Rutgers performance tuning experience

Rutgers performance issues

During Fall 2007, Rutgers migrated from WebCT to Sakai. This and other events greatly increased the load on our server. During the Fall and Spring 2008, we saw a number of serious performance issues, which it seems worth documenting. This issues continued into the Spring of 2010. As of Fall, 2010, I think we have resolved them. While this document reflects our current experience with Sakai 2.7 and Mysql 5.1, most of it applies to earlier versions.

Long garbage collection times

We have regularly seen minor garbage collections take on the order of a minute. Since the appeared to be a garbage collection issue, we spent lots of time adjusting JVM parameters. This never fixed it. We now believe that some of the problem was caused by bugs in dbcp and pool, i.e. in the code that manages connection pools to the database.

It's not entirely clear how this resulted in long garbage collections, but we're pretty sure it did. See below for our recommendations on database tuning. Do that before starting to tune your JVM

We tried c3po and it didn't really fix the problem.

JVM setup

We currently use a single JVM on a machine with 4 cores and 16 GB of memory. The biggest tuning challenge was the 2.3 version of Chat, which could generate as much as 200 MB of garbage per second. Almost all of it dies immediately after creation. The only way to survive this is to use a large New space. As of 2.5 this may not longer be an issue, but we have retained the tuning.

The other key JVM issue was the amount of space allocated. This may be Solaris-specific, but we have found that if the system has to do any paging at all, garbage collections can become intolerable. We've seen GCs take as long as 10 min, during which the system is unresponsive. Unfortunately Java takes more space than you might expect from the parameters, so we adjusted the space allocation based on seeing the amount of free space in the OS. Our current memory allocation is -Xmx10500m, i.e. 10.5 GB. You'd think more could be allocated on a 16 GB system, but any more than that is unsafe on our Solaris 10 systems.

With our current parameters, the only pauses longer than 1 sec are full GCs. They pause for about 35 sec. We see them about once a day per front end.

The current settings are the following:

```
JAVA_OPTS=" -d64 -Dsun.lang.ClassLoader.allowArraySyntax=true -Xmx10500m -Xms10500m -Xmn2500m
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:CMSInitiatingOccupancyFraction=80 -XX:MaxPermSize=512m
-XX:PermSize=512m -XX:+DisableExplicitGC -XX:+DoEscapeAnalysis -Dhttp.agent=Sakai "
```

I have split this into 3 lines for readability. It should be a single line in the file.

-Dsun.lang.ClassLoader.allowArraySyntax=true is currently needed for Java 6 and later, until Sakai is updated.

-Xmn2500m allocated 2.5 GB to new, as explained above.

XX:+UseConcMarkSweepGC -XX:+UseParNewGC selects the low-pause GC, which is currently the best GC for large interactive applications.

-XX:CMSInitiatingOccupancyFraction=80 is probably Java 6 specific. While Java 6 tunes itself fairly well, we find that it often doesn't start the incremental GC soon enough, thus leading to unnecessary full GC's. This causes it to start sooner.

-XX:MaxPermSize=512m -XX:PermSize=512m is a minimum for perm. You may need more. We recommend setting PermSize to MaxPerSize. If you let perm expand as needed, it will do a full GC every time it needs to expand. This tends to cause visible pauses. There's actually a way to do concurrent GC of perm, which might be worth trying.

-XX:+DisableExplicitGC is because we found a place where Sakai was explicitly requesting a full GC, for no good reason.

-XX:+DoEscapeAnalysis is experimental, and probably won't be needed in Java versions other than 6.

-Dhttp.agent=Sakai is necessary for RSS to work properly.

Other issues that can affect garbage collection

In addition to tuning, we found a few cases where local code used System.exec. This turns out to be a really, really bad idea. It causes the entire JVM to be duplicated. In theory this shouldn't matter, as the new copy should go away almost immediately, but we saw situations where the system was slowed for 30 min or so. This behavior was present in our WebCT conversion tool and the Respondus import tool. We're no longer using the WebCT converter and I fixed the Respondus tool not to use it.

We had updated inactiveInterval@org.sakaiproject.tool.api.SessionManager in sakai.properties to 8 hours. We prefer to use long idle timeouts, in order to avoid users losing work. However the longer the timeout, the more users have sessions active. We ran out of memory in the JVM one day. I am reasonably certain that it was not a tuning problem or a bug: we just had too much data. Putting the idle timeout back to 2 hours seems to have fixed it.

Jgroups timeout

Jforum uses Jgroups to maintain a single distributed cache of data on postings. When we have a long garbage collection on one system, other systems can give up on it. Jgroups is supposed to recover from this, but does not. The result is typically sites where Jforums shows no postings even though postings are actually there. We improved it by raising the timeout to be longer than the longest garbage collection time. You can see the times by looking for "Total time for which application threads were stopped" in catalina.out, assuming you use the recommended logging parameters. In jforum-cache-cluster.xml, we use

```
  <FD timeout="10000" max_tries="12" shun="true" up_thread="true" down_thread="true"/>
```

That results in 12 retries, each 10 sec (10,000 millisec), i.e. 120 sec. That's normally long enough that it doesn't result in a timeout at Rutgers.

However I haven't managed to find any parameters for Jgroups that works consistently. I have started running a cron job every 10 min on each front end, which flushes the Jforums cache. The cron job calls

/opt/sfw/bin/curl -u jforum:XXXX http://localhost/etudes-jforum-tool/clearcache.jsp

clearcache.jsp resides in .../webapps/etudes-jforum-tool. It is

```

<%@ page import="org.etudes.jforum.repository.ForumRepository" %><%
%><%@ page import="org.etudes.jforum.ConfigLoader" %><%

ForumRepository.clearCourseCategoriesFromCache();

out.println("cleared");
```

I think we may have had to change the declaration of clearCourseCategoriesFromCache in ForumRepository.java to be public.

XXXX is actually a password. This must match an entry in .../tomcat/conf/tomcat-users.xml:

```
  <user username="jforum" password="XXXX" roles="user"/>
```

Mysql jdbc connection

Please note: we are carrying over this configuration from Sakai 2.4. The underlying bug in DBCP has been fixed, but we don't think the fixed version has made it to Sakai as of Sakai 2.7.

With the default parameters, there is a problem talking to Mysql: The connection pools code that Sakai uses does additional database operations for each query: It checks that the connection is valid and then resets any connection parameters that do not have their default value. Each of these operations generates a database query. What's worse, those queries are done under a global lock, so only one connection can proceed at a time. With a quad-processor system, you lose much of the benefit from 3 of your processors, and generates lots of extra database queries.

I recommend the following parameters in sakai.properties. This is as of Sakai 2.7, for Mysql 5.1. For Mysql 4.x you would presumably use MySQLInnoDBDialect.

```
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
vendor@org.sakaiproject.db.api.SqlService=mysql
driverClassName@javax.sql.BaseDataSource=com.mysql.jdbc.Driver
url@javax.sql.BaseDataSource=jdbc:mysql://xxxx:3306/sakai?useUnicode=true&characterEncoding=UTF-
8&useServerPrepStmts=false&cachePrepStmts=true&prepStmtCacheSize=4096&prepStmtCacheSqlLimit=4096
username@javax.sql.BaseDataSource=xxx
password@javax.sql.BaseDataSource=xxx
testOnBorrow@javax.sql.BaseDataSource=false
validationQuery@javax.sql.BaseDataSource=
defaultTransactionIsolationString@javax.sql.BaseDataSource=TRANSACTION_READ_COMMITTED
initialSize@javax.sql.BaseDataSource=300
maxActive@javax.sql.BaseDataSource=300
maxIdle@javax.sql.BaseDataSource=300
minIdle@javax.sql.BaseDataSource=0
```

The JDBC parameters enable prepared statement caching. With Mysql 4.1 I had additional parameters in the Mysql URL, but they caused problems with Mysql 5.1.

This causes sakai to open 300 database connections to Mysql. This may be unnecessary in the current configuration. If you do this, make sure my.cnf is set for lots of connections. We use

```
max_connections = 1600
open_files_limit = 3000
```

If you're going to disable testOnBorrow (which is a key part of our improvements), you also need to disable timeouts on the server side, i.e. in my.cnf add

```
wait_timeout = 31536000
```

There is another mysql-related issue. Sakai seems not to close out connections all the time. This can result in some operations appearing to fail, although they may take effect much later. The most common seems to be "add participant" in Site Info. To fix this, we have patched the kernel

```



--- kernel/api/src/main/java/org/apache/commons/dbcp/SakaiPoolableConnectionFactory.java        (revision 2206)
+++ kernel/api/src/main/java/org/apache/commons/dbcp/SakaiPoolableConnectionFactory.java        (revision 2207)
@@ -238,7 +238,8 @@
                         }

                         conn.clearWarnings();
-                        // conn.setAutoCommit(true);
+                        // Rutgers: uncommenting setAutoCommit to see if that solves the held locks problems
+                        conn.setAutoCommit(true);
                 }
                 if (obj instanceof DelegatingConnection)
                 {
```

For reference, here is our my.cnf for Mysql 5.1. This is for two 4-core Xeon's and 32 GB of memory. However this configuration does not attempt to use all 32 GB of memory.

my.cnf