Getting the most out of Outrigger

June 11, 2002

Abstract

Outrigger is the name of an implementation of the JavaSpaces™ Service Specification of Jini™ Network Technology, built for, and contributed to the Jini Community^{sm} by Sun Microsystems. This paper discusses some performance tuning options available to developers deploying Outrigger on Sun's Solaris™ Operating Environment (Solaris OE).

Overview

Jini Network Technology (hereafter know simply as "Jini technology") utilizes the Java™ programming language and the associated virtual machine as its execution environment. Developers have a number of tools at their disposal to accomplish performance measurement and analysis of their applications written in the Java programming language. Applications consisting of a single executable running on a single CPU can be optimized in relatively straightforward ways using well proven, conventional means. Distributed applications, however, provide additional challenges. This paper provides some suggestions for increasing the performance of a distributed application that uses Outrigger.

Many of the interesting things discovered during a recent project to assess and improve the performance of Outrigger can be credited to the Scalable Infrastructure (SI) team at Cisco Systems. Cisco's significant and continued contributions to the Jini Community in general, and this ongoing investigation in particular, are sincerely appreciated.

Discussion

The following list of recommendations was developed during a brief project to assess the gross performance aspects of the Outrigger service. Specific performance numbers such as "operations against a space per second" are not provided because such a number would depend on too many factors to be useful for a general discussion. Nevertheless, there were some factors that significantly improved (or degraded) performance that should transcend specific machine and network configuration details. It is these factors that are discussed here.

Note: Many of the following specific recommendations are for users developing their applications on Sun Microsystem's Solaris 8 OE (or above). There are, however, a couple of noteworthy suggestions and observations that should benefit users of all platforms. The list follows below:

Use the new thread library

A new `libthread` library is available (usually installed in `/usr/lib/lwp`) which improves overall thread performance. This library can be included in your program's execution by setting the environment variable `LD_LIBRARY_PATH` to include this directory. A full discussion of this can be found at `http://java.sun.com/docs/hotspot/threads/threads.html`.

Beware of blocking actions against the space

Outrigger uses a clever protocol between its proxy and its service to manage blocking 'takes' and 'reads'. This protocol, however, may not scale sufficiently for high loads or large numbers of clients. Before specific recommendations and observations are provided, an overview of the Jini Technology Starter Kit (starter kit) v1.2.1 space proxy protocol is in order.

When a client issues a blocking operation such as a 'take', the proxy forwards the request to the Outrigger service where a match to the provided template is sought. If a match is found that is not locked by a transaction, it is returned immediately to the proxy and given to the calling client. If such a match cannot be immediately returned, then the proxy and service are obligated to wait (for up to the amount of time specified in the timeout of the request) to see if such an entry does become available. Rather than tying up a thread and connection (and this is the clever part), the service immediately returns to the proxy an *event registration* which can then be used as part of a *notify* later.

Say a blocking 'take' is made by a given client with a thirty minute time out. Also assume, for this example, that the initial probe within the space did not locate a matching, available entry. Instead of having this remote call sit in the space, it returns immediately to the caller (proxy) with an event registration object. Now assume that fifteen minutes later a matching entry is written to the space. Outrigger will then send an event notification (remote call) to the proxy. Upon receipt of this notification, the proxy will again probe the space searching for an entry matching the template. If one is located, the event registration is cancelled and the matching entry is returned to the client.

In the above example, there were four remote calls to accomplish the blocking 'take' operation:

1. the proxy does the initial probe into the space, but gets an event registration rather than an entry,
2. the space later does a remote call to deliver an event to the proxy when a matching entry appears in the space,
3. a second probe is made to the space by the proxy after receiving the event with the hope of locating that entry, and
4. once an entry is located (in step 3) the event registration is cancelled.

Four remote calls were made in the above interaction but all the calls were short-lived. One alternative to this scheme, of course, would be to design the protocol to ensure only one call is made with the call active for the entire time. This would tie up resources on both the client-side and the Outrigger side. For example, a thread (or perhaps several threads) would be tied up within Outrigger if the call did not return immediately. On many systems, where threads are cheap (such as Solaris OE), this wouldn't be an issue. However, generating dozens or even hundreds of threads in some operating systems would be disastrous.

Over a course of fifteen minutes, four such calls to complete a particular 'take' operation does not seem onerous. If this complex interaction were required to complete within twenty milliseconds, however, there may be problems. Further, the scheme outlined above suffers from a "fan out" problem as the number of clients increases. With one client, a single notify is sent for each matching entry. With ten clients, a thundering herd of ten such event notifications would be sent, which would trigger ten more probes to the space. And, while one might develop clever mechanisms to reduce the number of notifications for 'take' operations (since, presumably, only one client will be able to have the 'take' operation succeed anyway), no such mechanisms are possible for blocking 'read' operations. All blocking 'read' clients must be notified.

There are scaling issues associated with using blocking 'read' and 'take' operations with the current design of Outrigger. Such blocking can be caused by either an absence of a matching entry in the space or by transactional locks on matching entries (if transactions are utilized in the calls to the space). While there is little one can do to avoid blocking in the case where transactions are used, non-transaction-based interactions with the space can be written to avoid blocking entirely. This is done by specifying a zero length time out on 'take' and 'read' calls, or by using 'takeIfExists' and 'readIfExists' forms of these calls (since the 'IfExists' calls will, by definition, never block when no transactions are specified).

There are additional problems with the blocking take operations in the v1.2.1 starter kit (and earlier) Outriggers. These problems are enumerated below:

1. A bug (#4671487) caused Outrigger to create many more threads than necessary to service event delivery.

   There is a pool of threads within Outrigger used to orchestrate the delivery of event notifications to clients. Because of a bug within this area of the code, many thousands of threads may be unnecessarily created during a typical run.

2. Event notifications are queued inside Outrigger to conserve resources.

   This, in and of itself, isn't necessarily a problem. However, the size of the pool of threads (mentioned above) may be insufficient to handle high-traffic applications. Further, the delays associated with this extra level of queueing may introduce unacceptable performance.

   Two additional "knobs" are being considered for future versions of Outrigger. The first "knob" would allow the size of the thread pool used for event notification delivery to be configured at runtime. A second "knob" would set a threshold value to be used by this logic. When the number of event notifications exceeds the threshold value, additional resources would be employed to handle the work. For faster delivery, setting the threshold to "1" ensures that no additional queuing will be employed and the notification will be sent with the least possible delay.

Note that even after the two issues described above are addressed, the general scaling problem associated with the current implementation of Outrigger's blocking queries is not remedied. High performance applications may need to avoid blocking queries in order to scale well.

Consider using –server

Sun's virtual machine for the Java platform (VM) includes several command-line options that can help tune applications. One such option is the `-server` command option. VMs which have this flag set use a vastly more sophisticated runtime compiler that produces more optimized code for long-running applications and performs network operations better than those VMs which employ the command line option `-client` (which is the default). FrontEndSpace users should note that the placement of this option should be such that it is picked up by the *server* VM and not the *setup* VM. (See the FrontEndSpace man pages for more information on the proper placement of options within the command line.)

Unfortunately, there is an outstanding bug (#4615373) that causes a `ClassCastException` to be thrown on occasions. The defect only seems to exhibit itself in the `-server` VM. [This should be fixed in the J2SE™ 1.3.1_04 and J2SE 1.4.0_02 releases and is thought to be not a problem in the J2SE 1.4.1 release.]

Measure system performance with varying memory configurations

There are several command-line options that can be used to control memory allocations for the heap and stack. Use the `-mx`, `-ms`, and `-ss` command-line options to vary the footprint of the application to see which configuration yields the best performance. Note that providing an application with excess memory may not increase overall system performance. In some cases, excessive memory allocations may even degrade performance.

Use `snapshot` *where possible*

Section JS.2.6 of the JavaSpaces Service Specification identifies a `snapshot` method that can be used to reduce the overhead of passing a specific Entry repeatedly to other method calls. Applications using Entries that never change (such as templates) can reduce the marshalling overhead of method calls to the space by passing the snapshot of the Entry rather than the Entry itself.

Beware RMI distributed garbage collection

J2SE supports remote method invocation (RMI). RMI's distributed garbage collection (DGC) algorithm depends on the timeliness of local garbage collection (GC) activity. To address the situation where the local GC activity resulting from normal application execution is not sufficient for effective DGC, an RMI implementation may take steps to stimulate local GC to detect unreachable objects sooner than it otherwise would.

In Sun's J2SE implementations from 1.2 through 1.4.1, the particular steps taken are periodic invocations of the `System.gc` method. Each invocation causes the local GC to scan the entire heap (often called a *full GC*). The actual period of these `System.gc` invocations—the maximum interval between full GCs enforced by the RMI implementation—is, in the worst case, determined by the minimum of the values of the following two implementation-specific system properties:

```
sun.rmi.dgc.client.gcInterval

sun.rmi.dgc.server.gcInterval
```

where the values of these system properties are interpreted as `long` integers in units of milliseconds, and the default value for each of them is 60000 (one minute).

These periodic full GCs, when occurring at the default interval, can have a disastrous effect on the performance and scalability of a distributed application. Since each full GC suspends all application threads in a VM while the entire heap is scanned, applications with especially large heaps may be subject to pauses that impede application response time. Also, recall that these full GCs are only done to stimulate local GC—in many cases, normal application execution will cause sufficient local GC activity for DGC to be effective. Finally, a given RMI-based distributed application may not need DGC to occur on the short time order of the default GC intervals, or it may not need DGC behavior at all. Thus, it is often appropriate to set both of these `gcInterval` properties to values much higher than their default intervals of one minute.

The `sun.rmi.dgc.server.gcInterval` system property is used to ensure the timely release of the RMI implementation's non-daemon threads within the local VM when no associated remote objects remain exported. This allows the VM to exit naturally if there are no other non-daemon threads. For many applications (that do not care about this automatic VM exiting behavior), this system property can be safely set to an effectively infinite value, such as `Long.MAX_VALUE`:

```
-Dsun.rmi.dgc.server.gcInterval=0x7FFFFFFFFFFFFFFF
```

The effect of the `sun.rmi.dgc.client.gcInterval` system property is to place an upper bound on the time from when the set of live remote references to a given remote object becomes logically unreachable, to the time when the local GC detects that remote references have become unreachable. When that happens, the VM's RMI implementation can notify the RMI implementation in the remote object's VM that it no longer holds any live references to the remote object. The longer this time, the longer it may take to garbage collect remote objects in other VMs. Depending on how important this DGC latency is to the distributed application, this system property can be set to an higher interval, perhaps on the order of hours or even days.

With Outrigger's use of remote objects, and in particular their creation rate and life cycle, RMI's DGC behavior is unimportant. Therefore, it is desirable to set both of these system properties to high values in VMs that host or use Outrigger.

Conclusions

The included observations and recommendations are for users of the Outrigger included in the v1.2.1 starter kit. (They also apply to previous versions of Outrigger.) It is strongly recommended that users read the release notes accompanying each release for important information regarding, among other things, system performance characterizations, limitations, and enhancements associated with that particular release. It is anticipated, for example, that future releases of Outrigger will address some or all of the problems discussed above. The best place to find such information will be the release notes for that release or its accompanying man pages.

Application performance tuning is always challenging. Distributed applications inherit all of the traditional problems while creating quite a few new and interesting ones as well! Careful measurement is, if possible, even more important for distributed application tuning. This paper has provided starting points for anyone tuning a distributed application that uses Outrigger.