# How to use Asynchronous Servlets to improve performance

October 10, 2013 by Nikita Salnikov-Tarnovski  Filed under: **Java** **Locked Threads**

This post is going to describe a performance optimization technique applicable to a common problem related to modern webapps. Applications nowadays are no longer just passively waiting for browsers to initiate requests, they want to start communication themselves. A typical example would involve chat applications, auction houses etc – the common denominator being the fact that most of the time the connections with the browser are idle and wait for a certain event being triggered.

This type of applications has developed a problem class of their own, especially when facing heavy load. The symptoms include starving threads, suffering user interaction, staleness issues etc.

Based on recent experience with this type of apps under load, I thought it would be a good time to demonstrate a simple solution. After Servlet API 3.0 implementations became mainstream, the solution has become truly simple, standardized and elegant.

But before we jump into demonstrating the solution, we should understand the problem in greater detail. What could be easier for our readers than to explain the problem with the help of some source code:

```
@WebServlet(urlPatterns = "/BlockingServlet")
public class BlockingServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) throws ServletException, I
    try {
      long start = System.currentTimeMillis();
      Thread.sleep(new Random().nextInt(2000));
      String name = Thread.currentThread().getName();
      long duration = System.currentTimeMillis() - start;
      response.getWriter().printf("Thread %s completed the task in %d ms.", nar
    } catch (Exception e) {
      throw new RuntimeException(e.getMessage(), e);
    }
  }
```

The servlet above is an example of how an application described above could look like:

- Every 2 seconds some event happens. E.g. stock quote arrives, chat is updated and so on.
- End-user request arrives, announcing interest to monitor certain events
- The thread is blocked until the next event arrives
- Upon receiving the event, the response is compiled and sent back to the client

Let me explain this waiting aspect. We have some external event that happens every 2 seconds. When new request from end-user arrives, it has to wait some time between 0 and 2000ms until next event. In order to keep it simple, we have emulated this waiting part with a call to *Thread.sleep()* for random number of ms between 0 and 2000. So every request waits on average for 1 second.

Now – you might think this is a perfectly normal servlet. In many cases, you would be completely correct – there is nothing wrong with the code until the application faces significant load.

In order to simulate this load I created a fairly simple test with some help from JMeter, where I launched 2,000 threads, each running through 10 iterations of bombarding the application with requests to the */BlockedServlet*. Running the test with the deployed servlet on an out-of-the-box Tomcat 7.0.42 I got the following results:

- **Average response time: 9,324 ms**
- Minimum response time: 5 ms
- Maximum response time: 11,651 ms
- **Throughput: 193 requests/second**

The default Tomcat configuration has got 200 worker threads which, coupled with the fact that the simulated work is replaced with the sleep cycle of average duration of 1000ms, explains nicely the minimum and maximum response times – in each second the 200 threads should be able to complete 200 sleep cycles, 1 second on average each. Adding context switch costs on top of this, the achieved throughput of 193 requests/second is pretty close to our expectations.

The throughput itself would not look too bad for 99.9% of the applications out there. However, looking at the maximum and especially average response times the problem starts to look more serious. Getting the worst case response in 11 seconds instead of the expected 2 seconds is a sure way to annoy your users.

Let us now take a look at an alternative implementation, taking advantage of the Servlet API 3.0 asynchronous support:

```
@WebServlet(asyncSupported = true, value = "/AsyncServlet")
public class AsyncServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  protected void doGet(HttpServletRequest request, HttpServletResponse response
    Work.add(request.startAsync());
  }
}
```

```
public class Work implements ServletContextListener {
  private static final BlockingQueue queue = new LinkedBlockingQueue();

  private volatile Thread thread;

  public static void add(AsyncContext c) {
    queue.add(c);
  }

  @Override
  public void contextInitialized(ServletContextEvent servletContextEvent) {
    thread = new Thread(new Runnable() {
      @Override
      public void run() {
        while (true) {
          try {
            Thread.sleep(2000);
            AsyncContext context;
            while ((context = queue.poll()) != null) {
              try {
                ServletResponse response = context.getResponse();
                response.setContentType("text/plain");
                PrintWriter out = response.getWriter();
                out.printf("Thread %s completed the task", Thread.currentThread
                out.flush();
              } catch (Exception e) {
                throw new RuntimeException(e.getMessage(), e);
              } finally {
                context.complete();
              }
            }
          } catch (InterruptedException e) {
            return;
          }
        }
```

```
        }
      });
      thread.start();
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
      thread.interrupt();
    }
  }
```

This bit of code is a little more complex, so maybe before we start digging into solution details, I can outline that this solution performed **~75x better latency- and ~10x better throughput-wise**. Equipped with the knowledge of such results, you should be more motivated to understand what is actually going on in the second example.

The servlet itself looks truly simple. Two facts are worth outlining though, the first of which declares the servlet to support asynchronous method invocations:

```
@WebServlet(asyncSupported = true, value = "/AsyncServlet")
```

The second important aspect is hidden in the following line

```
Work.add(request.startAsync());
```

in which the whole request processing is delegated to the *Work* class. The context of the request is stored using an *AsyncContext* instance holding the request and response that were provided by the container.

Now, the second and more complex class – the *Work* implemented as *ServletContextListener* – starts looking simpler. Incoming requests are just queued in the implementation to wait for the notification – this could be an updated bid on the monitored auction or the next message in the group chat that all the requests are waiting for.

Now, the notification arrives every 2 seconds and we have simplified this as just waiting with *Thread.sleep()*. When it arrives all the blocked tasks in the queue are processed by a single worker thread responsible for compiling and sending the responses. Instead of blocking hundreds of threads to wait behind the external notification, we achieved this in a lot simpler and cleaner way – batching the interest groups together and processing the requests in a single thread.

And the results speak for themselves – the very same test on the very same Tomcat 7.0.42 with default configuration resulted in the following:

- **Average response time: 265 ms**
- Minimum response time: 6 ms
- Maximum response time: 2,058 ms
- **Throughput: 1,965 requests/second**

The specific case here is small and synthetic, but similar improvements are achievable in the real-world applications.

Now, before you run to rewrite all your servlets to the asynchronous servlets – hold your horses for a minute. The solution works perfectly on a subset of usage cases, such as group chat notifications and auction house price alerts. You will most likely not benefit in the cases where the requests are waiting behind unique database queries being completed. So, as always, I must reiterate my favorite performance-related recommendation – measure everything. Do not guess anything.

But on the occasions when the problem does fit the solution shape, I can only praise it. Besides the now-obvious improvements on throughput and latency, we have elegantly avoided possible thread starvation issues under heavy load.

Another important aspect – the approach to asynchronous request processing is finally standardized. Independent of your favorite Servlet API 3.0 – compliant application server such as Tomcat 7, JBoss 6 or Jetty 8 – you can be sure the approach works. No more wrestling with different Comet implementations or platform-dependent solutions, such as the Weblogic *FutureResponseServlet*.

Now, if you came with me this far, I can only recommend you to follow us on Twitter. I will keep posting on relevant and interesting tech topics, I promise.

**ADD COMMENT**

SEND

**COMMENTS**

Thank you for the post. Who will call contextInitialized() method?

Dinoop p January 7, 2015 Reply

I'm sorry to disappoint you but this example is wrong :p In the first example sleep() executes every call while async implementation calls sleep once before processing queue in a loop.

By the way, your blog is great and has already learnt me a lot. Thank you for your effort.

Dzafar Sadik November 22, 2014 Reply

> Thank you for your kind words 🙂 But I still believe that second implementation is absolutely correct. We try to simulate the situation, where some events arrive every 2 seconds, e.g. brokerage info. When it arrives, every waiting request is notified at once, without further delays.
>
> I have modified the blog post to make this clearer. 2 seconds are not the request processing time. It is time between events that requests are waiting for. When event arrives, further processing is very fast.
>
> Nikita Salnikov-Tarnovski Post author November 26, 2014 Reply

Why do you need the async context? Can't it be done passing just the servlet response?

rascio October 10, 2013 Reply

I beg your pardon, passing where?

iNikem October 10, 2013 Reply

to the Work class…implement the Work class like:
static void add(HttpServletResponse response){
queue.add(response);
}

Isn't this the same? I don't understand why there is the needs of the startAsync context

rascio October 10, 2013

You cannot write into Response after servlet container has done with the current request processing. That is the whole point, why these async servlets were invented.

iNikem October 10, 2013

Ok…I read a little bit around and did some test…for what I read the async context is done for comet like stuff…I was looking if it can be useful for my stuff but no 🙂 thanks

rascio October 10, 2013

Hello !

Good post, however your two implementations are not really the same.

In your blocking servlet, "backend process mock" (your Thread.sleep) takes 2 seconds.

In your async implementation, what takes 2 seconds is only a waiting time before two polling of the queue (and not the "backend process mock"

If you really wanted to have the same test, you should have your Thread.sleep into the while ((context = queue.poll()) != null) and not outside.

Can you take a look ?
I think your performance will sharply decrease, because one worker stacking process of 2 seconds won't be fast enough.

```
while ((context = queue.poll()) != null) {
while ((context = queue.poll()) != null) {
while ((context = queue.poll()) != null) {
```

jeetemplates October 10, 2013 Reply

I still believe that second implementation is absolutely correct. We try to simulate the situation, where some events arrive every 2 seconds, e.g. brokerage info.

When it arrives, every waiting request is notified at once, without further delays.

Buy may be our first implementation is not that correct… Right now every request waits for exactly 2 seconds, which is not as it should be… Thanks for pointing this out, I will try to find some time to clarify this.

iNikem October 10, 2013 Reply

> Thanks for your quick reply.
>
> jeetemplates October 10, 2013

Hi, i don't know how to understand this sentence:

in each second the 200 threads should be able to complete 100 sleep cycles, 2 seconds each

long October 10, 2013 Reply

Great. Just to test my understanding, isn't this even easier to do with a WriteListener in Servlet 3.1? Or is that something different?

Bit Crusher October 10, 2013 Reply

> May be. Had no chance so far to learn that part of the API 🙂
>
> iNikem October 10, 2013 Reply

great. I almost ignored Servlet 3.0 API. Will help in impressing others with my skills in next project (current one is like 5 year old app being supported with bug fixes and enhancements)

Java Experience October 10, 2013 Reply

> Same here. There is no much information floating around about what that Servlets 3 are all about and why should we care. We hoped to shed a little bit more light on it 🙂
>
> iNikem October 10, 2013 Reply