IBM WebSphere Portal: Performance testing and analysis

Alex Lang

May 01, 2009

(First published July 01, 2008)

This article provides advice and methods for finding and resolving common performance problems with IBM® WebSphere® Portal. The WebSphere Portal SEAL team engages with high-profile customers experiencing significant problems with their WebSphere Portal deployments. In roughly 50 percent of these engagements, the major complaint is performance related. The task usually becomes one of finding and resolving these performance issues, often after the system has already been put into production.

Finding and fixing performance problems in a production environment is challenging on a number of levels. Optimally, most bottlenecks in the system should be found and fixed before the system is allowed into production. This article explains a tested process that can ensure that, with high probability, most of the significant performance issues are found and addressed before you promote a system to production.

Performance testing and analysis have three main objectives:

1. Determining the load level at which a system under test fails
2. Finding bottlenecks in a system that throttle throughput and removing them as soon as possible and practical
3. Capacity planning: For example, predicting the amount of horsepower needed to sustain defined users loads within agreed-upon service level agreements (SLAs)

The system is defined as the complete end-to-end set of components required to deliver the requested Web page to the requesting user’s browser. The most visible and often the most troublesome components tend to be WebSphere Portal itself, the WebSphere Portal database, the Lightweight Directory Access Protocol (LDAP), and the back-end systems (databases, application servers, and so forth) that supply content to the portlets.

The back-end systems tend to present the most risk in WebSphere Portal deployments because they are frequently maintained by separate organizations. This separation dilutes the communication channel between the WebSphere Portal deployment team and the back-end teams with respect to performance objectives.
The methodology presented in this article is a holistic approach that meets all three of the objectives when it is executed successfully.

**The environment**

To meet the performance test objectives outlined previously, the performance test environment needs to be either the production environment itself or a mirror of production. That mirror has, as is practically possible, the same hardware, the same topology, and the same back-end systems. If any piece of this complex test topology is different from its production counterpart, you must extrapolate the results in the test environment to predict its effect in the production environment. These extrapolations generally require detailed implementation knowledge of the portal and the deployed applications, which generally are not available to the testing organization. By making the test environment equivalent to the production environment, your confidence in the test results as they relate to what actually happens in production becomes acceptable.

An important goal of the test environment is the repeatability of results. As slight changes are made in the system, repeatability ensures that you can accurately measure the effect of these changes. For that reason, it is optimal to have the system on an isolated network during the performance testing. Running the performance test on the production network introduces variability (for example, user traffic) that can skew such metrics as page render response time. There is also a more pragmatic reason to isolate the performance test network. Putting WebSphere Portal under stress likely puts the corporate network under stress. This stress is often problematic during normal business hours.

If placing the performance test on an isolated network is not feasible, you should at least try to ensure that the components of the test are all collocated on the same subnet of a network router. WebSphere Portal best practice recommends using a gigabit Ethernet connection between the portal and its database. Optimally, this connection extends to the LDAP servers, the Web servers, and important back-end services. It is crucial that any load generators also be on a LAN segment local to the Web server and/or the portal itself.

A common customer concern involves the load generators being on the same local LAN segment as the portal itself. In this case, this test does not get a true picture of the performance of the system as it excludes the network from the data center to the users. The answer to this concern is often difficult for customers to accept. The process described here is for tuning and resolving issues with the portal and its surrounding components. Trying to tune the network between the users (or the load generators) and the portals makes the analysis and problem resolution needlessly complex. We therefore remove it from the test. There are far better tools and processes for network tuning than the processes used here.

**Portal infrastructure baseline**

In contrast to the mirrored production environment, it is strongly advisable to also conduct an incremental set of baseline tests that exercise the infrastructure. At that point, subsequent tests should then gradually augment the portal with customer-written code. The test plan should
thus move from a simple topology to the final production topology to make it easier to isolate problematic components.

The first test is the complete WebSphere Portal infrastructure using an out-of-the-box portal. Transfer the database, and enable security. Make sure that all front-end Web servers, firewalls and load balancers are in place. Security managers (for example, Computer Associates SiteMinder or IBM Tivoli® WebSeal) should also be in place and correctly configured. Create a simple home page with a couple of portlets that do not access any back-end systems (for example, the World Clock portlet). Create a simple load testing script that accesses the unauthenticated home page and then logs in (authenticatees) and idles without logging out. From this point, you want to add simulated users (Vusers) until the system is saturated. Using the bottleneck analysis techniques described below, find and fix any bottlenecks in the infrastructure. Note the performance baseline of this system.

Now, add to the system any customized themes and skins, and repeat the previous test. Find and fix any important bottlenecks in the revised system. Finally, as described below, add the actual portlets to be used on the home page and perform bottleneck analysis.

This baseline environment can be very effective in finding bottlenecks in the infrastructure that are independent of the application. Further, it can provide a reference when analyzing the extent to which the applications place additional load above and beyond the basic WebSphere Portal infrastructure.

Your strategy is to conduct the same tests listed below for bottleneck analysis in this baseline environment, optimize the environment, and then perform bottleneck analysis with the actual applications.

**Application of the Portal Tuning Guide recommendations**

Apply the recommendations outlined in the [WebSphere Portal Tuning Guide](#) to all systems before you embark on any performance testing. The guide provides a good starting point because it fixes known performance inhibitors in a default WebSphere Portal installation. Although bottleneck analysis would likely find the same problems, it is better to remove them from the beginning.

Note that the tuning guide is a starting point for your performance testing and not the final set of configuration changes needed to optimize your Portal. Your application(s) along with your unique themes and skins can greatly affect the correct setting needed to optimize performance.

**Load generation**

A proper performance test also requires the use of a load generator that produces simulated user requests for Web pages. It is important that this tool produce such metrics as response time and page views per second. These metrics allow you to determine when the system fails its Service Level Agreement (SLA) contract or is saturated to the point that injecting more page requests per unit time does not result in higher page production. Saturation is discussed later in this article. The generator's ability to aggregate data such as CPU utilization on the portal and HTTP servers as well as mod_status data from the HTTP server aids in problem determination.
A number of load generators are commonly used to create Web traffic (also known as drive load) in the test system. Some of the more commonly used tools include Mercury LoadRunner, Borland SilkPerformer, and IBM Rational® Performance Tester.

It is important that the load generator have sufficient virtual users (vUsers) to drive the system to saturation. Note that virtual users do not map directly to actual users. A virtual user represents an active channel on the load generator. A virtual user may simulate multiple actual users; however, only one actual user can be active for each virtual user.

It is also important, especially in a WebSphere Portal context, that if the system requires authenticated access to the WebSphere Portal applications under test, that sufficient unique test user IDs exist in the LDAP directory and that scripts ensure that only a reasonable number of duplicated logins occur during the test. (A reasonable number in this context accounts for the fact that some users might have a couple of instances of the browser open, each with the same WebSphere Portal login ID.) WebSphere Portal has a large caching infrastructure for portal artifacts. These artifacts are generally cached on a per-user basis. If the load simulation uses the same user ID for all tests, performance appears artificially high because the artifacts do not need to be loaded from the LDAP directory and the database.

The general methodology

The sections below describe the iterative process that is used to do the actual analysis of the system. The sections that precede "The process" define concepts that are important to understand during the execution of that process.

User scenarios

To tune the WebSphere Portal system to handle large numbers of users and to accurately predict its ability to handle specific numbers of users correctly, it is important to determine the most probable scenarios for users of the system. The test must then accurately simulate those user scenarios using the load generator. One effective way to do this step is to list the most likely use cases. Write a script for each of these use cases or as many as are practical. Now, assign a probability of likelihood that a percentage of the whole user population will execute that scenario. As the test is run, assign use cases to Vusers in the same proportion as the expected general population. As the number of Vusers is ramped up (discussed later), try to maintain this proportion.

NOTE: "Vuser" is a LoadRunner term. It represents one active channel over which requests are made and returned.

Think time

Think time is the average amount of time that a normal user pauses during individual mouse clicks or key presses during the course of using WebSphere Portal. In the load generation tools, this time is usually programmable, yielding a random time within a predefined range.

As think time is reduced, the number of requests per second increases, which in turn increases the load on the system. Reducing think time generally increases the average response time for
WebSphere Portal login and page-to-page navigation. Therefore, accurately estimating real user think time is important for producing an accurate model of the system in production, particularly for capacity planning.

In most use cases, a think time of 10 seconds plus or minus 50 percent is reasonable for a portal having experienced users. A figure closer to 30 seconds is more reasonable for a portal with inexperienced users.

**Cookies and sessions**

Generally, most real users log into the portal and execute the task that needs to be done; however, they rarely log out by using the logout button. Rather, they let the browser sit idle until their session times out. Typically, a lot of sessions in memory are waiting for cleanup pending the WebSphere Application Server session timeout. This behavior increases the Java Virtual Machine (JVM) heap working set, which increases the probability of heap exhaustion in the JVM. Heap exhaustion can be both a performance bottleneck and a cause for a JVM failure.

Effective simulations must model this behavior of users who do not explicitly log out. As each individual simulation executes a particular use case, it should end the use case by going idle as opposed to logging out. As the script cycles back around to log in a new user on this particular Vuser, the cookies for old session (typically JSESSIONID) and Lightweight Third-Party Authentication (LTPA) along with any application-specific cookies need to be cleaned up appropriately before logging in the next user using that script. This model also implies that sufficient test IDs need to exist so that a test ID can sit idle for the length of the WebSphere Application Server session timeout without risk of being reused until the previous session times out.

**Metrics**

It is important that the scripts be instrumented for metrics. The most important metrics is page views per second. Also important are request response times. As login is very expensive in Portal, login response time, along with page-to-page response times, needs to be instrumented. Most of the load generators already provide aggregate Page View per second (PV/s) metrics.

At the conclusion of each test, a graph of Vusers ramp rate versus the three metrics is required for doing analysis.

In addition to the metrics gathered by the load generation tool, a system monitoring tool such as IBM Tivoli Composite Application Manager (ITCAM) for WebSphere or the Computer Associates Wily IntroScope product should be employed. These tools run on the WebSphere Portal instance and instrument the JVM directly. They are useful in both detection and resolution of system bottlenecks.

**vUsers as opposed to Think Time**
A common misconception is that to accurately simulate a large population that generates requests at a certain rate, a smaller number of users that generate requests with a smaller think time will suffice.

It's important to note that the effects of running with a small number of users and a low think time results in unrealistically high cache hit rates. It also means that too few sessions are created. Because session size is often a serious problem for many portlet applications, this approach gives an unrealistically good view of the system performance and leads to surprises in production.

Another poor practice is running a small set of vUsers with no think time.

**Repeatability principle**

In a large population, it is easy to assume that most user actions appear to be random as users navigate through the portal. Experienced users, though, typically use the same patterns over and over. Furthermore, from a test engineering perspective, the user scenarios need to be reasonably static so that system changes can be effectively measured from run to run.

Therefore, the definition of the repeatability principle is that for all runs of a particular scenario, the metrics (average response time, PV/s, saturation point, and so on) produced by the runs all converge to the same results if the runs are sufficiently long. Note that with more variation (that is, unique scenarios) in the test scripts, longer times are required to converge, on average.

The simulation scripts written for the performance tests should adhere to the repeatability principle.

**Driving to saturation**

Saturation is defined as the number of active Vusers at which point adding more Vusers does not result in an increase in the number of PV/s. Note that this saturation point is for a given simulation; each different simulation likely has a different saturation point. The saturation point varies depending on the usage pattern.

To effectively drive a system to saturation, add Vusers a few at a time, let the system stabilize, observe whether PV/s increase, and add more Vusers as possible. ("Stabilize," in this context, means that the response times are steady within a window of several minutes.) On LoadRunner, if you plot Vusers against throughput (PV/s), the PV/s initially rises linearly with the number of Vusers, then reaches a maximum and actually decreases slightly from that point. The saturation point is the number of Vusers at which the PV/s is at maximum.

**Bottleneck analysis**

The goal of bottleneck analysis is to remove impediments which inhibit driving the system to a higher load. The metric defined for higher load is a higher number of PV/s at saturation. Therefore, bottleneck analysis removes impediments to improve the saturation point.

Bottlenecks in a WebSphere Portal environment under load are generally the result of two issues. The first is contention for shared resources. This contention can be the result of synchronized Java classes, methods, or data structures, contention for serial resources (for example SystemOut.log).
The second issue is excessive response times in back-end databases, remote IBM Web Content Management (WCM) systems, or Web servers. You must also be mindful of bottlenecks such as the network itself. Components such as routers and firewalls can impose congestion control or can be poorly tuned.

As load increases, contention for these resources increases, making contention locks easier to detect and correct. This detail is why effective load testing is a requirement for bottleneck analysis.

A common mistake is to focus only on page response times. Many performance testers prefer to optimize render response times because this delay is the most obvious user requirement. This type of performance analysis requires path length reduction in the customer portlet applications. Response time optimization is generally more appropriately done in a non-loaded system and with tooling specific to the task (for example, JProbe).

**The process**

The process of performing bottleneck analysis is straightforward. For a particular performance analysis (for example, LoadRunner) simulation, follow these steps:

1. Ramp a single WebSphere Portal JVM to saturation.
2. Determine the bottlenecks that exists at saturation.
3. Resolve the bottlenecks.
4. Unless satisfied with system capacity, go to step 1 and find the next bottleneck.

Note that this process is iterative. The key concept is that you fix one bottleneck to find the next bottleneck. You stop the process either when you are satisfied with the system performance or when the cost to resolve the next bottleneck becomes unjustifiable. Most customers generally do not allocate enough time for this work because they fail to realize the iterative nature of this process.

A single JVM is used for this process because detection of the bottleneck is much simpler. Finding and resolving cross-JVM contention can be quite complex. After a single JVM has been tuned as much as desired, you move on to the capacity planning analysis for multiple nodes as described later in this article.

**Note on ramp rates**

A common question in performance testing is the rate at which Vusers should be ramped into the system.

Do not ramp in several hundred users as quickly as possible until the system collapses. This approach is not representative of reality, and it does not provide repeatable results.

You should model reality. Predict or measure the actual highest ramp rate that you would expect your portal to endure. This rate might typically occur during the hours that your users most often log into your portal, such as first thing in the morning when they arrive at the office. We recommend that you ramp a small fixed number (for example, two Vusers per minute) for a set period of time (for example, five minutes). Then wait for a time to let the system stabilize (for
example, five minutes) at which time you then loop back and add another batch of Vusers in the same fashion.

This technique gives the portal time to fill the various caches in an orderly fashion and provides for the ability to more accurately detect saturation points.

**Priming the portal**

After a portal restart, a short script should be executed prior to the main test to preload certain caches (for example, WebSphere Portal access control and the anonymous page cache before the real test starts. Failure to do so can skew the initial response times inordinately.

**Analysis techniques**

After you have a portal at saturation, you can determine the cause of a bottleneck in the system by taking a Java thread dump (using a kill -3 command) against the portal Java process under test. A thread dump shows the state of all threads in the JVM. The general procedure is to look for threads that are all blocked by the same condition or are all waiting in the same method of the same class.

In general, search for threads that are blocked or in a wait state. By ascertaining why certain classes statistically show up blocked, you can then proceed to remove that reason and thus remove the bottleneck. The next section discusses some common bottleneck problems. Going into all the problems you could encounter is really the art of WebSphere Portal bottleneck analysis and takes time and experience to master.

If the bottleneck is not the WebSphere Portal JVM itself, detection and resolution techniques are varied and outside the scope of this article.

**Common problems**

This section lists common problems that many customers have seen during their performance testing.

**JVM heap utilization**

The topic of JVM tuning, especially garbage collection, is long and very application dependent. The first rule when initiating performance analysis is to ensure that you have applied the initial JVM tuning recommendations as outlined in the Portal Tuning Guide.

In addition, there are several other recommendations that you should adhere to prior to significant load testing.

1. Enable verboseGC. Leave it enabled, even during production. The amount of log data is not large; however, it is invaluable in terms of the visibility that this log brings to heap utilization problems.
2. If the size of the native_stderr.log file becomes a concern due to verboseGC logging, consider setting the following generic JVM parameter to force verboseGC log rolling:
Navigate to Servers - Application Servers - WebSphere_Portal - Java and process management - Process definition - Java Virtual Machine - Custom properties. Then create the environment variable ALLOCATION_THRESHOLD with a value of 1000000. This variable causes any Java object allocations greater than 1M to be recorded in the native_stderr.log for analysis. Allocations this large are troublesome in a WebSphere Content Management or WebSphere Portal JVM unless additional tuning parameters are used. First and foremost is -Xloratio, which reserves a larger area for large objects in the heap than the default. If you are experiencing Out of Memory errors and they coincide with large object allocations when there seems to be plenty of heap available in the verboseGC log, heap fragmentation due to large object allocations is the likely culprit. Setting -Xloratio0.1 will likely help.

4. If the verboseGC log indicates a large number of mark stack overflows (MSOs), performance under load likely suffers. The use of -Xgcthreads to override the default provides additional mark stack space, which provides relief from MSOs.

Logging

Logging using direct writes to SystemOut.log or using a logging class such as log4j causes serialization between running threads and significantly degrades portal performance. In production portal systems, log only what is absolutely needed. When using log4j, log only errors; do not log warnings or informational messages. If logging is required for audit purposes, consider using a portal service or a different service running in a separate JVM.

Turn off all logging and remove all debug code that writes to files before doing performance testing.

Java class and variable synchronization

Use of method-level synchronization blocks where a method is in a monitor wait (MW) state with one method holding a lock can be problematic. In this case, you have Java code that is synchronized and is causing serialization in the system.

Use of synchronized class variables or synchronized HashMaps can also cause this problem.

In both cases (method or variable synchronization), the problem can be exacerbated by arbitrarily increasing the number of WebSphere Application Server transport threads in which the portal runs. By increasing the number of threads, you increase the probability of hitting portal code that is synchronized in this fashion, which ultimately serializes all the threads.

Database contention

If the thread dump indicates numerous threads waiting in Java Database Connectivity (JDBC) classes in Socket.read() methods, then there are likely response time issues in the database itself.

At initial Database Transfer time, Portal sets up the databases with indexes that should be good initial starting points. It is imperative, though, that an excellent DBA monitors the database to
ensure efficient operations. As a result of this monitoring, the DBA might need to effect changes on
the DB to remove bottlenecks in the system.

Some common problems and resolutions that have been seen include the following:

- Queries taking excessive time due to table scans
- Insufficient processor and memory resources on the DB server itself
- Insufficient allowed connections as opposed to the configured JDBC pool sizes on Portal and
  Lotus Web Content Management

DBAs should, especially when thread dumps indicate excessive JDBC wait times, take snapshots
for long queries. Generally, Portal and Lotus Web Content Management queries all execute in
subseconds, if not in milliseconds. Look at the execution plans for long-running queries, and see if
additional indexes might be required to improve response times on problematic queries.

When threads are waiting on JDBC pool resources in WebSphere Application Server, you see
the threads in a condition wait (CW) state in the WebSphere Application Server connection pool
(J2C) classes. In this case, you might need to increase the pool size for this data source. Note that
in doing so, you might need to increase the number of connections that the database server can
handle concurrently.

**LDAP responsiveness**

If several threads are in the Socket.read() method of the Java Naming and Directory Interface
(JNDI) classes, they are likely waiting on results from the LDAP directory.

**Excessive session sizes**

If customer-written portlets are storing too much data in the session, that condition invariably leads
to memory and performance issues.

**Exceptions being thrown**

Even though this problem might seem obvious, in many customer situations performance analysis
and bottleneck reduction are attempted in systems that are repeatedly throwing exceptions in the
logs. When the JVM is handling unchecked exceptions, it slows the JVM down and causes serial I/
O (printing) to the SystemOut.log print stream, which serializes the WebSphere Application Server
transport threads.

A more general issue involves trying to characterize and tune a system that is inherently flawed.
All results that are generated in such an environment must be labeled as non-repeatable and
subject to change (potentially in a significant way) as the flaws are eliminated.

Finally, it should be your policy that the WebSphere Portal system is not allowed to enter a high-
load production environment with any errors in the logs.

**Dynacache concerns**

- **DRS replication modes**
WebSphere Portal requires that the WebSphere Application Server Dynamic Cache Service be enabled. The dynamic cache (or `dynacache` as it is commonly known) is a data structure that is used to provide caching of data from back-end services (for example, database results) in WebSphere Portal. Dynacaches can ensure cache synchronization across a cluster of WebSphere Portal members. For proper operation in a cluster, WebSphere Portal requires that cache replication be enabled. The default mode of replication, PUSH, can cause performance problems, though, in the WebSphere Portal environment. WebSphere Portal V6.0.1.5 and V6.1 change the default for all Portal and Lotus Web Content Management dynacaches to be NOT SHARED instead of PUSH.

The use of NOT SHARED is strongly recommended for the vast majority of WebSphere Portal configurations. Three actions are needed to ensure that each Portal cluster member is fully optimized for WebSphere Portal Version 6.0.1.4 and earlier. The first is to set the replication mode to NOT SHARED using the WebSphere Application Server console for each cluster member. The second is to install Portal PK64925. The third is to install WMM PK62457 and add the parameter `cachesSharingPolicy` with a value of NOT_SHARED to the LDAP section of the wmm.xml files on each node. You can check out further details [here](#).

WebSphere Content Manager’s (WCM) dynacaches also should be set to NOT SHARED. To complete this task, in the Deployment Manager console, navigate to Resources -> Cache Instances -> Object Cache Instances and change each of the individual cache instances to a mode of NOT SHARED. As of the time of this writing, there are 11 instances for WebSphere Content Manager.

Finally, there are WebSphere Application Server changes that can further, although marginally, reduce the amount of network traffic between cluster members due to replication events. For each cluster member (either WebSphere Content Management or WebSphere Portal), navigate to Servers - Application Servers - WebSphere_Portal - Java and process management - Process definition - Java Virtual Machine - Custom properties. Then click New to define the following properties:

- `com.ibm.ws.cache.CacheConfig.filterLRUInvalidation=true`
- `com.ibm.ws.cache.CacheConfig.filterTimeOutInvalidation=true`
- `com.ibm.ws.cache.CacheConfig.cacheInvalidateEntryWindow=2`
- `com.ibm.ws.cache.CacheConfig.cacheEntryWindow=2`

**Dynacache eviction concerns**

Since WebSphere Portal version 5.1.0.2, the size of the WebSphere Portal dynacaches has been increased to a default that is appropriate for most customers’ WebSphere Portal applications. There are situations, though, in which these defaults are inadequate and can cause significant performance problems. For example, if a portal has a large number of derived pages with a common parent in WebSphere Portal V5.1.0.x, the portal access control (PAC) caches can
be small enough to cause cache thrashing. Similarly, if the portal objectID cache is too small, thrashing occurs.

Customers need to install and use the advanced dynacache monitor and monitor all the dynacaches. If one or more of the caches seem to have large amounts of least recently used (LRU) evictions, the size of that cache might need to be increased. The sizes of the WebSphere Portal caches are mostly located in the WebSphere Application Server Resource Environment Provide named WP_CacheManagerService. The size of Lotus Web Content Management dynacaches is controlled from the Deployment Manager console in the Object Caches section.

**Operating system concerns**

Efficient operation of Portal and Lotus Web Content Management depends on adequate tuning of the host operating system and availability of sufficient resources. While tools like *techline* can size the host environment required for enterprise deployments of Portal and Lotus Web Content Management, there are problems that can arise even on adequately sized hosts.

- Under no circumstances should memory paging occur on an operating system hosting Portal or Lotus Web Content Management. If it is, actions must be taken to alleviate this situation. Performance will immediately and dramatically degrade in the presence of paging.
- Enable large page support on AIX and set the JVM property `Xlp` to dramatically improve memory utilization.
- On AIX, consider setting the memory management option `lru_file_repage` to 0 to ensure that computational memory is prioritized over file I/O buffers. This setting ensures that in situations where physical memory becomes limited, AIX will not swap out the Java processes in favor of file I/O buffers.

**Customer portlets**

Some common problems noted from past customer engagements include the following:

1. Use of synchronized class variables.
2. Excessive database calls. Consider using DB caching layers or dynacache to reduce the load on application databases or back-end services.
3. Unsynchronized use of HashMaps. There are timing scenarios in which these classes get into infinite loops if separate threads hit the same HashMap without being synchronized.

**Capacity planning**

The goal of capacity planning is to estimate the total number of WebSphere Portal JVMs required that satisfy a certain user population within predetermined SLA metrics prior to entering production.

Typical metrics include:

1. Portal login response time (typically around four seconds)
2. Page-to-page response times after being already logged in (typically around two seconds)

**The process**

The process for running the load test looks very much like the one for running the test for bottleneck analysis except that there is now a second criterion for stopping the test. One criterion is saturation, as previously defined. The second criterion is failure of any of the SLA metrics.

If the test reaches saturation before any of the SLA metrics are exceeded and if it has already been determined that there are no bottlenecks that can or will be excised, then you can immediately calculate the number of nodes required.

If the SLA metrics are exceeded before reaching saturation, then you must analyze the failure to determine the next course of action. If you determine that you do not need to resolve the response time issues, then proceed directly to calculating the number of nodes, as discussed in the next section of this article.

**Extrapolating results**

In general, if a single WebSphere Portal node can sustain $n$ users within given SLA metrics, then 2 nodes can sustain $1.95 \times n$ users. The accepted horizontal scaling factor for a portal is .95. Thus, if a single WebSphere Portal node can sustain $n$ users within given SLA metrics, then $m$ nodes can sustain:

$$n (1 + .95 + .95^2 + .95^3 + \cdots + .95^m)$$

Thus, the horizontal scaling factor is slightly less than linear.

This scaling factor assumes that the database capacity does not bottleneck the system. In fact, this scaling factor is primarily a metric of the degeneration of the WebSphere Portal database for logging in users.

Vertical cloning (scaling) is somewhat different. Vertical cloning is indicated when a single JVM saturates a node at a processor utilization around 80 percent or less. Note that in most cases, bottleneck analysis usually provides relief. In the absence of Java heap issues, a single JVM can usually be tuned to saturate a node at 85 to 90 percent processor utilization.

Vertical scaling is discussed more fully later in this article.

**Testing with the full cluster**

If sufficient load generation capacity exists (including test IDs), it is wise to do a final series of tests in which the whole user community is simulated against the full cluster to ensure viability of the entire system.

**Failover testing**

If there is a system requirement for full performance during a failover, this scenario should also be scripted and tested.
Before running this scenario, review the plugin-cfg.xml file at the HTTP server to ensure that the cluster definitions are correct. Consider adding the parameter ServerIOTimeOut to the cluster members. This parameter augments the ConnectIOTimeout parameter. ConnectIOTimeout is the amount of time before a cluster member is marked as down in the event that the remote server fails to open a socket connection upon request. The parameter is normally present in the plugin-cfg.xml file and defaults to 0, which means that it relies on the operating system to return timeout status to the plug-in instead of the plug-in explicitly timing the connection itself.

The parameter ServerIOTimeout is, by default, not included in plugin-cfg.xml. This parameter sets a time-out on the actual HTTP requests. If the portal does not answer in the allotted time, the server is marked down. This step is useful because there are certain classes of failures whereby the WebSphere Portal cluster member accepts a socket open request, but the JVM has hung and will not respond to HTTP requests. Without ServerIOTimeout, the plug-in does not mark the cluster member as down; however, it is not able to handle requests. This situation results in requests being routed to a hung server.

During this test, start with the cluster fully operational. Enable Vusers in your simulation to the maximum number that your SLA mandates. Then, stop one or more cluster members. You can do this step gracefully by stopping the cluster members from the deployment manager or by simulating a network failure by removing the Ethernet cable from a cluster node. Many other failure modes might be worth investigating (for example, database failures, Web service failures, and so on). After the simulated cluster member outage, ensure that the surviving cluster members handle the remaining load according to your system requirements. Then, restart the offline cluster members to ensure that the load returns to a balanced state over time.

**Ongoing capacity planning**

If a system is already in production and is meeting its current SLA goals, you also want to plan for future growth in the number of users of the system. Assuming that the applications on the WebSphere Portal do not significantly change, you can derive the necessary measurements and calculations from a running production system. You need proper tooling, though, to take the measurements.

In short, if n JVM can support x users, then each JVM can support \((x/n)^{(1/95)}\) users. Using the formula explained previously, you can easily plan for future growth.

**Vertical clustering considerations**

A common technique for improving performance is to vertically clone the WebSphere Portal JVM on the same physical system. Engineers initially assume that if one JVM is good, two must be better.

The ultimate goal of vertical cloning is to increase the net aggregate throughput in transactions per second of the sum of the cluster members (clones) on a single node. This goal is usually possible only if, when running under the load, a single, well-tuned cluster member does not consume most of the CPU available in that node. In fact, in a well-tuned WebSphere Portal, vertical cloning always carries a cost. Vertical cloning is indicated when the benefits outweigh the costs.
WebSphere Application Server clustering comes in two flavors. The first is the horizontal type. In this arrangement, a functionally equivalent duplicate of an application server is created on another node. This duplication is done with a WebSphere component known as the deployment manager. The resulting set of equivalent nodes is known as a cluster. The result is that a front-end HTTP server can forward a request from a client to either of the cluster members (clones), and the result is identical.

Similarly, you can also create cluster members vertically, which means that multiple JVMs are created on the same node. Each cluster member can serve the same content just as in the horizontal cluster member case.

In the WebSphere Portal case, each cluster member shares the one (and only one) WebSphere Portal database. This statement changes slightly in WebSphere Portal V6, but it is true for V5.x. Therefore, as the number of cluster members increases, the WebSphere Portal database has a higher likelihood of becoming a bottleneck due to the dilution of its capacity.

Costs of vertical clustering

When additional cluster members are active on the same physical node, costs are associated with it. First, there is process context switching. The operating system must now manage additional processes (JVMs).

Second, there is more contention for processor resources. Generally, vertically clustering is always a bad choice if the number of active cluster members exceeds the number of processors in the node less one. You should never have three cluster members on a three-processor node, for example. Two cluster members on a three-processor node might be acceptable under certain conditions.

Indications for vertical clustering

This section describes some of the situations in which vertical cluster members provide value.

Reliability

Apart from performance concerns, having additional cluster members might make sense strictly for reliability reasons. If a WebSphere Portal installation is on a single node, then in the event of a software failure that crashes one JVM (without crashing the operating system), you can mitigate the effect of the crash by adding vertical cluster members. The assumption is that most software failures are localized to a single JVM and do not affect the others on the same node. Therefore, the cluster continues serving requests while the failing JVM is restarted.

Memory utilization

In a 32-bit operating system, process address spaces are limited to 4 gigabytes of memory. Most operating systems split this space as 2 gigabytes of user space and 2 gigabytes of kernel space. There are exceptions whereby the user space can be increased to ~3 gigabytes and the kernel reduced to 1 gigabyte (Solaris, AIX®, and Microsoft® Windows® 2003 Enterprise, for example).
If the address space available to the JVM is 2 gigabytes, then the JVM can allocate approximately a 1.5-gigabyte heap space.

There are cases when the combination of the WebSphere Portal base memory working set, along with the total memory required for all the portlets running during stress, could approach and exhaust the 1.5-gigabyte heap. When this happens, and if there is still a significant amount of processor resource available (20 to 30 percent or more), then vertical cloning could increase the total throughput of the box by effectively creating 3 gigabytes of JVM heap and dividing the workload evenly between the two 1.5-gigabyte heap JVMs.

**Java synchronized methods and class variables**

If your WebSphere Portal application (and the portal itself) uses enough synchronized methods or class variables, you can, under load, end up with a high and frequent number of blocked threads in the application server. You can identify this situation by taking thread dumps under load and noticing that there are lots of Web container threads sitting in MW state waiting for these synchronized artifacts.

In this case, reducing the maximum number of Web container threads on a per-cluster-member basis reduces these stalls. If, after that change, the processor is not consumed as described previously, then vertical cloning can increase the aggregate throughput for the whole node.

**Conclusion**

With proper testing before putting WebSphere Portal into production, you can remove many common performance problems, thereby providing for a much smoother user experience. This article provided a framework for building the test plan and execution processes needed to ensure that performance is acceptable and predictable as the system is deployed to production.
Related topics

- Learn more about IBM WebSphere Extended Cache Monitor.
- Refer to the IBM WebSphere Portal Information Center tips on performance tuning.
- Refer to the IBM WebSphere Portal Performance and Tuning Guide.

© Copyright IBM Corporation 2008, 2009
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)