1,000,000 Software Giveaway
TRANSACTIONS: And Now for Something Completely Different page 30
PRODUCT REVIEW: SOAPtest page 40
PORTAL: Enterprise Information Bus page 46
$9.99US $9.99CAN

ARE ALL JDBC DRIVERS CREATED EQUAL?
wldj
The BEA WebLogic Message Bridge

REDUCING COMPLEXITY OF REAL BUSINESS CHANGES
wldj
WebLogic on the Mac

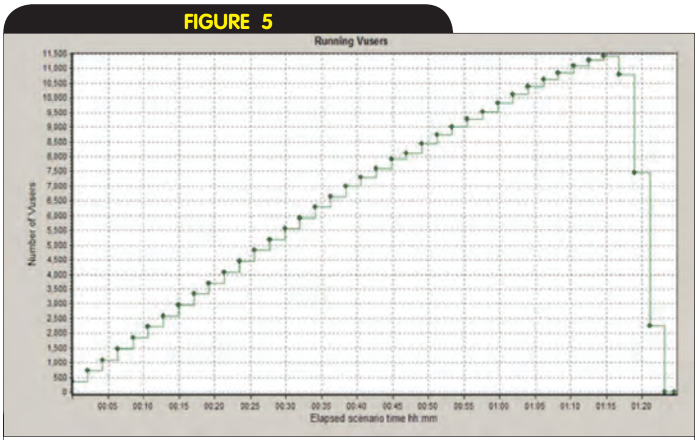
ARE YOU A WEBLOGIC EXPERT?
wldj
Database Controls
Best Practices

FIGURE 4



This is what a flat run looks like. All of the users are loaded simultaneously.

FIGURE 5



This is what a ramp-up run looks like. The users are added at a constant rate (x number per second) throughout the duration of the test.

world application with a real user load. Often the specific goal is to find out how many concurrent users the system can support below a certain server response time. As an example, the question you may ask is, "How many servers do I need to support 8,000 concurrent users with a response time of 5 seconds or less?" To answer this question, you'll need more information about the system.

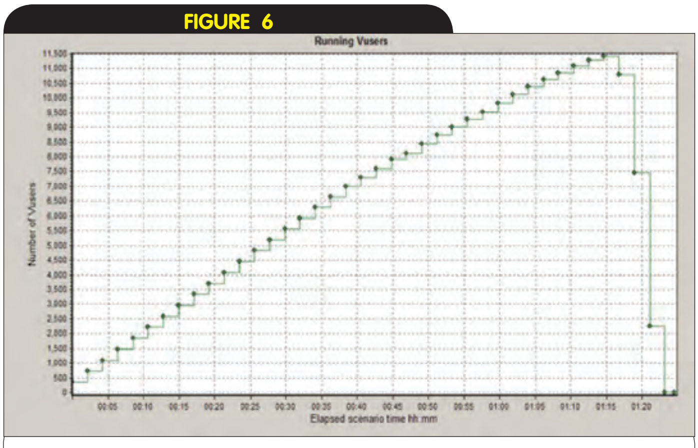
To attempt to determine the capacity of the system, several factors must be taken into consideration. Often the total number of users on the system is thrown around (in the hundreds of thousands), but in reality, this number doesn't mean a whole lot. What you really need to know is how many of those users will be hitting the server concurrently. The next thing you need to know is what the think-time or time between requests for each user will be. This is critical because the lower the think-time, the fewer concurrent users the system will be able to support. For example, a system that has users with a 1-second think-time will probably be able to support only a few hundred concurrently. However, a system with a think-time of 30 seconds will be able to support tens of thousands (given that the hardware and application are the same). In the real world, it is often difficult to determine exactly what the think-time of the users is. It is also important to note that in the real world users won't be clicking at exactly that interval every time they send a request.

This is where randomization comes into play. If you know your average user has a think-time of 5 seconds give or take 20 percent, then when you design your load test, ensure that there is 5 seconds +/- 20 percent between every click. Additionally, the notion of "pacing" can be used to introduce more randomness into your load scenario. It works like this: after a virtual user has completed one full set of requests, that user pauses for either a set period of time or a small, randomized period of time (say, 2 seconds +/- 25 percent), and then continues on with the next full set of requests. Combining these two methods of randomization into the test run should provide more of a real world–like scenario.

Now comes the part where you actually run your capacity-planning test. The next question is, "How do I load the users to simulate the load?" The best way to do this is to try to emulate how users hit the server during peak hours. Does that user load happen gradually over a period of time? If so, a ramp-up–style load should be used, where x number of users are added ever y seconds. Or, do all of the users hit the system in a very short period of time all at once? If that is the case, a flat run should be used, where all of the users are simultaneously loaded onto the server. These different styles will produce different results that are not comparable. For instance, if a ramp-up run is done and you find out that the system can support 5,000 users with a response time of 4 seconds or less, and then you follow that test with a flat run with 5,000 users, you'll probably find that the average response time of the system with 5,000 users is higher than 4 seconds. This is an inherent inaccuracy in ramp-up runs that prevents them from pinpointing the exact number of concurrent users that a system can support. For a portal application, for example, this inaccuracy is amplified as the size of the portal grows and as the size of the cluster is increased.

This is not to say that ramp-up tests should not be used. Ramp-up runs are great if the load on the system is slowly increased over a long period of time. This is because the system will be able to continually adjust over time. If a fast ramp-up is used, the system will lag and artificially report a lower response time than what would be seen if a similar number of users were being loaded during a flat run. So, what is the best way to determine capacity? Taking the best of both load types and running a series of tests will yield the best results. For example, using a ramp-up run to determine the range of

FIGURE 6



The throughput of the system in pages per second as measured during a flat run. Note the appearance of waves over time. The throughput is not smooth but rather resembles a wave pattern.

eclipse) developer's journal

A DIGITAL PUBLICATION

Your One-Stop Guide to the Eclipse Ecosystem

The main purpose of *Eclipse Developer's Journal* (*EDJ*) is to educate and inform users of Eclipse and developers building plug-ins or Rich Client Platform (RCP) applications. With the help of great columnists, news you can use, and technical articles, *EDJ* is a great information source that you will be able to use to educate yourself on just about anything Eclipse related

WHY 'ECLIPSE DEVELOPER'S JOURNAL' AND WHY NOW? PG 2

eclipse) developer's journal

PREMIER 2005 VOL 1 ISSUE 1

http://eclipse.sys-con.com

What's New in Eclipse 3.1

Polishing the apple ...4

PLUS...

WEB TOOLS PLATFORM: J2EE DEVELOPMENT THE ECLIPSE WAY ...12

'BEING A BETTER PLATFORM ...22
INTERVIEW WITH MIKE MILINKOVICH, EXECUTIVE DIRECTOR, ECLIPSE FOUNDATION

MYECLIPSE 4.0 UP ON DECK: EXCLUSIVE INTERVIEW WITH MAHER MASRI, PRESIDENT, GENUITEC ...36

ECLIPSE IDE FOR STUDENTS: USEFUL TIPS AND TRICKS ...28

Subscribe Today!

FOR YOUR DIGITAL PUBLICATION
(AVAILABLE IN DIGITAL FORMAT ONLY)

6 Issues for $19.99

Visit our site at www.eclipse.sys-con.com or call 1-888-303-5282 and subscribe today!

OFFER SUBJECT TO CHANGE WITHOUT NOTICE

http://eclipse.sys-con.com

SYS-CON.COM
SYS-CON Media, the world's leading publisher of *i*-technology magazines for developers, software architects, and e-commerce professionals, brings you the most comprehensive coverage of Eclipse

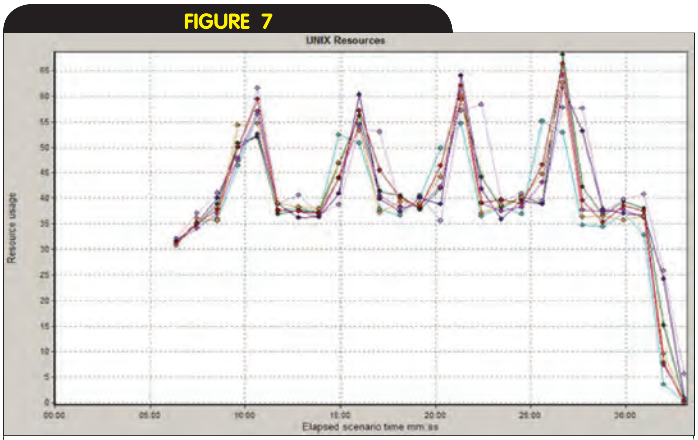
users that the system can support should be used first. Then, once that range has been determined, doing a series of flat runs at various concurrent user loads within that range can be used to more accurately determine the capacity of the system.

Soak Tests

A soak test is a straightforward type of performance test. Soak tests are long-duration tests with a static number of concurrent users that test the overall robustness of the system. These tests will show any performance degradations over time via memory leaks, increased garbage collection (GC), or other problems in the system. The longer the test, the more confidence in the system you will have. It is a good idea to run this test twice – once with a fairly moderate user load (but below capacity so that there is no execute queue), and once with a high user load (so that there is a positive execute queue).

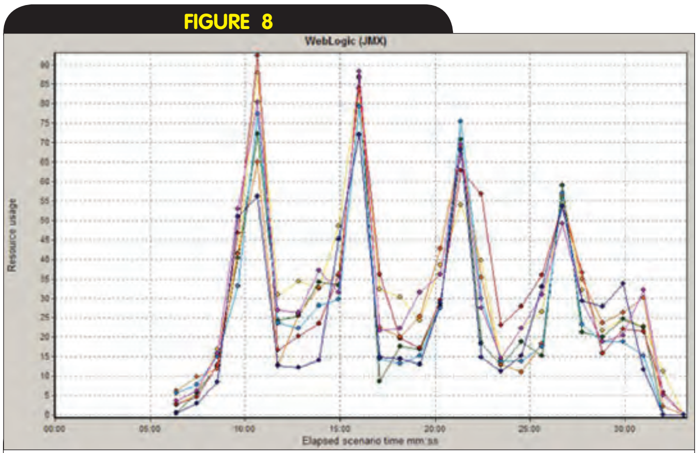
These tests should be run for several days to really get a good idea of the long-term health of the application. Make sure that the application being tested is as close to real world as possible with a realistic user scenario (how the virtual users navigate through the application), testing all of the features of the application. Ensure that all the necessary monitoring tools are running so that problems will be accurately detected and tracked down later.

FIGURE 7



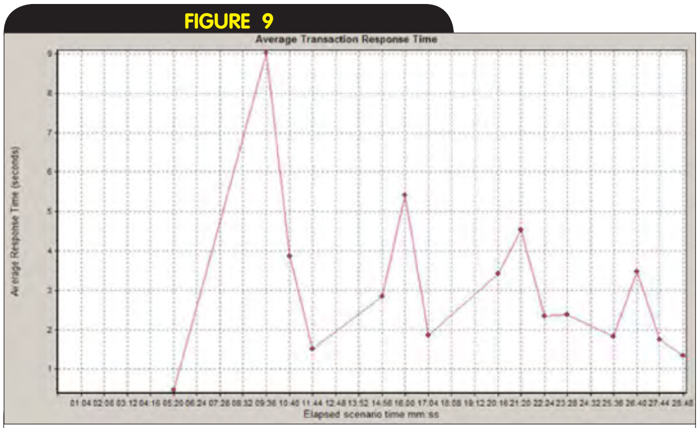
The CPU utilization of the system over time, as measured during a flat run. Note the appearance of waves over a period of time. The CPU utilization is not smooth but rather has very sharp peaks that resemble the throughput graph's waves

FIGURE 8



The execute queue of the system over time as measured during a flat run. Note the appearance of waves over time. The execute queue exactly mimics the CPU utilization graph above.

FIGURE 9



The response time of a transaction on the system over time as measured during a flat run. Note the appearance of waves over time. The transaction response time lines up with the above graphs, but the effect is diminished over time.

Peak-Rest Tests

Peak-rest tests are a hybrid of the capacity-planning ramp-up-style tests and soak tests. The goal here is to determine how well the system recovers from a high load (such as one during peak hours of the system), goes back to near idle, and then goes back up to peak load and back down again.

The best way to implement this test is to do a series of quick ramp-up tests followed by a plateau (determined by the business requirements), and then a dropping off of the load. A pause in the system should then be used, followed by another quick ramp-up – then you repeat the process. A couple of things can be determined from this: Does the system recover on the second "peak" and each subsequent peak to the same level (or greater) than the first peak? And does the system show any signs of memory or GC degradation over the course of the test? The longer this test is run (repeating the peak/idle cycle over and over), the better idea you'll have of what the long-term health of the system looks like.

Summary

This article has described several approaches to performance testing. Depending on the business requirements, development cycle, and life cycle of the application, some tests will be better suited than others for a given organization. In all cases though, you should ask some fundamental questions before going down one path or another. The answers to these questions will then determine how to best test the application. These questions are:

- How repeatable do the results need to be?
- How many times do you want to run and rerun these tests?
- What stage of the development cycle are you in?
- What are your business requirements?
- What are your user requirements?
- How long do you expect the live production system to stay up between maintenance downtimes?
- What is the expected user load during an average business day?

By answering these questions and then seeing how the answers fit into the aforementioned performance test types, you should be able to come up with a solid plan for testing the overall performance of your application.

A LIMITED TIME SAVINGS OFFER FROM *SYS-CON MEDIA*

SUBSCRIBE TODAY TO MULTIPLE MAGAZINES

AND SAVE UP TO $340 AND RECEIVE UP TO 3 FREE CDs!*

CALL TODAY! 888-303-5282

Pick a 3-Pack, a 6-Pack or a 9-Pack

□ 3-Pack	□ 1YR	□ 2YR	□ U.S.	□ Can/Mex	□ Intl.
□ 6-Pack	□ 1YR	□ 2YR	□ U.S.	□ Can/Mex	□ Intl.
□ 9-Pack	□ 1YR	□ 2YR	□ U.S.	□ Can/Mex	□ Intl.

TO ORDER •Choose the Multi-Pack you want to order by checking next to it below. •Check the number of years you want to order. •Indicate your location by checking either U.S., Canada/Mexico or International. •Then choose which magazines you want to include with your Multi-Pack order.

□ **LinuxWorld Magazine**

U.S. - Two Years (24) Cover: $143	You Pay: $79.99 /	Save: $63 + FREE $198 CD
U.S. - One Year (12) Cover: $72	You Pay: $39.99 /	Save: $32
Can/Mex - Two Years (24) $168	You Pay: $119.99 /	Save: $48 + FREE $198 CD
Can/Mex - One Year (12) $84	You Pay: $79.99 /	Save: $4
Int'l - Two Years (24) $216	You Pay: $176 /	Save: $40 + FREE $198 CD
Int'l - One Year (12) $108	You Pay: $99.99 /	Save: $8

□ **JDJ**

U.S. - Two Years (24) Cover: $144	You Pay: $99.99 /	Save: $45 + FREE $198 CD
U.S. - One Year (12) Cover: $72	You Pay: $69.99 /	Save: $12
Can/Mex - Two Years (24) $168	You Pay: $119.99 /	Save: $48 + FREE $198 CD
Can/Mex - One Year (12) $120	You Pay: $89.99 /	Save: $40
Int'l - Two Years (24) $216	You Pay: $176 /	Save: $40 + FREE $198 CD
Int'l - One Year (12) $108	You Pay: $99.99 /	Save: $8

□ **Web Services Journal**

U.S.- Two Years (24) Cover: $168	You Pay: $99.99 /	Save: $68 + FREE $198 CD
U.S. - One Year (12) Cover: $84	You Pay: $69.99 /	Save: $14
Can/Mex - Two Years (24) $192	You Pay: $129 /	Save: $63 + FREE $198 CD
Can/Mex - One Year (12) $96	You Pay: $89.99 /	Save: $6
Int'l - Two Years (24) $216	You Pay: $170 /	Save: $46 + FREE $198 CD
Int'l - One Year (12) $108	You Pay: $99.99 /	Save: $8

□ **.NET Developer's Journal**

U.S. - Two Years (24) Cover: $168	You Pay: $99.99 /	Save: $68 + FREE $198 CD
U.S. - One Year (12) Cover: $84	You Pay: $69.99 /	Save: $14
Can/Mex - Two Years (24) $192	You Pay: $129 /	Save: $63 + FREE $198 CD
Can/Mex - One Year (12) $96	You Pay: $89.99 /	Save: $6
Int'l - Two Years (24) $216	You Pay: $170 /	Save: $46 + FREE $198 CD
Int'l - One Year (12) $108	You Pay: $99.99 /	Save: $8

□ **Information Storage + Security Journal**

U.S. - Two Years (24) Cover: $143	You Pay: $49.99 /	Save: $93 + FREE $198 CD
U.S. - One Year (12) Cover: $72	You Pay: $39.99 /	Save: $39
Can/Mex - Two Years (24) $168	You Pay: $79.99 /	Save: $88 + FREE $198 CD
Can/Mex - One Year (12) $84	You Pay: $49.99 /	Save: $34
Int'l - Two Years (24) $216	You Pay: $89.99 /	Save: $126 + FREE $198 CD
Int'l - One Year (12) $108	You Pay: $59.99 /	Save: $48

□ **Wireless Business & Technology**

U.S. - Two Years (12) Cover: $120	You Pay: $49.00 /	Save: $71 + FREE $198 CD
U.S. - One Year (6) Cover: $60	You Pay: $29.99 /	Save: $30
Can/Mex - Two Years (12) $120	You Pay: $69.99 /	Save: $51 + FREE $198 CD
Can/Mex - One Year (6) $60	You Pay: $49.99 /	Save: $10
Int'l - Two Years (12) $120	You Pay: $99.99 /	Save: $20 + FREE $198 CD
Int'l - One Year (6) $72	You Pay: $69.99 /	Save: $2

□ **MX Developer's Journal**

U.S. - Two Years (24) Cover: $143	You Pay: $49.99 /	Save: $93 + FREE $198 CD
U.S. - One Year (12) Cover: $72	You Pay: $39.99 /	Save: $32
Can/Mex - Two Years (24) $168	You Pay: $79.99 /	Save: $88 + FREE $198 CD
Can/Mex - One Year (12) $84	You Pay: $49.99 /	Save: $34
Int'l - Two Years (24) $216	You Pay: $89.99 /	Save: $126 + FREE $198 CD
Int'l - One Year (12) $108	You Pay: $59.99 /	Save: $48

□ **ColdFusion Developer's Journal**

U.S. - Two Years (24) Cover: $216	You Pay: $129 /	Save: $87 + FREE $198 CD
U.S. - One Year (12) Cover: $108	You Pay: $89.99 /	Save: $18
Can/Mex - Two Years (24) $240	You Pay: $159.99 /	Save: $80 + FREE $198 CD
Can/Mex - One Year (12) $120	You Pay: $99.99 /	Save: $20
Int'l - Two Years (24) $264	You Pay: $189 /	Save: $75 + FREE $198 CD
Int'l - One Year (12) $132	You Pay: $129.99 /	Save: $2

□ **WebSphere Journal**

U.S. - Two Years (24) Cover: $216	You Pay: $129.00 /	Save: $87 + FREE $198 CD
U.S. - One Year (12) Cover: $108	You Pay: $89.99 /	Save: $18
Can/Mex - Two Years (24) $240	You Pay: $159.99 /	Save: $80 + FREE $198 CD
Can/Mex - One Year (12) $120	You Pay: $99.99 /	Save: $20
Int'l - Two Years (24) $264	You Pay: $189.00 /	Save: $75
Int'l - One Year (12) $132	You Pay: $129.99 /	Save: $2

□ **PowerBuilder Developer's Journal**

U.S. - Two Years (24) Cover: $360	You Pay: $169.99 /	Save: $190 + FREE $198 CD
U.S. - One Year (12) Cover: $180	You Pay: $149 /	Save: $31
Can/Mex - Two Years (24) $360	You Pay: $179.99 /	Save: $180 + FREE $198 CD
Can/Mex - One Year (12) $180	You Pay: $169 /	Save: $11
Int'l - Two Years (24) $360	You Pay: $189.99 /	Save: $170 + FREE $198 CD
Int'l - One Year (12) $180	You Pay: $179 /	Save: $1

□ **WLDJ**

U.S. - Four Years (24) Cover: $240	You Pay: $99.99 /	Save: $140 + FREE $198 CD
U.S. - Two Year (12) Cover: $120	You Pay: $49.99 /	Save: $70
Can/Mex - Four Years (24) $240	You Pay: $99.99 /	Save: $140 + FREE $198 CD
Can/Mex - Two Year (12) $120	You Pay: $69.99 /	Save: $50
Int'l - Four Years (24) $240	You Pay: $120 /	Save: $120 + FREE $198 CD
Int'l - Two Year (12) $120	You Pay: $79.99 /	Save: $40

RECEIVE YOUR DIGITAL EDITION ACCESS CODE INSTANTLY WITH YOUR PAID SUBSCRIPTIONS

3-Pack

Pick any 3 of our magazines and save up to $210⁰⁰ Pay only $99 for a 1 year subscription plus a FREE CD

- 2 Year – $179.00
- Canada/Mexico – $189.00
- International – $199.00

6-Pack

Pick any 6 of our magazines and save up to $340⁰⁰ Pay only $199 for a 1 year subscription plus 2 FREE CDs

- 2 Year – $379.00
- Canada/Mexico – $399.00
- International – $449.00

9-Pack

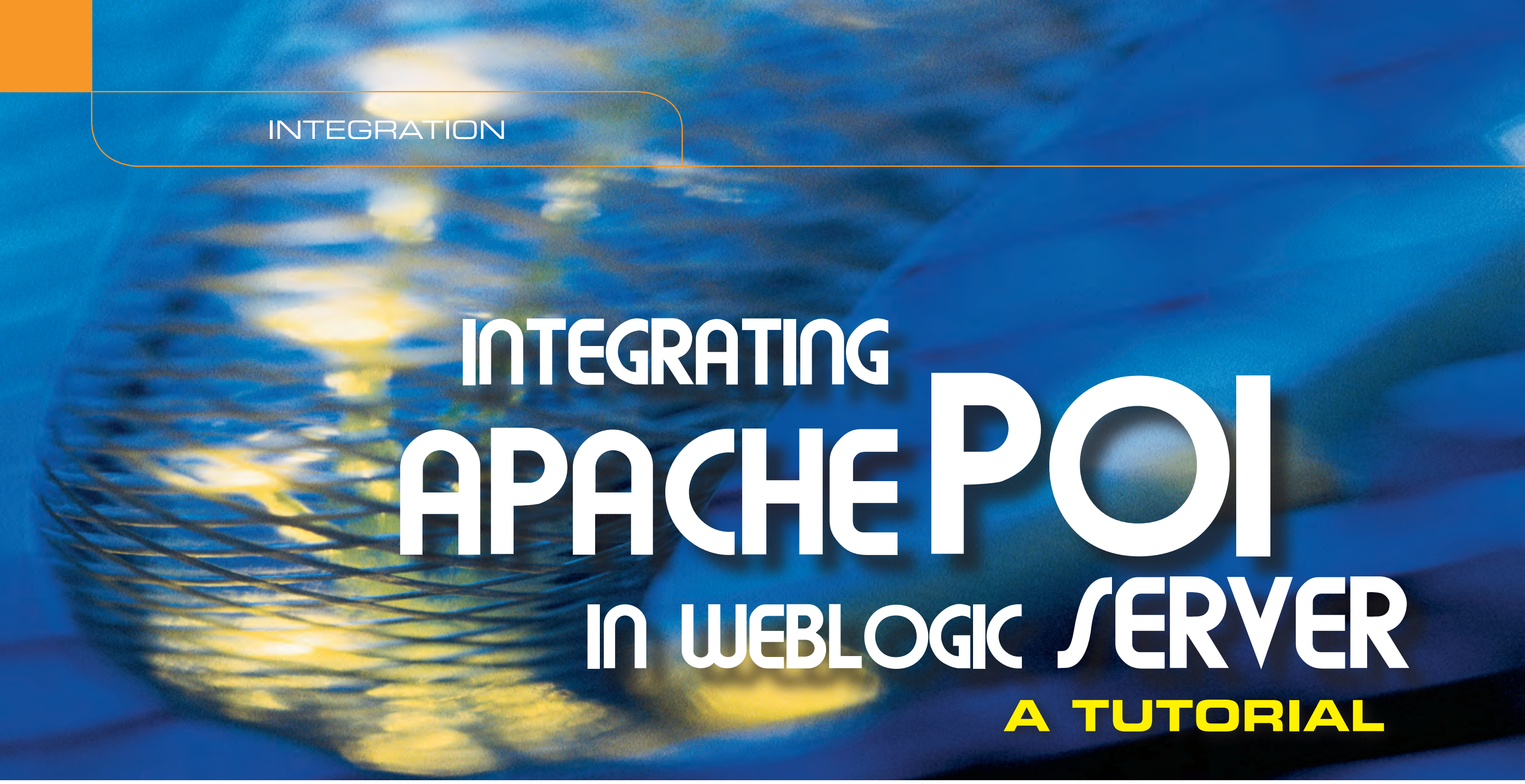
Pick 9 of our magazines and save up to $270⁰⁰ Pay only $399 for a 1 year subscription plus 3 FREE CDs

- 2 Year – $699.00
- Canada/Mexico – $749.00
- International – $849.00

*WHILE SUPPILES LAST. OFFER SUBJECT TO CHANGE WITHOUT NOTICE

Subscribe Online Today **www.sys-con.com/2001/sub.cfm**

SYS-CON MEDIA



INTEGRATING APACHE POI IN WEBLOGIC SERVER

A TUTORIAL

By Deepak Vohra & Ajay Vohra

The Apache Jakarta POI project provides components for the access and generation of Excel documents. The POI HSSF API is used to generate Excel Workbooks and to add Excel spreadsheets to a workbook. An Excel spreadsheet consists of rows and cells. The layout and fonts of a spreadsheet are also set with the POI HSSF API.

A database table is often required to be presented in an Excel spreadsheet. Also, a developer's requirement could be to store an Excel spreadsheet in a database table. The Apache POI HSSF project is an API to create an Excel spreadsheet. The data in the Excel spreadsheet generated with the POI HSSF project may be static data in an XML document, or dynamically retrieved data from a database. Also, an Excel document may be converted to an XML document or be stored in a database. In this tutorial we will discuss the procedure to create an Excel spreadsheet from a MySQL database table in WebLogic Server. Subsequently, an Excel spreadsheet will be stored in a database table.

Preliminary Setup

The implementation of the POI HSSF project is provided in the org.apache.poi.hssf.usermodel package. The org.apache.poi.hssf.usermodel package classes are required in the Classpath to generate an Excel spreadsheet or to parse an Excel spreadsheet. Download the Apache POI library poi-bin-2.5.1-final-20040804.zip file and extract the zip file to an installation directory (http://jakarta.apache.org/poi/). Install WebLogic 8.1 Server. Download the MySQL database (www.mysql.com). Extract the MySQL zip file mysql-4.0.25-win32.zip to a directory. Install the MySQL database. Download the MySQL Connector/J JDBC driver (www.mysql.com/products/connector/j/). Extract the MySQL zip file mysql-connector-java-3.1.10.zip to a directory. Add the MySQL JDBC driver jar file, mysql-connector-java-3.1.10-bin.jar, to the <weblogic81>\samples\domains\examples\startExamplesServer script CLASSPATH variable.

Login to the MySQL database with the DOS command:

```
>mysql
```

Access the example database test with the command:

```
mysql>use test
```

Create an example database table in the MySQL database from which an Excel spreadsheet will be generated. The SQL script to create example table Catalog is shown in Listing 1.

Next, we will add the Apache POI .jar file to the WebLogic Server Classpath and create a JDBC datasource in the WebLogic Server to retrieve data for an Excel spreadsheet.

Add the poi-2.5.1-final-20040804.jar file to the CLASSPATH variable in the <weblogic81>\samples\domains\examples\startExamplesServer script. <weblogic81> is the directory in which the WebLogic Server is installed.

Author Bio:
Deepak Vohra is a Sun Certified Java 1.4 Programmer and a Web developer.

Ajay Vohra is a senior solutions architect with DataSynapse Inc.

Contact:
dvohra09@yahoo.com
ajay_vohra@yahoo.com

Next, create a JDBC connection with the MySQL database in WebLogic Server.
Start the examples server with the script startExamplesServer. Access the administration console with the URL http://localhost:7001/console or with the Administration Console link in the WebLogic Server Examples index. In the administration console, right-click on the examples>Services>JDBC>Connection Pools node and select Configure a new JDBCConnectionPool. Specify the following connection properties to configure a JDBC connection pool:

- The database type: MySQL
- The JDBC driver: MySQL's Driver (Type 4)
- Database name: test
- Host name: localhost
- Port number: 3306
- Driver class name: com.mysql.jdbc.Driver
- Connection URL: jdbc:mysql://localhost:3306/test

Next, configure a JDBC datasource in the administration console. Right-click on the examples>Services>JDBC>DataSources node and select Configure a new JDBCTxDataSource. In the "Configure the data source" frame, specify a datasource name and a JNDI Name – MySqlDS, for example. In the "Connect to connection pool" frame select the Connection Pool previously configured with the MySQL database. In the "Target the datasource frame," select the examplesServer. A datasource gets configured with the MySQL database.

Generating an Excel Document with Apache POI

In this section we will generate an Excel spreadsheet from the example database table. First, create a JSP application to generate an Excel spreadsheet.

In the JSP an Excel spreadsheet will be created from a MySQL database table. The Apache POI HSSF API is used to generate an Excel spreadsheet. The Apache POI HSSF package has classes for the different components of an Excel spreadsheet. Some of the commonly used classes of the Apache POI HSSF package are listed in Table 1.

First, import the Apache POI HSSF package:

```
<%@ page import="org.apache.poi.hssf.usermodel.*, java.sql.*, java.
io.*,javax.naming.InitialContext"%>
```

Create an Excel stylesheet workbook:

```
HSSFWorkbook wb=new HSSFWorkbook();
```

TABLE 1

Class Name	Description
HSSFWorkbook	Represents an Excel spreadsheet workbook
HSSFSheet	Represents an Excel spreadsheet
HSSFHeader	Specifies a spreadsheet header
HSSFRow	Specifies a spreadsheet row
HSSFCell	Specifies a spreadsheet cell
HSSFCellStyle	Specifies the cell style
HSSFFont	Specifies the font for the spreadsheet
HSSFChart	Respresents a chartHSSFDateUtil Specifies Excel dates
HSSFPrintSetup	Specifies the print setup for an Excel document

Apache POI HSSF Package Classes

Next, create an Excel spreadsheet:

```
HSSFSheet sheet1=wb.createSheet(“sheet1”);
```

The data for the stylesheet is retrieved from a MySQL database table. Obtain a JDBC connection from the database. The JDBC connection is obtained with the datasource JNDI MySqlDS.

```
InitialContext initialContext = new InitialContext();
javax.sql.DataSource ds = (javax.sql.DataSource)
initialContext.lookup(“MySqlDS”);
java.sql.Connection conn = ds.getConnection();
```

Create a java.sql.Statement and get a result set from the example table Catalog:

```
Statement stmt=conn.createStatement();
ResultSet resultSet=stmt.executeQuery(“Select * from Catalog”);
```

Create a header row for the Excel spreadsheet. The rows in an Excel spreadsheet are "0" based.

```
HSSFRow row=sheet1.createRow(0);
```

Set the header row cell values corresponding to the table columns. The row cells are also "0" based. For example, the value for the first cell in the row is set with the setCellValue method to CatalogId.

```
row.createCell((short)0).setCellValue(“CatalogId”);
```

To add rows to the spreadsheet, iterate over the result set and add a row for each of the table rows. Retrieve the column values from the ResultSet and set the values in the row cells.

```
for (int i=1;resultSet.next(); i++)
        {
 row=sheet1.createRow(i);
row.createCell((short)0).setCellValue(resultSet.getString(1));
row.createCell((short)1).setCellValue(resultSet.getString(2));
row.createCell((short)2).setCellValue(resultSet.getString(3));
row.createCell((short)3).setCellValue(resultSet.getString(4));
row.createCell((short)4).setCellValue(resultSet.getString(5));
}
```

Create a FileOutputStream to output the Excel spreadsheet to an XLS file. An XLS file represents an Excel spreadsheet:

```
FileOutputStream output=new FileOutputStream(new File(“c:/excel/catalog.
xls”));
```

Output the Excel spreadsheet to an XLS file:

```
wb.write(output);
```

The ExcelWebLogic.jsp JSP used to generate an Excel spreadsheet is available in the References section.

To run the ExcelWebLogic.jsp JSP in the WebLogic Server, copy the JSP to the <weblogic81>\samples\server\examples\

build\mainWebApp directory. Run the JSP with the URL http://localhost:7001/ExcelWebLogic.jsp.

An Excel spreadsheet gets generated which may be opened in Excel (http://office.microsoft.com/en-us/FX010858001033.aspx) or the Excel Viewer tool (http://office.microsoft.com/en-us/assistance/HA011620741033.aspx).

Storing an Excel Document in a Database Table

In this section we will store an Excel spreadsheet in a MySQL database table with the Apache POI API. The example Excel document stored is the spreadsheet, catalog.xls, which was generated in the previous section. The Excel spreadsheet is stored in MySQL table Catalog. Drop the Catalog table from which the Excel document was generated in the previous section with the MySQL command:

```
MySQL>DROP table Catalog;
```

Develop a JSP application to store the example Excel document in the MySQL database. In the JSP application import the Apache POI packages org.apache.poi.poifs.filesystem and org.apache.poi.hssf.usermodel. The org.apache.poi.poifs.filesystem package has classes to create an Excel workbook and the org.apache.poi.hssf.usermodel package has classes that represent an Excel workbook, spreadsheet, spreadsheet row, and row cell.

```
<%@ page import=”org.apache.poi.poifs.filesystem.*, org.apache.poi.hssf.
usermodel.*, java.sql.*, java.io.*,javax.naming.InitialContext”%>
```

As in the previous section, obtain a JDBC connection from the MySQL datasource:

```
InitialContext initialContext = new InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource)
    initialContext.lookup(“MySqlDS”);
   java.sql.Connection conn = ds.getConnection();
```

Create java.sql.Statement from the JDBC connection:

```
Statement stmt=conn.createStatement();
```

Create a MySQL table in which the Excel spreadsheet will be stored:

```
String createTable=”CREATE TABLE Catalog(CatalogId VARCHAR(25) PRIMARY
KEY,Journal VARCHAR(25),Section VARCHAR(25),Edition VARCHAR(25),Title
Varchar(125),Author Varchar(25))”;

stmt.execute(createTable);
```

Create a POIFSFileSystem to read the Excel document:

```
File catalogExcel=new File(“C:/ExcelWebLogic/catalog.xls”);
FileInputStream inputStream=new FileInputStream(catalogExcel);
POIFSFileSystem fileSystem=new POIFSFileSystem(inputStream);
```

Obtain a HSSF workbook from the POIFSFileSystem:

```
HSSFWorkbook wb=new HSSFWorkbook(fileSystem);
```

Obtain an Excel spreadsheet from the Excel workbook:

```
HSSFSheet sheet1=wb.getSheet(“sheet1”);
```

Iterate over the rows in the spreadsheet with a row iterator:

```
java.util.Iterator rowIterator=sheet1.rowIterator();
HSSFRow row=(HSSFRow)rowIterator.next();
```

Retrieve the row cell values for each of the rows. For example, the CatalogId row cell value is retrieved with:

```
String catalogId=row.getCell((short)0).getStringCellValue();
```

Add a table row for each of the rows in the Excel spreadsheet:

```
String exceltable=”INSERT INTO Catalog VALUES(“+”\’”+catalogId+”\
’”+”,”+”\’”+journal
+”\’”+”,”+”\’”+section+”\’”+”,”+”\’”+edition+”\’”+”,”+”\’”+title+”\
’”+”,”+”\’”+author+”\’”+”)”;
stmt.execute(exceltable);
```

Copy the POIWebLogic.jsp to the <weblogic81>\samples\server\examples\build\mainWebApp directory. Run the JSP with the URL http://localhost:7001/POIWebLogic.jsp. A MySQL database table gets generated from the Excel spreadsheet. The POIWebLogic.jsp used to generate a database table from an Excel spreadsheet is available in the References section.

Conclusion

In this tutorial, an Excel spreadsheet was generated from a MySQL database table and subsequently the spreadsheet was stored in a database table. The WebLogic Server facilitates the conversion from database table to Excel spreadsheet and from spreadsheet to database table by providing a datasource and a J2EE application server to run a JSP application.

Listing 1: Catalog.sql

```
CREATE TABLE Catalog(CatlogId VARCHAR(25)
PRIMARY KEY, Journal VARCHAR(25), Section VARCHAR(25),
 Edition VARCHAR(25),  Title Varchar(125), Author Varchar(25));

INSERT INTO Catalog VALUES(‘catalog1’, ‘dev2dev’,
‘WebLogic Platform 8.1’,
 ‘Oct 2004’,  ‘BEA WebLogic Platform 8.1 SP3 Evaluation Guide’,
‘dev2dev’);

INSERT INTO Catalog VALUES(‘catalog2’, ‘dev2dev’,
 ‘WebLogic Server’,
 ‘Feb 2005’,  ‘Application Architecture for Applications Built on
 BEA WebLogic Platform 8.1’, ‘Bob Hensle’);

INSERT INTO Catalog VALUES(‘catalog3’, ‘dev2dev’,
‘WebLogic Integration’,
 ‘March 2005’,  ‘The BEA WebLogic Platform and Host Integration’,
‘Tom Bice’);
```

ENGAGE AND EXPLORE...

The Technologies, Solutions and Applications that are Driving Today's Initiatives and Strategies...

CALL FOR PAPERS NOW OPEN!

SOA WebServices Edge 10th International conference+expo Edge 06

June 2006 | New York, NY

The Sixth Annual SOA Web Services Edge 2006 East - International Web Services Conference & Expo, to be held June 2006, announces that its Call for Papers is now open. Topics include all aspects of Web services and Service-Oriented Architecture

Suggested topics...

- > *Transitioning Successfully to SOA*
- > *Federated Web services*
- > *ebXML*
- > *Orchestration*
- > *Discovery*
- > *The Business Case for SOA*
- > *Interop & Standards*
- > *Web Services Management*
- > *Messaging Buses and SOA*
- > *Enterprise Service Buses*
- > *SOBAs (Service-Oriented Business Apps)*
- > *Delivering ROI with SOA*
- > *Java Web Services*
- > *XML Web Services*
- > *Security*
- > *Professional Open Source*
- > *Systems Integration*
- > *Sarbanes-Oxley*
- > *Grid Computing*
- > *Business Process Management*
- > *Web Services Choreography*

CALL FOR PAPERS NOW OPEN!

2006 ENTERPRISE> OPENSOURCE CONFERENCE+EXPO

June 2006 | New York, NY

The first annual Enterprise Open Source Conference & Expo announces that its Call for Papers is now open. Topics include all aspects of Open Source technology. The Enterprise Open Source Conference & Expo is a software development and management conference addressing the emerging technologies, tools and strategies surrounding the development of open source software. We invite you to submit a proposal to present in the following topics. Case studies, tools, best practices, development, security, deployment, performance, challenges, application management, strategies and integration.

Suggested topics...

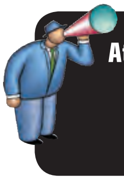
- > *Open Source Licenses*
- > *Open Source & E-Mail*
- > *Databases*
- > *ROI Case Studies*
- > *Open Source ERP & CRM*
- > *Open-Source SIP*
- > *Testing*
- > *LAMP Technologies*
- > *Open Source on the Desktop*
- > *Open Source & Sarbanes-Oxley*
- > *IP Management*

Submit Your Topic Today! www2.sys-con.com/events

Sponsored by

WebServices JOURNAL · XML JOURNAL · .NET DEVELOPER'S JOURNAL · eclipse developer's journal · WebSphere JOURNAL · Information STORAGE+SECURITY journal · wldj · JDJ · LinuxWorld MAGAZINE · MX developer's journal · asp.netPRO · SD Times · CoDe · Software Test & Performance

*Call for Papers email: jimh@sys-con.com



Attention Exhibitors:

An Exhibit-Forum will display leading Web services and OpenSource products, services, and solutions

For Exhibit and Sponsorship Information ► Call 201 802-3066

Produced by SYS-CON EVENTS

Test-Driven Portal Application Development

COMBINING BASIC SOFTWARE ENGINEERING PRINCIPLES WITH NEW TOOLS AND TECHNOLOGIES

By Vidar Moe

With the advent of BEA WebLogic Portal 8.1, a host of new technologies was introduced. These are, among others: Java Page Flow with annotations, Java Controls, and a new IDE to support it. Online tutorials were also thrown into the package to show how the new technologies were supposed to be put to work in the most effective way.

We have used BEA WebLogic Workshop, Java Page Flow, and Java Controls to create a large portal application that consists of about 200 pages covering five different functional areas. Before we started we did an evaluation of the technologies. We aimed to create an architecture that lets us do test-driven development with focus on the following software engineering principles:

- Loose coupling
- Layered architecture with clear responsibilities for the individual layers
- Maintainable code through simple and small classes
- Unit testable code
- Continuous integration

The BEA reference documentation does not explicitly cover how to achieve these characteristics in a BEA Portal application. This means that we had to find a way to incorporate these values taken from traditional software engineering into the new BEA technologies. In this article we will discuss ways of fulfilling these values through a Java Page Flow and Java Controls architecture.

An Extended Page Flow Architecture

The main reason for creating loose coupling between components in architecture is to make sure changes can be introduced with as small an impact as possible for everything but the changing component itself. The basic Java Page Flow architecture is shown in Figure 1.

As we can see from Figure 1, this architecture has a clear layered approach. However, this basic architecture is too simple for all but the smallest projects. A typical Web application use-case consists of three main parts:

- Prepare a screen with some pre-filled data and a form and present this to the user
- User enters data into the form
- The data are validated and sent to a back-end system

It is often necessary to add extra code responsible for converting data from the back-end

Author Bio:
Vidar Moe is a senior consultant at Capgemini Norway, providing technical consulting to customers on J2EE architectures and application development. He has been working with enterprise Java solutions on the BEA WebLogic Platform for several years.

Contact:
vidar.moe@capgemini.com

system to a presentable form, to perform complex validations and converting data back to the back-end system's format. This suggests introducing a new layer in the application. We have chosen to call this the Action Delegate layer (see Figure 2).

The Action Delegate layer is responsible for orchestrating the application's logic. This means that we are delegating the orchestration from the individual action methods in the Page Flow Controller (PFC) to corresponding methods in the ActionDelegate classes. Our experience is that it in normal cases is preferable to have one ActionDelegate class per PFC. This gives us a good division of responsibility. The PFC's action methods now become facades. Their sole purpose is to delegate their work to the ActionDelegate, and to forward to the appropriate jsp/nested page flow based on the return value of the ActionDelegate.

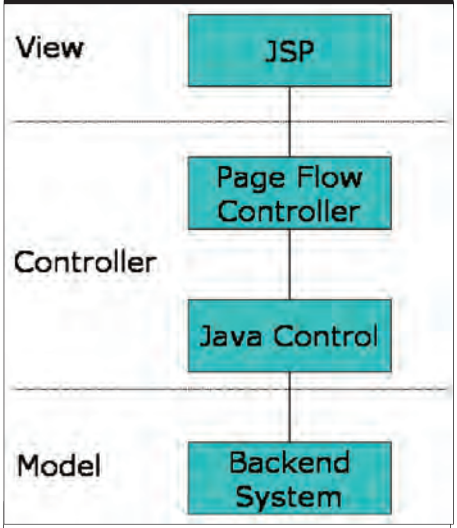
If the PFC has many action methods, the ActionDelegate layer gives us the possibility of using several ActionDelegates from one PFC to group logically connected functionality, and to keep the ActionDelegates small and maintainable. A point to note here is that if you find yourself having a lot of action methods in your PFC, you could probably look at refactoring it into one or more nested page flows.

Another way of reducing the size of the PFC is to give up on using the built-in functionality for creating form beans. The form beans created by the Workshop Wizard are realized as static inner classes in the corresponding PFC. You can perfectly well create your own Form Beans and use them instead. This also gives you cleaner code because you get less code duplication.

Testable Code

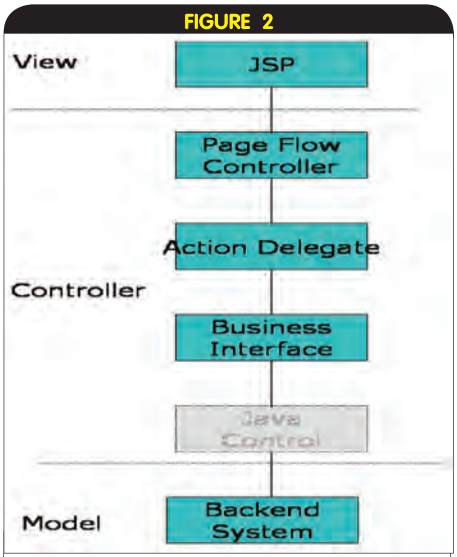
The Java Controls concept is great. It lets you easily access different back-end technologies by calling regular Java methods. However there are some limitations that make them less flexible than they ought to be. Java Controls cannot be instantiated outside of the BEA WebLogic container. This goes for PFCs as well. Java Controls can only be instantiated in three places – in a PFC, a Workflow, or from another Java Control. This presents challenges when it comes to creating JUnit tests for both Java Controls and PFCs – your tests must run inside your container. This conflicts with the principle of having as short a development-build-test cycle as possible.

FIGURE 1



Basic layered architecture

FIGURE 2



Extended layered architecture

There is, however, one upside to this: the fact that Java Controls MUST be instantiated in the PFC forces dependency injection of Java Controls into our ActionDelegates! You can easily add a business interface to your Java Control. It is this interface that you will use as a parameter to your ActionDelegate to implement the dependency injection. This brings us to another advantage of the ActionDelegate layer: it makes it possible to test the actual functionality of the PFCs outside of the container in regular JUnit test cases. Remember that the PFC action methods now only serve as facades – the real work is done in the corresponding ActionDelegate methods. What you typically will do when creating a JUnit test for a PFC action method is to

1. Create mock versions of any Java Controls this method is using
2. Set up the correct values in the HttpServletRequest and HttpSession mock objects

3 Instantiate the ActionDelegate class and call the actual method with the mock objects

4 Assert on the method's return value or changes in modified data in the request, session, or elsewhere

Loose Coupling

As stated above, one of the good reasons for creating loose coupling between components in an architecture is to make sure changes can be introduced with as small an impact as possible for everything but the changing component itself. This translates into (among other things) the ability to replace the whole component with another. One place where this is especially useful is with regard to the integration layer components. This makes it possible to easily switch between the real integration layer components going toward the real back-end systems, and for example an integration layer with components going toward a back-end system simulator. It can be vital to be able to easily switch between live and simulator components, if for example you are developing towards a back-end system layer that is also under development.

A way of realizing this type of loose coupling between components is to use an inversion of control (IoC) framework like Spring that hosts its own bean container. Spring lets you configure which components will be created through an XML configuration file. Let's go back to the back-end simulator example: if you would like to toggle between the integration component going towards the real back-end system and the integration component going towards the back-end system simulator, all you would have to do is to change the entry for the integration component's caller in the XML file.

Sadly, it is hard to get ordinary Java Controls to take part in such an architecture. The reason for this is that Java Controls suffer under the fact that they cannot be created outside of a PFC, a Workflow, or another Java Control. You cannot get the Spring IoC Container to instantiate a Java Control.

Luckily, BEA has spawned an open source project, which takes the Java Controls concept further. The project is called Beehive (see the first entry in the References section) and the Beehive Java Controls have another implementation model rather than BEA's original Java Controls. The Beehive Java Controls are ordinary POJOs, and thus they can be instantiated from the Spring Bean Container. BEA WebLogic Server 9 will also use the POJO Java Controls implementation model, so with this version Spring integration will be a lot easier. Spring will actually become a first-class citizen within BEA WebLogic Server 9, thereby making it possible to monitor Spring beans both from the WebLogic console, and through JMX (see the second entry in the References section).

Code Quality and Continuous Integration

To be able to get IDE help for refactoring, syntax and semantics checking, and code generation, we used Eclipse in addition to WebLogic Workshop. We used Workshop for Portal, JSP, Page Flow, and Java Controls development. ActonDelegates with helper classes, JUnit tests, and mock objects we coded in Eclipse. Eclipse has better support for refactoring, code checking, and code generation, so this was a natural choice for us. To get the code as close to our coding standard as possible, we configured our coding standard into the Eclipse code analyzer, getting real-time feedback on possible violations in the code. There is a small overhead involved in using two IDEs, but we feel that it was well worth it.

BEA has seen the advantages of Eclipse as well, and the next version of Workshop, Workshop Studio 3.0, comes with Eclipse plugins. It will then be possible to perform the whole development in one tool, getting the best of both worlds.

To get even more automatic feedback on the state of the code, both when it comes to passing JUnit tests as well as different aspects of code quality and code coverage, we have used Maven and CruiseControl to implement continuous integration. BEA WebLogic Workshop provides Ant tasks for building a portal application, so it is easy to integrate this into a Maven build process. We created goals that compiled our "Non-workshop" modules first, then copied these into APP-INF/lib and WEB-INF/lib, depending on their responsibilities, and then finally built the complete Portal application using the Ant tasks from workshop. We based our project's directory structure on Vincent Massol's "Enterprise builds" presentation from TSSS 2004 (see the third entry in the References section). This is a logical and pragmatic structure, and has worked very well for us.

Conclusion

When new technologies are introduced, they often come with a set of new development practices. We tend to forget the basic craftsmanship and the existing best practices that are the basis for all professional software development. By focusing on the sound software engineering principles of test-driven development, we have been able develop a large BEA WebLogic Portal application using a host of new technologies, achieving a loosely coupled architecture with high-quality code developed through continuous integration.

References

- Apache Beehive: http://beehive.apache.org/
- Spring Integration with WebLogic Server: http://dev2dev.bea.com/pub/a/2005/09/spring_integration_weblogic_server.html
- Vincent Massol's TSSS 2004 Enterprise Builds Presentation: www.pivolis.com/pdf/Enterprise_Builds_V1.0.pdf

›Linux ›Java ›Web Services ›.NET ›XML ›Wireless ›Storage ›Security
The Premier Resource for Today's Corporate & IT Decision Makers
IT solutions
›GUIDE
WWW.SYS-CON.COM/IT
TECHNOLOGIES YOU NEED NOW!
How to Manage Your Ideas Using Today's i-Technologies
Top IT Solution Providers!
› Candle
› ClearNova
› Computer Associates
› Creative Science
› DataSoft
› GraphOn
› InterSystems
› ITKO
› Merant
› O'Reilly
› Parasoft
› QuadBase
› SAP
› Wily Technology
› Zero G

The World's Leading Java Resource Is Just a >Click< Away!

JDJ is the world's premier independent, vendor-neutral print resource for the ever-expanding international community of Internet technology professionals who use Java.

SYS-CON MEDIA

ONLY $69.99

ONE YEAR
12 ISSUES

Subscription Price Includes FREE JDJ Digital Edition!

OUT OF MEMORY! PAGE 36
Magazine in the World
JDJ
IN THIS ISSUE...
Leveraging Open Source Solutions to Improve Productivity While Using EJBs
PAGE 8
Spring Web Flow
PAGE 14
Intelligent Web Applications with AJAX
PAGE 24
Navigating the Global Enterprise
PAGE 60
JDJ.SYS-CON.COM
ARE PORTALS THE MAGIC BULLET OF WEB APPLICATION DEVELOPMENT?
The many advantages to utilizing portal software
PLUS...
The Performance of EJB 3.0
Dealing with Open Source Software
A Blueprint for Developing Language Tools
THE WORLD'S LEADING i-TECHNOLOGY MAGAZINE JDJ.SYS-CON.COM

www.JDJ.SYS-CON.com

or 1-888-303-5282

OFFER SUBJECT TO CHANGE WITHOUT NOTICE

Scaling AJAX Applications Using Asynchronous Servlets

MULTIPLEXING CLIENT SOCKETS

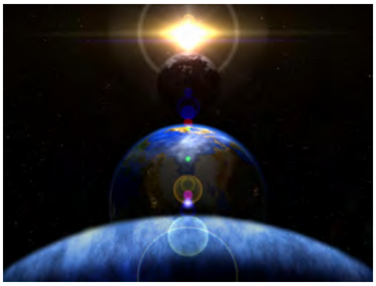
By: Bahar Limaye

The advent of AJAX as a Web application model is significantly changing the traffic profile seen on the server side. The typical Web pattern usage of a user sitting idle on a Web page filling out fields and hitting the submit button to the next link is now transforming into sophisticated client-side JavaScript and rich user interfaces that constantly communicate with the server whenever an event is posted on a form such as a checkbox click, key press, or tab focus.

Think about the amount of data transmitted from the client to the server. From a usability standpoint, the user gets rich user interfaces on a thin-client browser without having to install anything. However, there is a price to pay when scaling these applications on the server. Your typical capacity-planning numbers for AJAX applications can shift significantly in the magnitude of three to four times than that of standard Web applications.

One might ask how would this impact the WebLogic Server. For every HTTP request to WebLogic, an execute thread is consumed. Given the nature of AJAX programming and many short-lived HTTP requests in the form of polling, this behavioral pattern can potentially bombard the server with client requests. For several years now, WebLogic has taken this into consideration and built a wonderful feature called the FutureResponseServlet. This paradigm builds off the notion of asynchronous servlets. From version 6.1 onward, this functionality has allowed developers to have the ability to provide true asynchronous notification from the server without the client polling for events and consuming an execute thread on the server. BEA was not too keen to make this class public until 9.x.

How can one leverage this class in the real world? Well, let's look at an example. Suppose you have a business requirement to build a Web-based application that presents server data in near-real time without refreshing the browser. Such an application can submit a request to the server that can take a long time to process, and still be able to receive asynchronous events about its status, as well as listen for events. From a technology standpoint, there are many ways you can build this. One way is to use a Java Applet that communicates with a Java Servlet to get asynchronous information. This is nice but becomes inconvenient to the user because they have to download a JVM and carry the baggage and weight of downloading an applet to the browser. Moreover, a persistent socket connection must be maintained from the client to the server to receive asynchronous events. Imagine: if there are 1,000 users using the applet, there are 1,000 execute threads that are mostly idle waiting to send event notifications back to the client. Yes, there are other approaches such as building polling mechanisms from an applet or AJAX application to check for new data every so often. If data is not received that often, it is useless to poll and waste server resources and consume execute threads. Instead, the server can poll periodically and relay events back to the client and maintain the socket threads without consuming a persistent execute thread. This is very similar to how Java NIO works. Ideally, you would want to build an application, whether it is an applet or a thin AJAX-based Web application, that "asynchronously" receives event notifications from the server without consuming a persistent execute thread on the server.



One solution to this problem is to create a servlet that extends the FutureResponseServlet class. The browser establishes a single

Author Bio:
Bahar Limaye is a system architect at The College Board. He has extensive experience building distributed object-oriented systems.

blimaye@collegeboard.org

connection to the FutureResponseServlet and registers itself as a listener in a different thread. Whenever an event is received on the server, the thread notifies the client with the event. The server maintains the client asynchronously without having to consume a persistent execute thread. This model can scale several concurrent users.

This article does not describe how to build an AJAX application. There are several articles available that discuss this topic. It discusses the importance of asynchronous processing for presentation layers, such as AJAX, applets, or any front-end application. Listing 1 shows an example.

As you can see, this example is extremely trivial. The AsynchronousServerResponseServlet class extends FutureResponseServlet and overrides the service method. A single thread, the Notifier class, processes all client connections response. For every HTTP request, the servlet registers the socket connection to the Notifier thread and returns. Asynchronous events get delivered to the client while the persistent socket connection is maintained.

A single thread can manage multiple client connections!

The run() method can be used to call back events to the client based on some message selection criteria. This example just performs a server-side push operation and is very simplistic in nature. Thread pools can be utilized for certain types of event processing.

In conclusion, when processing long running tasks, the FutureResponseServlet is a good feature that allows developers to increase performance and process responses in separate threads and minimize overhead. This approach allows scalability when building asynchronous applications.

Listing 1

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.Stack;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;

import weblogic.servlet.FutureResponseServlet;
import weblogic.servlet.FutureServletResponse;

// An AsynchronousServlet that handles HTTP requests from a "separate" 
thread and
// not the execute thread used to invoke this servlet.
public class AsynchronousServerResponseServlet extends FutureRespons-
eServlet {
   private final Notifier notifier;

   public AsynchronousServerResponseServlet() {
          this.notifier = new Notifier();
          this.notifier.start();
   }

   public void service(HttpServletRequest request, FutureServletRe-
sponse response)
          throws IOException, ServletException {

          // push this client's request to a buffer and return immedi-
ately.
          // asynchronous processing occurs in the run method of the 
Notifier Thread
          notifier.poll(request, response);
   }

   class Notifier extends Thread {

          private static Stack clients = new Stack();

          void poll (HttpServletRequest request, FutureServletResponse 
response) {
                    clients.push(new Client(request, response));
          }

          public void run() {
                    while (!clients.empty()) {
                              Client client = null;
                              try
                              {
                                        client = (Client) clients.
pop();
                                        PrintWriter pw = client.
response.getWriter();
                                        for(int j = 0; j < 10; j++) {
                                                  pw.println("Time 
is:" + new Date() + "<BR>");
                                                  pw.flush();

                                        }
                                        pw.close();
                              } catch(Throwable t) {
                                        t.printStackTrace();
                              } finally {
                                        try {
                                        client.response.send();

                                        } catch(IOException ioe) {
                                        ioe.printStackTrace();
                                        }
                              }

                    }
          }
   }

   // inner class that holds on to the clients http request and re-
sponse
   class Client {
          private HttpServletRequest request;
          private FutureServletResponse response;

          private Client(HttpServletRequest request, FutureServletRe-
sponse response) {
                    this.request = request;
                    this.response = response;
          }
   }
}
```

If a Resource Thread Hangs in the Portal

WORKAROUND SOLUTION

By Michael Poulin

Author Bio:
Michael Poulin is working as a technical architect for a leading Wall Street firm. He is a Sun Certified Architect for Java Technology. For the past several years Michael has specialized in distributed computing, application security, and SOA.

Contact:
mpoulin@usa.com

This article describes a workaround design that allows a Portal to survive if its resource starts hanging request threads.

Business Task

How frequently does your Portal experience user requests hanging in the resource? Not frequently, I hope. However, if this happens and the resource continues hanging user requests, the Portal is exposed to a fatal risk of spending all of the configured concurrent user requests and eventually dies. This is a disaster. I faced such situation a few times and decided to protect my Portal from even rare surprises such as these.

Let a Portal include several Portlets with login control provided via a backing file for each Portlet. The backing file is created per request and is thread-safe with regard to the user requests. In the login control, the baking file's `init()` method (or `preRender()` method) calls an API of separated security service (resource) to obtain a resource access authorization. For a business processing, Portlets also delegate user requests via other API calls to the business layer. Everything works fine until a call to the resource is not returned to the Portlet, i.e., it hangs somehow somewhere. Here is a concrete example.

We assume that WebLogic Portal uses EJB as a resource in the business layer and/or in security service. The EJB operates on other resources and reports processing status by sending a message via JMS. The EJB is deployed in a cluster together with other applications. Portal is deployed on a different physical machine. One of the co-deployed applications causes memory error and the whole cluster, including our EJB, hangs, i.e., it does not return requests and does not throw exceptions. Actually, it is not necessary to discuss such dramatic failure: "hanging" mode, from a Portal perspective, is just a response that returns too slowly, with too much latency to be acceptable in the dynamic concurrent life of the Portal. Latency may be caused, for instance, by server overload or problems with databases, but it is irrelevant – the threads in backing files run longer than allowed for normal Portal work and the "maximum number of concurrent users" in the Portal is reached. The Portal stops accepting user requests at all – this is the problem.

Solution Design

The first thing that came to my mind was to set a time-out on the RMI client, i.e., the EJB client, used by the Portlet to access its resource (`remote-client-timeout`). However, the WebLogic Guidelines on Using the RMI Timeout list several restrictions on such time-outs, including "No JMS resources are involved in the call." That is, I am not supposed to use a time-out for my resource EJB. I refer to this case for only one reason: to outline that there are situations where the Portal may be unprotected from hanged resource threads. Even if no restrictions would be applied to the time-out, if requests hang faster than the time-out frees related threads, the problem stays.

I would like to present one of the possible solutions for this problem. The solution is effective on one condition: the Portal has some contents or functions that are independent of potentially hanging resources. That is, Portal can operate with partial functionality.

The solution includes three components: Monitoring, Decision Rule, and the Rule Enforcement Method. The concept of the solution is straightforward: Portal monitors running calls to the resource, henceforth called resource threads, counts the amount of too long running resource threads (riskCounterValue) and applies a Decision Rule such as "If riskCounterValue reaches or exceeds predefined threshold – riskThreshold – all incoming calls to that resource are denied until riskCounterValue becomes smaller than riskThreshold." Due to the Rule, the number of potentially "hanged" resource threads gets limited and the user request may be served with reduced functionality. For example, if a Portal includes four Portlets, and some resource threads for one of the Portlets is considered "at risk of hanging," the Portal can skip the Portlet-in-risk and display just three Portlets to the user.

The implementation of the Rule Enforcement Method is very important. If the rule is enforced in the scope of every call, we may expect performance degradation but gain simplicity in the control of potentially "hanged" resource threads. If the rule is enforced outside of the calls, we can preserve performance but tuning of such control becomes tricky. We will discuss the latter case with details. The diagram in Figure 1 describes it.

As the diagram shows, in the first step the Portal initializes a Helper object that, in turn, initializes a CallRegistry object. The latter may be implemented as a java.util.HashMap and used for registering all calls to the resource API. Then Helper starts a "watchdog" thread. If you use Struts, for example, this thread starts in the Model. The "watchdog" thread periodically reviews records in the CallRegistry, counts the number of too long running calls, and sets it as a riskCounterValue variable in the Helper.

It is assumed that we approximately know normal execution time of API calls. This may be one value for all APIs – the longest duration – or every API may have its individual execution time. Therefore, when an API method is invoked, we can calculate the time at which the API is expected to complete in a normal situation, for example:

```
java.lang.System.currentTimeMillis()
long apiExecutionTime = …;// property
long timeToComplete =
     java.lang.System.currentTimeMillis() +
     apiExecutionTim
```

When a Helper's method is called, it adds a new record into the CallRegistry. The record consists of a unique Call ID (used as a key in the java.util.HashMap) and expected completion time (timeToComplete) for the API (used as a value in the java.util.HashMap). If the method successfully completes, it removes its record from the CallRegistry.

Let's review how a user request is processed. Upon receiving a user request, the Portlet's backing file delegates it to the Helper API method (the latter invokes the resource API). First the Helper API method checks if it may execute. If the riskThreshold is not reached by the moment of the request, the Helper API method continues its work. Otherwise, it throws an exception and the Portal moves to the next function or next API call.

The permission to execute may be given only if the amount of too long running resource threads (riskCounterValue) is less than the riskThreshold. The riskThreshold is set via configuration properties. For example, if the maximum of concurrent user requests is configured as 25, the riskThreshold may be set to 10. That is, the Portal risks only a half of its capability of handling concurrent user requests and it still can operate if resource threads start to hang.

Notice that we do not do anything with too long running API calls. Some of them can eventually complete successfully and Helper API methods will remove their records from the CallRegistry, i.e., the next route of counting may result in lower number than the riskThreshold and the next user request for the resource may not be denied (by throwing an exception).

The Portal cannot know if there is an accidental latency in the network or if the resource thread is really hanging. Because of that, it is recommended to send a notification (e.g., via an e-mail) to the Operation Team if the riskThreshold is reached or exceeded in several sequential control cycles. Received notification will allow the Operation Team to analyze logs promptly, and find and resolve the reason of long running calls in timely manner.

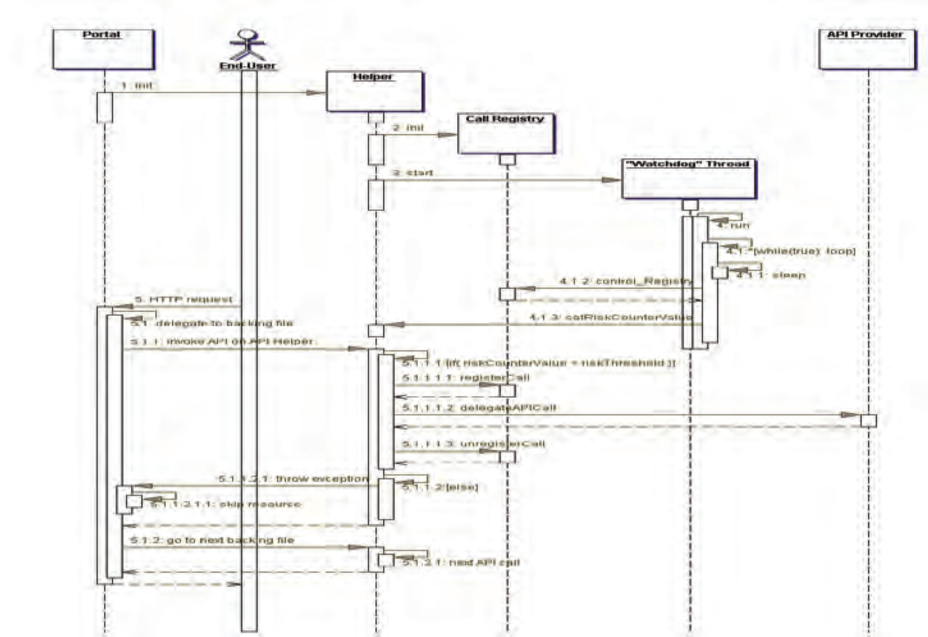
Analysis and Tuning

A control of "hanged" resource threads is quite dynamic and not simple in tuning. Its effectiveness is based on the balance of three parameters:

- ratio of average period of time (TUR) between user requests to period of time (TRC) between "watchdog" thread control cycles – risk control cycles: R = TUR / TRC
- risk threshold (riskThreshold) for a particular resource
- expected time of execution of the resource API calls

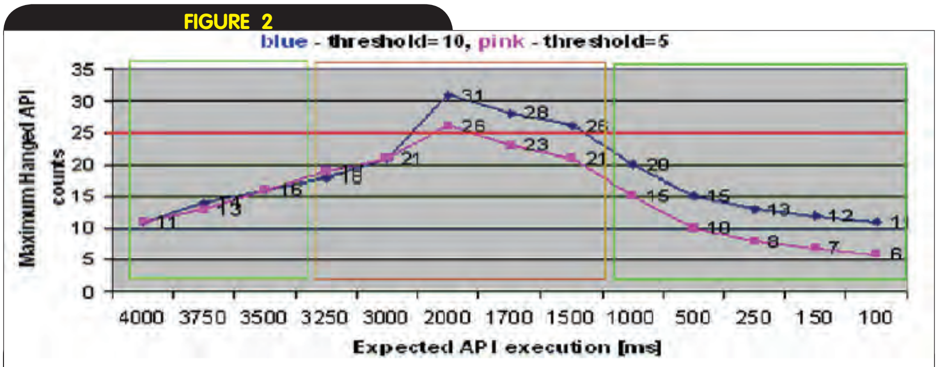
The research and testing of the control have concluded that parameter tunings depend on a particular Portal

FIGURE 1


Portal
End-User
API Provider
Helper
Call Registry
"Watchdog" Thread
1: init
2: init
3: start
4: run
4.1: *[while(true) loop]
4.1.1: sleep
4.1.2: control_Registry
4.1.3: setRiskCounterValue
5: HTTP request
5.1: delegate to backing file
5.1.1: invoke API on API Helper
5.1.1.1 [if(riskCounterValue < riskThreshold)]
5.1.1.1.1: registerCall
5.1.1.1.2: delegateAPICall
5.1.1.1.3: unregisterCall
5.1.1.2 [else]
5.1.1.2.1: throw exception
5.1.1.2.1.1: skip resource
5.1.2: go to next backing file
5.1.2.1: next API call

Monitor for long-running resource threads – sequence diagram

FIGURE 2



Effectiveness of "watchdog" thread control

implementation but have common tendencies. The graph in Figure 2 demonstrates guidelines for the tuning. In the tests, the ratio was set as R = 95% where TUR was 95 [ms] while TRC was set as 100 [ms]. In general, a reliable ratio is 90% and higher.

The graph shows how the number of "hanged" API calls, counted in the control, depends on call execution time. Points on the graphs represent maximum numbers of user requests "hanged" in between risk control cycles, i.e., maximum of riskCounterValue in the series of tests for given call execution time. Remember that some user requests are denied when the Decision Rule is enforced and the number of "hanged" resource threads does not increase.

The horizontal red line in Figure 2 marks the amount of allowed maximum concurrent users in the Portal. The purpose of the control is to keep maximum riskCounterValue strictly below the red line. The closer the points in the graphs are to the red line, the more probability that the riskThreshold may be reached or exceeded.

As we can see, the behavior of the control is not obvious. For some values of call execution time (from 3250 ms to 1500 ms), the control yields user requests and number of "hanged" resource threads gets close to and exceeds the allowed maximum of concurrent users in the Portal. This is the interval during which the control is ineffective in the given conditions. At the same time, there are two intervals – from 100 ms to 1000 ms and from 3500 ms to 4000 ms – where the control is effective: the Decision Rule with the particular riskThreshold reliably protects Portal from "hanged" API calls and leaves enough concurrent request threads to serve other user requests.

The graph also shows that smaller riskThreshold provides better protection. However, if riskThreshold for a resource is set too low, the resource may become unavailable in most of user sessions just due to the slight fluctuations in the network latency. This is another subject for balance and tuning.

Conclusion

The proposed solution of run-time control of "hanged" resource calls allows a Portal to isolate the resources that are in trouble and to continue its work with remaining resources making a minimal impact on the performances. The solution effectiveness depends on several tuning parameters: the ratio between request frequency and frequency of the risk control cycles, the value of the risk threshold, and the expected call execution time.

Tuning is not a trivial task in this case – it requires intensive testing. Moreover, the numbers given in the article are specific to my test Portal and you should expect other values in the tests on your Portal though you find the same dependencies. On the other hand, if certain performance degradation is acceptable, it is recommended to perform risk control cycles in the scope of every API call that significantly simplifies tuning of the solution parameters.

References

- WebLogic RMI Features and Guidelines: Guidelines on Using the RMI Timeout http://e-docs.bea.com/wls/docs81/rmi/rmi_api.html
- Nyberg, G., Patrick, R., Bauerschmidt, P., McDaniel, J., and Mukherjee, R. "Mastering BEA WebLogic Server: Best Practices for Building and Deploying J2EE Applications." Wiley E-Book published March 2004. ISBN: 0-471-48090-8.

Top 10 Reasons Why You Should Upgrade to WebLogic 9

WHAT ARE YOU WAITING FOR?

By Hank Li & Henry Chen

Author Bio:
Hank Li, PhD, has more than 10 years of software development and project management experience. He works as DRE for BEA Systems. His new hobby is AJAX/WEB 2.0 technologies.

Henry Chen, PhD, has been working for BEA for six years supporting enterprise customers on various BEA products, and currently is the senior manager of the Worldwide Service Division.

Contact: hli@beacom henryc@bea.com

WebLogic Server 9.0 is the most significant BEA Java application server release to date. Fully compliant with J2EE 1.4, this release tackles head-on the biggest challenge facing enterprise networks today: reducing overall cost of management and operations while delivering high reliability, continuous uptime, scalability, and mission-critical integration solutions.

Since WebLogic 9.0's released, numerous people have said "Give me three reasons to upgrade to WebLogic 9.0." Because WebLogic 9.0 is so feature-rich and performance-driven, we can easily come up with 10 major reasons.

Reason 10: Enhanced Web Services and SOA Architecture Now SOA Comes to its Prime Time

WebLogic 9.0 delivers a consolidated and fully integrated Web services stack. BEA has pioneered some of the significant technologies in the Web services field, such as annotations-based Web services programming and conversational Web services. Web services conform to all of the requisite J2EE and most of the important WS-* specifications. Support for asynchronous, conversational, reliable, and secure Web services has also been enhanced.

In a world of new Web services, you will enjoy:

- A next-generation Web services programming model
- Improved performance for conversational Web services
- More flexible, secure, and reliable asynchronous Web services
- Support for long-running, asynchronous, and reliable message exchange
- Reduced complexity
- A streamlined and standards-based programming model via JSR-181 JWS-based Web services
- Eases authoring of Web services

Reason 9: JMS – The Proud Enhancement of WebLogic 9

WebLogic Server 9.0 introduces major changes in the configuration, deployment, and dynamic administration of WebLogic JMS. It officially supports the JMS 1.1 specification. What is more, the long waiting advanced message ordering feature is added to the system. The XML API has been enhanced for XML message processing. On WebLogic 9.0 platform, using JMS is fun, reliable, and fast. Below are a few highlights of the exciting new features.

Automatic JMS Failover

Automatic JMS failover is a long waiting feature in the industry. JMS leverages the feature "Automatic WebLogic Server Migration" to provide automatic JMS failover. JMS will automatically get failed over when the whole WebLogic Server instance is failed over. Some other JMS server providers have provided such functions with some tricky setup, but the WebLogic 9.0 implementation is most straightforward and clear.

Unit of Order

Message ordering is one of the fundamental requirements of major messaging applications. WebLogic Server JMS will guarantee the sequential processing of messages even in the cluster environment. It is even possible to define multiple groups to group the messages so that each group has its own sequence for processing.

Store and Forward (SAF)

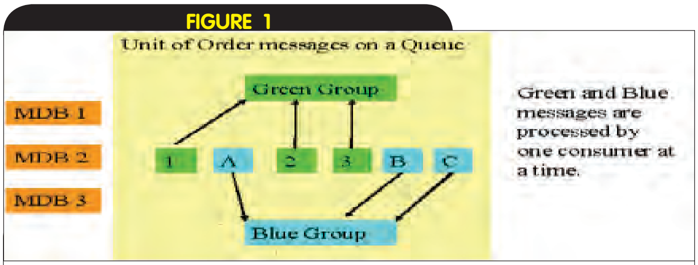
The WebLogic store and forward (SAF) service enables WebLogic Server to deliver messages reliably between applications that are distributed across WebLogic Server instances. The power of the SAF makes it easy link multiple message services together.

Reason 8: Side-by-Side Deployment – A Dream Comes True

Every J2EE developer has experienced the pain of releasing a new version of the product. After numerous development cycles and QA cycles, finally it is time to deploy the new version to the application server. A weekend is booked to shut down the servers, swap the new version of product in, and then pray there is no issue on Monday. If you are lucky, only a few issues will show up. However if you are not in luck then you have to roll back the old applications, and it is even more painful than deploying the new version, because sometimes you can not roll back all of the changes.

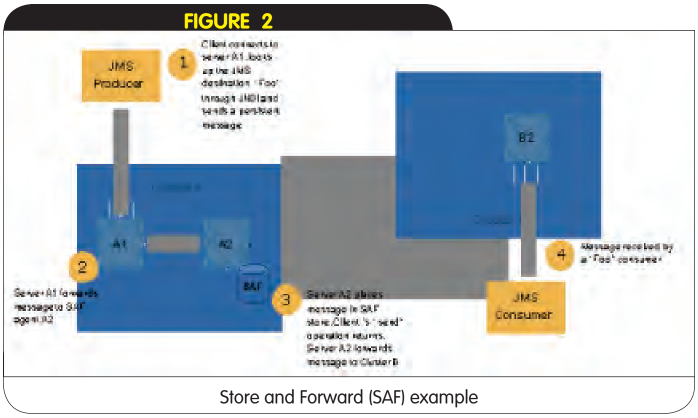
You wish you could have multiple versions on the application server and could switch between each version without interrupt the system.

FIGURE 1


Unit of Order messages on a Queue
MDB 1
MDB 2
MDB 3
Green Group
1
A
2
3
B
C
Blue Group
Green and Blue messages are processed by one consumer at a time.

Sequentially processing messages for multiple groups

FIGURE 2


JMS Producer
1
Client connects to server A1, looks up the JMS destination "Foo" through JNDI and sends a persistent message
A1
A2
SAF
2
Server A1 forwards message to SAF agent A2
3
Server A2 stores message in SAF store. Client 's "send" operation returns. Server A2 forwards message to Cluster B
B2
4
Message received by a "Foo" consumer
JMS Consumer

Store and Forward (SAF) example

Now your dream comes true. Side-by-side application deployment controls the process for deploying new versions of Web-based applications without the need to disrupt service. The new version of an application is deployed alongside existing version – WebLogic will gradually migrate traffic. The older version is un-deployed after all current clients complete their work. The Administrator explicitly un-deploys the older version, or a configured timeout is reached.

Rolling back the new version is simple: just stop the redeployment process if problems are detected in the newer application version.

For new applications, administrators can deploy an application in "administration mode," which makes it inaccessible to non-admin clients, in order to do sanity checks to ensure that the application is working as expected and then open it up to clients.

Here is the list of the features of side-by-side deployment of WebLogic 9:

- Multiple application versions can coexist
- Test versions before opening up to users
- Roll back to previous versions
- Automatic retirement: graceful, timeout, immediate
- Creates version-aware application artifacts/resources
- Reduces hardware, software, maintenance, and support costs

JSR-88 is part of the J2EE 1.4 specification. JSR-88 specifies a standard API for the configuration and deployment of J2EE applications. WebLogic 9 not only implements the JSR-88, but also offers a number of value additions on top of what J2EE specifies.

Reason 7: The WebLogic Diagnostics Framework: Reduced TCO

The mission of WebLogic Diagnostics Framework (WLDF) is to reduce the customer's Total Cost of Ownership through significant diagnostic enhancements. WLDF is laced throughout all of the BEA WebLogic Server 9.0 containers, thus creating a unified framework for the nondisruptive control of data collection that is critical to the health of your enterprise applications. This framework will capture and archive meaningful diagnostic data that may be used to monitor and diagnose problems that arise in a running server. WLDF is a unified framework and a public API so that you can easily plug your applications into the framework to take advantage of the diagnostic capabilities of the server.

WLDF consists of a number of components that work together to collect, archive, and access diagnostic information about the server and the applications it hosts. All of the framework components operate at the server level and are only aware of server scope. All of the components except for the Manager exist entirely within the server process and participate in the standard server life cycle. All artifacts of the framework are configured and stored on a per-server basis. The WLDF Manager provides a configuration and control interface for managing the diagnostic framework. Additionally, the WLDF Image Capture facility provides a model for capturing a diagnostic snapshot of key server state. It provides a minimally invasive means of diagnosing and troubleshooting the server.

Here is the list of the features of WLDF:

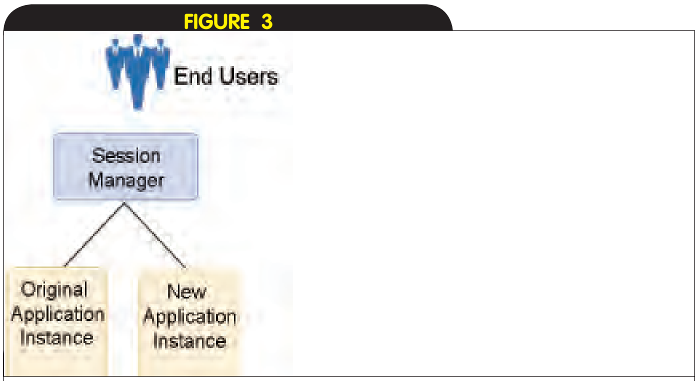
- Introduces a unified diagnostic framework that provides the blueprint for continued BEA enhancement

- Improves the overall visibility of WLS and stack products to alleviate blind spots in monitoring and diagnosis
- Improves control and real-time tunability of the character and quantity of diagnostic data available to the diagnostician
- Introduces additional diagnostic tools to aid in the apprehension and resolution of system faults
- Provides more clearly defined interfaces and tools for instrumentation of customer applications for diagnostics
- Used across WL Platform
- Helps prevent need for Support to issue custom instrumentation patches
- Instrumentation of application and WLS classes
- Watches and Notifications to trigger alerts
- Request dyeing (spans JVMs!): Diagnostic Context for Reconstruction and Correlation of Events
- Fine-grained control – turn the volume up or down
- Data visualization via Console extension: htps://codesamples.projects.dev2dev.bea.com/servlets/Scarab?id=s96
- Programmatic access via JMX

Reason 6: Portal-Based Extensible Administration Console: Extensible Console!

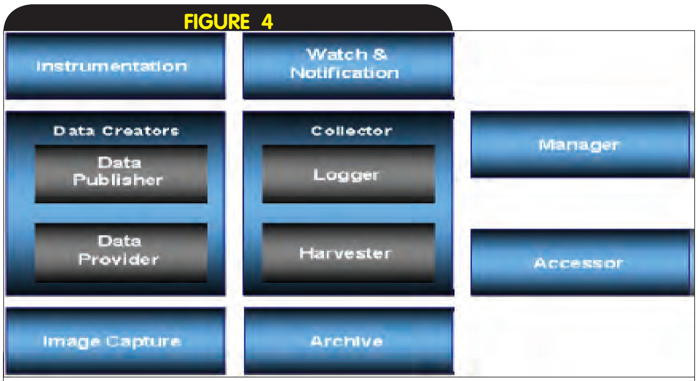
WebLogic 9.0 Administration Console offers a totally redesigned user interface that features standardized and improved look and feel and navigation across all of the subsystems. It is built on the WebLogic Portal Framework. Although using portal slows down the startup process of the admin console, it makes it more open and readily extensible – applications can plug in extensions to the Console if they need to.

FIGURE 3



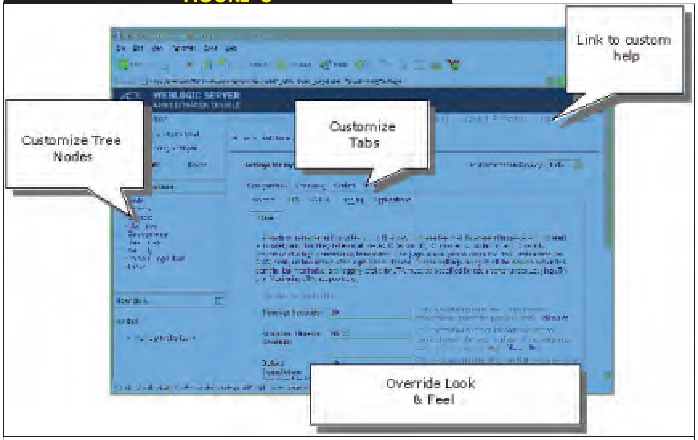
Side-by-side deployment

FIGURE 4



WebLogic Diagnostics Framework overview

FIGURE 5



New look and feel of the administrator console

Among the new features are:

- Improved navigation and user interface design.
- New "change center" to control Domain Configuration Changes. Using the Console, administrators can "batch" changes to their WebLogic Server configuration, thereby making changes predictable and reliable. Upon the administrator's command, all of the configuration changes in a batch are distributed across the domain and applied to all the servers; if they are unacceptable to any of the servers, they are rolled back across the entire domain! They are then kept in a pending state awaiting further action from the administrator.
- New application deployment and configuration tools, including assistants for installing applications, more configuration screens, and new deployment and redeployment controls for production applications. Additional Console updates enable you to assign values more easily to deployment plan variables when deploying an exported application.
- The WebLogic Diagnostic Service, with many new features for configuring, collecting, and viewing diagnostic information in a run-time environment. You access the service through the Console. See Centralized Diagnostic Service with More Visibility and Run-Time Control.
- The Administration Console can now be extended in the same ways in which portal applications generally can be extended. In addition to adding and replacing content, a Console extension can add a node to the navigation tree and change aspects of the Console's appearance, such as colors or branding images.

Through the Administration Console, you are able to perform the following functions:

- Configure, start, and stop WebLogic Server instances
- Configure WebLogic Server clusters
- Configure WebLogic Server services, such as database connectivity (JDBC) and messaging (JMS)
- Configure security parameters, including managing users, groups, and roles
- Configure and deploy your applications
- Monitor server and application performance
- View server and domain log files
- View application deployment descriptors

- Edit selected run-time application deployment descriptor elements

Reason 5: New Zero Downtime Architecture: How about MAN/WAN Clustering?

WebLogic 9.0 has extended the zero downtime concepts to the next level by introducing the MAN/WAN clustering. It offers enhanced HTTP session replication capabilities that enable "Disaster Recovery" across WebLogic Server clusters connected via wide-area (WAN) or metropolitan-area networks (MAN).

HTTP Session Replication over MAN

Applications can configure the backup for the HTTP session replication to be in another WebLogic Server cluster. In the event that the primary WebLogic Server cluster becomes unavailable, HTTP clients can be routed to the secondary WebLogic Server cluster.

This feature assumes the availability of a high-speed interconnect between the two WebLogic Server clusters and relies heavily on correct configuration of the global and local load balancers.

HTTP Session Replication over WAN

This is ideal for disaster recovery centers (DR sites). Applications can configure a second backup for HTTP session replication in another WebLogic Server cluster. WebLogic Server will asynchronously forward HTTP session data to this second backup where it would get persisted in a database. In the event the primary WebLogic Server cluster becomes unavailable, HTTP clients can be routed to the secondary WebLogic Server cluster.

Given that the replication to the second backup is asynchronous, failed-over HTTP clients may encounter stale data. Also, this feature relies heavily on correct configuration of the global and local load balancers.

Reason 4: Whole Server Migration: Ensure Clustered Server Failover

In the Cluster environment, how you use singleton service and also ensure the failover is always a dilemma in the J2EE world. Now the dilemma is over.

BEA WebLogic 9 provides a whole server migration feature. It supports automatic and manual migration of a clustered server instance from one machine to another. A managed server that can be migrated is referred to as a migratable server. This feature is designed for environments with requirements for high availability. Singleton services can be hosted on one server and they can be migrated in the presence of failure. Migratable servers provide for both automatic and manual migration at the server level, rather than at the service level. The server migration capability is useful for:

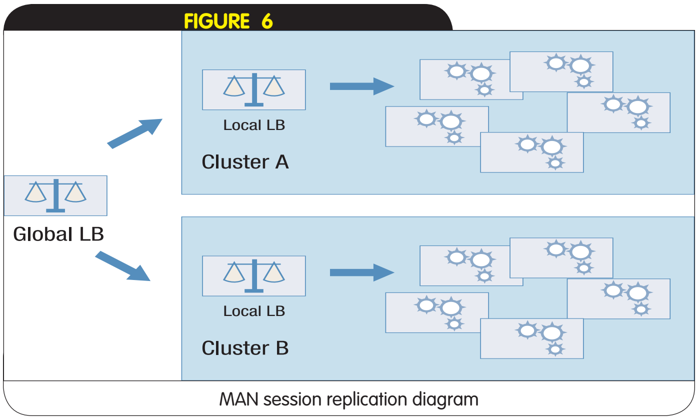
- Ensuring uninterrupted availability of singleton services that must run on only a single server instance at any given time, such as JMS and the JTA transaction recovery system, when the hosting server instance fails. A managed server configured for automatic migration will be automatically migrated to another machine in the even of failure.
- Easing the process of relocating a managed server, and all of the services it hosts, as part of a planned system administration process. An administrator can initiate the migration of a managed server from the Administration Console or command line.
- The server migration process relocates a managed server in its entirety, including IP addresses and hosted applications, to a predefined set of available host machines.

Reason 3: WebLogic Scripting Tool (WLST): Good News for Admin Guys!

WebLogic 9 features an impressive command-line administration tool that is standards based (Jython) and offers powerful capabilities, such as:

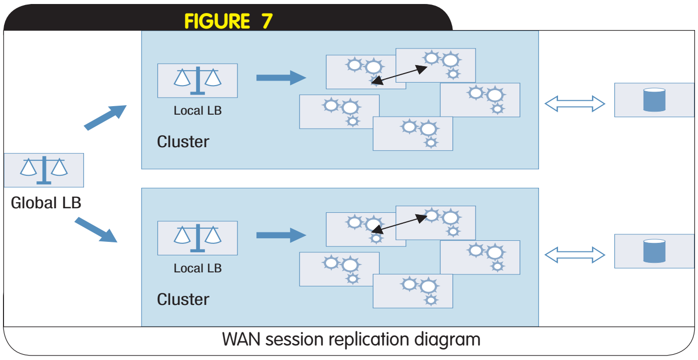
- Navigation and editing of the domain configuration, including user-created and non-WebLogic Server MBeans
- Obtaining run-time information about the domain
- Performing various administrative tasks such as deploying ap-

FIGURE 6



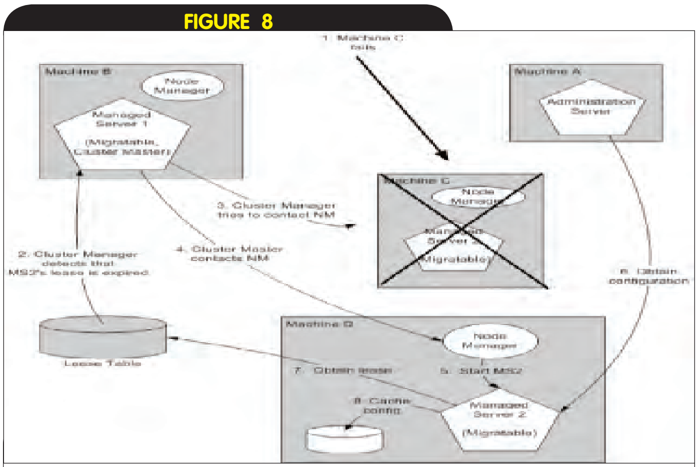
MAN session replication diagram

FIGURE 7



WAN session replication diagram

FIGURE 8



Automatic migration of a failed server

plications, starting/stopping servers via the Node Manager, etc WLST deprecates weblogic.Admin in WebLogic 9, though the latter is still fully supported.

Reason 2: Work Manager and Thread Self-Tuning: Where Is the Execute Queue Concept?

All of the senior J2EE developers have done performance tuning to some extent. In previous versions of WebLogic Server, processing was performed in multiple execute queues. Different classes of work were executed in different queues, based on priority and ordering requirements, and to avoid deadlocks. Users have to control thread usage by altering the number of threads in the execute queues. Now the execute queue concept has been replaced by Work Manger. WebLogic 9.0 implements the Work Manager 1.1 Specification. There is a single thread pool in which all types of work are executed. WebLogic Server prioritizes work based on rules you define and run-time metrics, including the actual time it takes to execute a request and the rate at which requests are entering and leaving the pool, which will provide greater throughput and increased response time. The Work Manager API enables an application to break a single request task into multiple work items, and assign those work items to execute concurrently using multiple Work Managers configured in WebLogic Server. Applications can configure scheduling guidelines (for example, module A should get 70% of CPU time, module B can be "shutdown" if the thread stacks) that WebLogic Server will use in conjunction with data collected on actual run-time performance to schedule CPU resources for the application. Applications no longer have to configure individual thread pools for specific components; instead, they can rely on WebLogic Server to monitor, tune, and allocate these resources.

Among the key self-tuning features in WebLogic Server are:

- Workload management – Administrators can define scheduling policies and constraints at the domain level, application level, and module level.
- Automatic thread count tuning – A thread pool can maximize throughput by automatically changing its size, based on throughput history and queue size
- Thread scheduling functionality – WebLogic Server 9.0 implements the common work manager API, exposing thread scheduling functionality to developers. Applications can also use the Work Manager API to execute work asynchronously and receive notifications on the execution status.

One very nice feature we have to emphasize here is the overload protection. BEA is always trying its best to move ahead of the customer, and overload protection is one of the excellent examples of this effort. WebLogic 9.0 provides two type of protection here.

Type 1: Graceful degradation – overload protection

- Work rejection based on low-memory and queue-capacity thresholds
- Critical systems maintained at the expense of noncritical
- Deny rather than degrade when overloaded

Standard	WebLogic 9	Standard	WebLogic 9
J2EE	1.4	JNDI	1.2
JDKs	5.0 (aka 1.5), 1.4	RMI/IIOP	1.0
WebService	1.1	JMX	1.2
J2EE EJB	2.1, 2.0 and 1.1	JAAS	1.0
J2EE JMS	1.1, 1.0.2b	J2EE CA	1.5
J2EE JDBC	3.0, 2.0	JCE	1.4
Servlet	2.4, 2.3 and 2.2	JAXP	1.2, 1.1
JSP	2.0, 1.2 and 1.1	JAX-RPC	1.1 and 1.0

Type 2: Graceful failure – when all else fails, nuke server

- Option to shutdown/suspend server on critical failures such as deadlocks, OOME
- Well-defined exit codes for scripting
- Ability to specify stuck thread actions

Reason 1: It's All About Performance, Performance, Performance, Stupid!

Of course performance is the always the number one driver to the purchasing decision, migration, and upgrade.

SPECjAppServer2004 is the standard benchmark for evaluating the J2EE application server. BEA WebLogic 9.0 achieves the best SpecjAppServer2004 performance results in the J2EE world. The result of 3,328.80 JOPS, which is reported by Sun instead of BEA, holds new highest record. For the latest SPECjAppServer2004 results, visit the SPEC results page: www.spec.org/jAppServer2004/results/res2005q4/jAppServer2004-20051122-00024.html. You can clearly tell who the top market player is. So what about WebLogic 8.1 – is WebLogic 9.0 faster than WebLogic 8.1 and earlier releases?

BEA creates the Server Performance Index (SPI) to compare the relative WLS from release to release. Similar to the "Dow Jones," the SPI of WLS performance is calculated by looking at a basket of representative performance benchmarks composed of micro and application benchmarks, and then calculating the geometric mean of these benchmarks relative to a baseline. The internal data shows that WLS 9.0 is 17 percent faster than WLS 8.1 SP4 by this measure. Also by leveraging the new WebLogic 9 capability to support new hardware, O/S, and database system, it is possible to gain much higher performance. While the effort for boosting a higher performance is never ending, obviously WebLogic 9 will give you a solid gain at the current stage.

Conclusion

With so many new features in WebLogic 9.0, the reasons to migrate to this new WebLogic Server are so overwhelming, what are you waiting for? Let's do it. Besides, you can always get BEA's professional services and technical support when you need it, so there is no more excuse not to, right?

Resources

- SPEC and SPECjAppServer are trademarks of the Standard Performance Evaluation Corp. For the latest SPECjAppServer2004 results, visit www.spec.org/
- *Side-by-side deployment:* http://e-docs.bea.com/wls/docs90/deployment_api/deploy.html
- *Diagnostic framework:* http://e-docs.bea.com/wls/docs90/ConsoleHelp/taskhelp/diagnostics/CreateDiagnosticModules.html
- *Administrator Console:* http://e-docs.bea.com/wls/docs90/ConsoleHelp/core/index.html
- *Server clustering:* http://e-docs.bea.com/wls/docs90/cluster/Introduction.html
- *Server migration:* http://e-docs.bea.com/wls/docs90/cluster/failover.html#server_migration
- *JMS:* http://e-docs.bea.com/wls/docs90/messaging.html
- *WLST reference for WebLogic 9.0:* http://e-docs.bea.com/wls/docs90/config_scripting/reference.html
- *Java Standard support by WebLogic 9:*

WLDJ ADVERTISER INDEX

ADVERTISER	URL	PHONE	PAGE
Arcturus	www.arcturus.com	866-769-8166	52
Eclipse	http://eclipse.sys-con.com		37
Hp	www.hp.com	800-752-0900	19
Intersperse	www.intersperse.com	800-340-4553	2
Issj	http://issj.sys-con.com/	888-303-5282	39
Jinfonet	www.jinfonet.com/jp12	301-838-5560	23
Motive	www.motive.com/within1	512-531-2527	29
Netiq	www.netiq.com/solutions/web	408-856-3000	51
Parasoft	www.parasoft.com/	888-305-0041	3
Quest Software	www.quest.com/wldj	949-754-8633	7
Reporting Engines	www.reportingengines.com	888-884-8665	11
Sandcherry	www.sandcherry.com	866-383-4500	15
Tangosol	www.tangosol.com	617-623-5782	13
Wily Technology	www.wilytech.com	888-get-wily	5
Windward	www.windwardreports.com	303-499-2544	27
Wldj	http://weblogic.sys-con.com/		35

This index is provided as an additional service to our readers. The publisher does not assume any liability for errors or omissions

keep your J2EE problems...at bay. Get freedom for more in life

Eliminate endless hours of resolving J2ee problems.
Let AutoPilot ™ from Arcturus Technologies, do that for you.

It's an affordable new product that eliminates worries about WebLogic upkeep and maintenance. AutoPilot™ is so powerful and efficient that not only does it monitor, analyze and optimize your WebLogic for maximum performance, it also advises you about how to cure problems and avoid costly failures. It does this all automatically, giving you the freedom to put back what you have been missing in life

- Automatically optimizes WebLogic server, saving you time and effort.
- Detects problems automatically.
- Gives advice from a dynamic knowledge base.
- Provides WebLogic system state for analysis in the event of a catastrophic failure.
- Optimizes existing J2EE resources.
- Increases the life, effectiveness, and quality of existing resources by reducing or eliminating problem areas.
- Improves resource communication and response time.
- Enhances server capacity to handle more loads and transactions.
- Provides advanced system monitoring.
- Furnishes customizable reports.
- Generates email alerts.

www.arcturustech.com
866/769-8166
sales@arcturustech.com

AUTOPILOT™
AUTO OPTIMIZER