High-performance Ajax with Tomcat Advanced I/O

Reduce the number of concurrent threads by using a different request-handling model

Skill Level: Intermediate

Adriaan de Jonge (adriaandejonge@gmail.com)
Software Professional

30 Sep 2008

Using Non-Blocking I/O (NIO) improves server performance drastically because of its efficient use of system resources (threads). The gain in performance is noticeable in Asynchronous JavaScript + XML (Ajax) applications with long polling mechanisms. It also lets you control system-resource usage on a server under pressure. This article explains how to optimize your server for performance during the handling of both Ajax and regular requests.

Browsers communicate with Web servers by sending a request and displaying the response received from the server. This sounds trivial, but it's relevant for this article for several reasons.

Rich Web applications are increasingly trying to offer functions that behave differently than a request/response model. Ajax frameworks are abstracting away from the underlying request/response model, offering a model that behaves like a graphical user interface (GUI) running on a client, like a local heavyweight Microsoft® Windows® or KDE application. This means that instead of responding to requests, you create code that responds to events. Fine-grained event handling is the key to a rich user interface.

But underneath, there is still a request/response model. Not only is this a limited programming model to work with—you can use API abstractions to work around that—but in many cases this model causes performance issues and harms the server's ability to control its resources when running under pressure. These problems get worse when you use this model for purposes it wasn't originally meant for.

For example, a browser client may want to be notified of changes on the server. Originally, this was impossible; you could only mimic this behavior by having a page completely reload itself every five minutes. Since the introduction of JavaScript code and Ajax, new possibilities let the page accomplish change notifications in a far more efficient and elegant fashion.

The page can poll the server regularly, creating a large number of server requests and a considerable latency between the event and the notification. Or it can keep a connection open to the server waiting for a response. The second model is called a *long poll*: This article discusses this polling model—because it allows the quickest response to events on the server—as well as some hurdles to overcome.

In the classic servlet model, an open connection means a dedicated thread is doing nothing but waiting until it needs to update the client in response to an event. Threads are relatively expensive, and a limited number of threads are available on a server. An increase in the number of concurrent visitors quickly adds to the resource usage on the server. A single visitor doesn't just mean a few incidental page requests; it also means an open connection (with a waiting thread) for as long as the visitor stays. A few hundred visitors leaving their browsers open while they're away from their computers can quickly become problematic. No amount of abstraction will resolve these performance issues.

The solution

The solution to a low-level problem should be found in a low-level API. While sticking with the request/response model, you can use Non-Blocking I/O (NIO) to keep a connection open without wasting waiting threads. To facilitate NIO in a servlet, you need an event-based API that initiates appropriate reading and writing actions on the open connections at the right time. Tomcat 6 offers the CometProcessor API to facilitate such an event-based model. This article introduces the CometProcessor API.

NIO

First, you need to know a little about NIO. In classic blocking I/O, you read a stream from beginning to end in one thread that waits until the entire stream is finished. This works great for a stream with a short lifetime that needs to be processed in one iteration: you start reading, you read until there is no more data, and then you close the connection. In one iteration, this usually takes little time, so the system resources are claimed for only a short period. A large number of subsequent short-lived connections is no problem.

For notification, you want a mechanism where you read only when there is something to read and write only when there is something to write, but you keep the connection open to respond quickly to occurring events. To facilitate this, you need NIO, which has been available as part of the Java™ language since version 1.4.

Using NIO, you don't loop through a stream until it's finished. You loop through the stream until no more data is available, and then the system is free to do something else. When an event occurs that affects the stream — for example, if more data comes in — you continue looping through the stream. Listing 1 shows an example handler for such an event; pay attention to `is.available()`.

Listing 1. A single-read iteration on an InputStream

```
InputStream is = request.getInputStream();
byte[] buf = new byte[512];
do {
    int n = is.read(buf); //can throw an IOException
    if (n > 0) {
        //read n bytes
    } else if (n < 0) {
        //error
        return;
    }
} while (is.available() > 0);
```

You can use `OutputStreams` and `Writers` in a similar way. Make sure you flush the data after you're done writing an iteration; otherwise, it may be stuck in a buffer until the next iteration is written to the stream.

Comet

This article focuses on Tomcat 6, which supports NIO in its Advanced IO modules. Similar functionality is available in Jetty, but the API is different. Of course, you don't want to write solutions that are specific to a single container implementation. Good news is on the horizon, with the upcoming Servlet 3.0 specification: when it's ready and implemented by common containers, there will be a standardized mechanism supporting event-based NIO.

For now, you can facilitate this functionality with the Tomcat-specific CometProcessor. It's worth experimenting with even before the Servlet 3.0 spec is released. The Tomcat implementation seems easy to use if you want to set up your own framework; therefore, it's used in this article. The Jetty implementation integrates well with existing frameworks and solutions and could be the practical choice if you want to use the added value of Comet without diving into the technical details. In Resources there is also a pointer to an article demonstrating similar techniques with WebSphere® Community Edition.

For the CometProcessor to work on Tomcat 6.0, you need to change two things: one in the configuration and the other in the code.

First, find the connector in your server.xml file. It should look like this:

```
<Connector connectionTimeout="20000" port="8080"
protocol="HTTP/1.1" redirectPort="8443"/>
```

Replace the HTTP/1.1 protocol with the following:

```
<Connector connectionTimeout="20000" port="8080"
protocol="org.apache.coyote.http11.Http11NioProtocol"
redirectPort="8443"/>
```

The second step is to let your servlet implement the `org.apache.catalina.CometProcessor` interface. This interface requires you to implement a single method called `event()`. The `Http11NioProtocol` configured in the first step calls the `event()` method instead of `doGet` or `doPost` to process requests.

The most basic implementation of a Comet-aware servlet, which does nothing (yet), is shown in Listing 2.

Listing 2. Handling a basic Comet event

```
package eu.adriaandejonge.comet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

import org.apache.catalina.CometEvent;
import org.apache.catalina.CometProcessor;
import org.apache.catalina.CometEvent.EventType;

public class CometServlet extends HttpServlet implements
CometProcessor {

            public void event(CometEvent event) throws
IOException, ServletException {
                       if(event.getEventType() ==
EventType.BEGIN) {
                                  // fill in code handling
here
                       }
                       // and continue handing other
events
            }
}
```

The event is thrown when the connection starts (`BEGIN`), every time new data is available (`READ`), and when the connection ends (`END`). That's the happy flow. In case of an error, you receive an `ERROR` event, which can have several subevent types. For example, it's interesting to separate a `TIMEOUT` event from a `SERVER_SHUTDOWN` event, for logging purposes or to investigate the need for transaction rollbacks.

Comet lets you specify different connection time-outs for different events. This means you can give regular requests a short lifetime, but you can also extend this lifetime to several minutes for a mechanism that responds to long polling requests. This way, you won't make the error of increasing all timeouts to facilitate long polling and then end up with unexpected trouble when regular requests keep hanging without being terminated like they're supposed to.

Ajax handling

Looking at the code examples, it may seem that receiving `CometEvents` is much more difficult than handling a regular request. In some ways, it is. At least, the amount of work seems to increase. A single request is split into multiple phases, so instead of handling a single `doGet`, you must handle one `BEGIN`, at least one `READ`, and one `END`.

This complexity is partly caused by the common mistake of implementing the `event()` method with a large `if` statement to capture all the handling logic. The problem isn't entirely in the `if` statement, but mostly in the amount of code in the `if` statement. This approach is flawed, but for more reasons than code maintainability.

The `if` statement lets you read from the connection when data is available and process the requests immediately. For a notification mechanism, this isn't what you want. You need it the other way around: the connections should remain open until an event arises that you want to respond to by sending new information to the client. You can do that several ways, but they have one thing in common: they're all based on the introduction of a proper object model that handles the program logic instead of trying to capture all the code in the `if` statement.

One option is to keep a central queue and use it once to deliver all notifications when an event arises. After that, it's the client's responsibility to reconnect to the queue. On `END` or `ERROR` events, however, the `CometProcessor` needs to deregister the connections from the queue. See Listing 3.

Listing 3. Registering a Comet event

```
package eu.adriaandejonge.comet;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.catalina.CometEvent;
import org.apache.catalina.CometProcessor;
import org.apache.catalina.CometEvent.EventType;

public class CometServlet extends HttpServlet implements
CometProcessor {

           private static final long serialVersionUID =
3616604581787849064L;

           private static final String EVENT_REGISTRY =
"event.registry";

           private EventRegistry eventRegistry;

           public CometServlet() {
      eventRegistry = (EventRegistry)
this.getServletContext()
.getAttribute(EVENT_REGISTRY);
           }

           public void event(CometEvent event) throws
IOException, ServletException {
                      HttpServletRequest request =
event.getHttpServletRequest();
                      HttpServletResponse response =
event.getHttpServletResponse();
                      if(event.getEventType() ==
EventType.BEGIN) {
eventRegistry.register(request, response);
                      } else if(event.getEventType() ==
EventType.END) {
eventRegistry.deregister(request, response);
                      } else if(event.getEventType() ==
EventType.ERROR) {
eventRegistry.deregister(request, response);
                      }

           }
}
```

An alternative is to keep a central `ListenerRegistry` that clients can hook into as event listeners. The difference is that the connection remains open after an event notification. NIO lets you return to the connection later and send additional data. On the client side, you can use a standardized API such as Bayeux to support this. But currently, it's implemented only by Dojo and works best with the Jetty Comet implementation.

Basically, Comet is a server-side technique. On the client, handling requests isn't much different from a regular request. To demonstrate, Listing 4 shows a test of the most basic Ajax request possible: the classic `XMLHttpRequest`-based example from Wikipedia. It responds well to an event sent by the Comet server.

Listing 4. Sending an Ajax request

```
<script>
function ajax(url, vars, callbackFunction) {
  var request =  new XMLHttpRequest();
  request.open("POST", url, true);
  request.setRequestHeader("Content-Type",
                           "application/x-javascript;");

  request.onreadystatechange = function() {
    if (request.readyState == 4 && request.status == 200)
{
      if (request.responseText) {
        callbackFunction(request.responseText);
      }
    }
  };
  request.send(vars);
}

function testFunction(myText) {
        alert("myText = " + myText);
}


</script>

<input type="button" caption="test"
onclick="ajax('http://localhost:8080/comettest/CometServlet',
        '', testFunction);">
```

This implies that you can use other frameworks like Prototype to handle Comet events. Nevertheless, it's an interesting exercise to read the Bayeux specification and learn about a more formalized event-exchange protocol.

General-performance tuning

Using the queue or event listener in combination with the CometProcessor saves a lot of open threads, but it doesn't help control the number of concurrent threads processing pages to be served back to the browser. As a result, the server may still be flooded with a large number of requests that it wants to process concurrently. The event-based model introduced by the CometProcessor interface may also help improve the control you have over performance in general.

The solution is to introduce a mechanism that handles the events as quickly as possible, does nothing else, and then registers them with a `WorkerQueue` and ends the handling. At the same time, you should have a limited number of `WorkerThreads` running that pick up requests from the `WorkerQueue` and handle them one by one. This way, there are never more concurrent threads processing pages to be served to the browser than there are `WorkerThreads` configured to do so.

When the number of requests increases and the number of worker threads is too small, it may take a long time for requests to be processed. But the server will never (or at least rarely) collapse under its load. The request-processing time and the queue length are proper measures for the system administrator to monitor in order to make decisions about adding servers or shutting down superfluous servers. This approach to request handling works nicely in a cloud computing environment.

The code to pass Comet events to such a handler mechanism may look like Listing 5. This example omits the actual handling of the queue because it's specific to your requirements and involves many concurrency subtleties. But the code shows how to implement a large `if` statement with a more object-oriented solution. It would be even more object oriented if you replaced the last bits of the `if` statement with a `Map` connecting `EventTypes` to `EventHandler` factory objects. But, this example shows the general idea.

Listing 5. Registering the Comet event on an EventWorker queue

```
package eu.adriaandejonge.comet;

import java.io.IOException;

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

import org.apache.catalina.CometEvent;
import org.apache.catalina.CometProcessor;
import org.apache.catalina.CometEvent.EventType;

public class CometServlet extends HttpServlet implements
CometProcessor {

   private static final long serialVersionUID =
3657376753893664777L;

           private static final String EVENT_WORKER =
"event.worker";

           private EventWorker eventWorker;

           public CometServlet() {
                      eventWorker = (EventWorker)
this.getServletContext()
.getAttribute(EVENT_WORKER);
           }

   public void event(CometEvent event) throws IOException,
ServletException {
                      HttpServletRequest request =
event.getHttpServletRequest();
                      HttpServletResponse response =
event.getHttpServletResponse();
                      if (event.getEventType() ==
EventType.BEGIN) {
                                 eventWorker.enqueue(new
BeginEvent(request, response));
                      } else if (event.getEventType() ==
EventType.READ) {
                                 eventWorker.enqueue(new
ReadEvent(request, response));
                      } else if (event.getEventType() ==
EventType.END) {
                                 eventWorker.enqueue(new
EndEvent(request, response));
                      } else if (event.getEventType() ==
EventType.ERROR) {
                                 eventWorker.enqueue(new
ErrorEvent(request, response));
                      }
           }
}
```

Serving both

A much more common scenario requires you to handle both general requests and Ajax requests. Some of the Ajax requests must be handled instantly, other Ajax requests should be handled as long polls, and normal requests should be posted on the `WorkerQueue`.

It makes sense to create multiple servlets to handle these different requests and give them each their own responsibility. The problem is that the connection timeout is the same for the whole HTTP connector defined earlier in this article -- at least, the configured value of the timeout is the same. Comet events let you specify another connection timeout per request. This may be dynamically computed, but it helps when you specify different values for different servlets.

Conclusion

Optimizing your server for performance during the handling of both Ajax and regular requests is as simple as implementing a `CometProcessor` interface and adding a different protocol handler in your server.xml file. To keep your request-handling manageable, however, requires software development skills. You need to think about queues, event listeners, worker threads, and registering and deregistering request/response combinations.

If you develop your own framework, you should be willing to go through this trouble in order to get a more stable, better-performing server. But when you're developing a simple Web application, the complexity of event handling may be more expensive than the cost of additional resources.

In the latter case, the best way to go is to limit your use of the `CometProcessor` to event-notification requests and fall back on the classic `doGet` and `doPost` for all your other requests.

Keep an eye open for the Servlet 3.0 spec when it's released. It may encourage the further adoption of NIO on Web servers.

Resources

- The Tomcat Advanced I/O documentation provides more background information.
- Check the status of the upcoming Java Servlet 3.0 Specification.
- See another example of NIO using similar techniques with WebSphere Community edition in "Developing Rich Internet Applications for WebSphere Application Server Community Edition" (developerWorks, September 2008).
- Get more information about Jetty, an open-source, standards-based, full-featured Web server implemented entirely in Java.
- Read more about the Bayeux protocol for routing JSON events.
- "The Servlet API and NIO: Together at last" (developerWorks, February 2004) gives you more information about Non Blocking I/O.

About the author

Adriaan de Jonge
Adriaan de Jonge is a software professional currently working for the Dutch government, juggling a few projects in several roles. Adriaan has written XML-related articles for IBM developerWorks and Amazon. You can reach Adriaan at adriaandejonge@gmail.com.