Technical Note TN1157

Don't println to a Socket

CONTENTS

A common cause of deadlocks in client-server Java applications when running on Mac OS stems from improper use of the `println` method when writing to an `OutputStream` connected to a Socket. Here's why, and what you can do about it.

Updated: [Mar 1 1999]

Does This Ring A Bell?

You've written a networked client application or applet that communicates with a server using either a standard Internet protocol like HTTP or FTP, or a custom protocol that uses a similar line-oriented syntax. Your application works fine on Windows and Unix, but when you run it on a Mac, the client gets stuck when it tries to receive a response from the server. If you check the server, you find that the thread communicating with your app is similarly blocked reading a command from the client.

Chances are you've just run into a little-known Java networking gotcha (we've seen many reports of this from developers.) The bad news is that it's a bug in your app, not in MRJ. The good news is that it's easy to work around.

Back to top

println Considered Harmful

The underlying problem is a confusion between different line-breaking conventions. Line-based Internet protocols almost universally use CRLF as a line break -- that is, an ASCII carriage return (`'\r'` or hex `0D`) followed by a linefeed (`'\n'` or hex `0A`). If you're using Java's excellent stream classes to send commands using such a protocol, the temptation is to use a `PrintStream` or `PrintWriter` object and call its `println` method to send commands, since `println` sends a line break. Right?

The problem is that `println` was designed for use with file streams, and the line break it appends is *the local platform's line break* . On Windows this is a CRLF, on Unix it's an LF, and on Mac OS it's a CR. (The exact line break string is read from the system `line.separator` property.) This is clearly the right thing to do when writing to a local file, but it can cause problems when communicating with a server.

Here's what happens: Your client calls

```
out.println("HELO foo@bar.com");
```

When running in MRJ, this sends the bytes:

```
HELO foo@bar.com\r
```

The server receives the text and the CR, then expects a LF to follow the CR and waits to receive one. Actually, most servers are smart enough that if the character received following the CR is *not* an LF, they'll understand the nonstandard line break and treat the second character as the start of the next line. But in any case, the server typically waits for the second character before processing the command.

Meanwhile, the client waits to receive a response line from the server before sending any more text. Both sides are blocked

in `InputStream.read` calls ... a classic case of deadlock.

Back to top

The Easy Fix

The lesson here is not to use `println` when you need to generate a specific line break sequence. Instead, do it yourself. The above example should correctly have read:

```
out.print("HELO foo@bar.com\r\n");
```

This is pretty simple, but unfortunately requires changing all of the `println` calls in your code.

The really nice fix that won't work

There's a really nice fix that is, unfortunately, not possible. If you look at the source to the `java.awt.PrintStream` and `java.awt.PrintWriter` classes, you'll note that they use a `newLine` method to send the actual line break. It would be very elegant to make your own subclass of `PrintStream` or `PrintWriter` that overrode `newLine` to send an explicit CRLF. Unfortunately, due to a bit of misdesign, the `newLine` method was made private, not protected, so it's impossible to override. Oh well.

Fixing the server

If you happen to be developing the server as well, and can make changes to its code, then there's an even better fix you can make on the server side instead of changing the client. In the previous section, I described how the server blocks waiting for a character to follow the CR so it can tell whether it's an LF or not. A more intelligent way to handle nonstandard line breaks is for the server to process the input line immediately once it receives the CR, but to set a flag that indicates that if the next character received is an LF it should be ignored.

This way the server will not block after it receives the CR, will process the command and return a response, and the client code will receive the response and continue running. No deadlock.

Back to top

Downloadables

Acrobat version of this Note (40K). Download

Back to top
