Java Networking and Proxies

1) Introduction

In today's networking environments, particularly corporate ones, application developers have to deal with proxies almost as often as system administrators. In some cases the application should use the system default settings, in other cases it will we want to have a very tight control over what goes through which proxy, and, somewhere in the middle, most applications will be happy to delegate the decision to their users by providing them with a GUI to set the proxy settings, as is the case in most browsers.

In any case, a development platform, like Java, should provide mechanisms to deal with these proxies that are both powerful and flexible. Unfortunately, until recently, the Java platform wasn't very flexible in that department. But all that changed in J2SE 5.0 as new API have been introduced to address this shortcoming, and the purpose of this paper is to provide an in-depth explanation of all these APIs and mechanisms, the old ones, which are still valid, as well as the new ones.

2) System Properties

Up until J2SE 1.4 system properties were the only way to set proxy servers within the Java networking API for any of the protocol handlers. To make matters more complicated, the names of these properties have changed from one release to another and some of them are now obsolete even if they are still supported for compatibility's sake.

The major limitation of using system properties is that they are an “all or nothing” switch. Meaning that once a proxy has been set for a particular protocol, it will affect all connections for that protocol. It's a VM wide behavior.

There are 2 main ways to set system properties:

- As a command line option when invoking the VM
- Using the `System.setProperty(String, String)` method, assuming, of course that you have permission to do so.

Now, let's take a look, protocol by protocol, at the properties you can use to set proxies. All proxies are defined by a host name and a port number. The later is optional as, if it is not specified, a standard default port will be used.

2.1) HTTP

There are 3 properties you can set to specify the proxy that will be used by the http protocol handler:

- `http.proxyHost`: the host name of the proxy server
- `http.proxyPort`: the port number, the default value being 80.
- `http.nonProxyHosts`: a list of hosts that should be reached directly, bypassing the proxy. This is a list of patterns separated by '|'. The patterns may start or end with a '*' for wildcards. Any host matching one of these patterns will be reached through a direct connection instead of through a proxy.

Let's look at a few examples assuming we're trying to execute the main method of the GetURL class:

```
$ java -Dhttp.proxyHost=webcache.mydomain.com GetURL
```

All http connections will go through the proxy server at `webcache.mydomain.com` listening on port 80 (we didn't specify any port, so the default one is used).

```
$ java -Dhttp.proxyHost=webcache.mydomain.com -Dhttp.proxyPort=8080
-Dhttp.noProxyHosts=”localhost|host.mydomain.com” GetURL
```

In that second example, the proxy server will still be at `webcache.mydomain.com`, but this time listening on port 8080. Also, the proxy won't be used when connecting to either `localhost` or `host.mydonain.com`.

As mentioned earlier, these settings affect all http connections during the entire lifetime of the VM invoked with these options. However it is possible, using the System.setProperty() method, to have a slightly more dynamic behavior.

Here is a code excerpt showing how this can be done:

```
//Set the http proxy to webcache.mydomain.com:8080

System.setProperty("http.proxyHost", "webcache.mydomain.com");
System.setPropery("http.proxyPort", "8080");

// Next connection will be through proxy.
URL url = new URL("http://java.sun.com/");
InputStream in = url.openStream();
```

```
// Now, let's 'unset' the proxy.
System.setProperty("http.proxyHost", null);

// From now on http connections will be done directly.
```

Now,this works reasonably well, even if a bit cumbersome, but it can get tricky if your application is multi-threaded. Remember, system properties are “VM wide” settings, so all threads are affected. Which means that the code in one thread could, as a side effect, render the code in an other thread inoperative.

2.2) HTTPS

The https (http over SSL) protocol handler has its own set of properties:

- htttps.proxyHost
- https.proxyPort

As you probably guessed these work in the exact same manner as their http counterparts, so we won't go into much detail except to mention that the default port number, this time, is 443 and that for the "non proxy hosts" list, the HTTPS protocol handler will use the same as the http handler (i.e. `http.nonProxyHosts`).

2.3) FTP

Settings for the FTP protocol handler follow the same rules as for http, the only difference is that each property name is now prefixed with '`ftp`.' instead of '`http`.'

Therefore the system properties are:

- `ftp.proxHost`
- `ftp.proxyPort`
- `ftp.nonProxyHosts`

Note that, this time, there is a separate property for the "non proxy hosts" list. Also, as for http, the default port number value is 80. It should be noted that when going through a proxy, the FTP protocol handler will actually use HTTP to issue commands to the proxy server, which explains why this is the same default port number.

Let's examine a quick example:

```
$ java -Dhttp.proxyHost=webcache.mydomain.com
-Dhttp.proxyPort=8080 -Dftp.proxyHost=webcache.mydomain.com -Dftp.proxyPort=8080 GetURL
```

Here, both the HTTP and the FTP protocol handlers will use the same proxy server at webcache.mydomain.com:8080.

2.4) SOCKS

The SOCKS protocol, as defined in RFC 1928, provides a framework for client server applications to safely traverse a firewall both at the TCP and UDP level. In that sense it is a lot more generic than higher level proxies (like HTTP or FTP specific proxies). J2SE 5.0 provides SOCKS support for client TCP sockets.

There are 2 system properties related to SOCKS:

- `socksProxyHost` for the host name of the SOCKS proxy server
- `socksProxyPort` for the port number, the default value being 1080

Note that there is no dot ('.') after the prefix this time. This is for historical reasons and to ensure backward compatibility. Once a SOCKS proxy is specified in this manner, all TCP connections will be attempted through the proxy.

Example:

```
$ java -DsocksProxyHost=socks.mydomain.com GetURL
```

Here, during the execution of the code, every outgoing TCP socket will go through the SOCKS proxy server at `socks.mydomain.com:1080`.

Now, what happens when both a SOCKS proxy and a HTTP proxy are defined? Well the rule is that settings for higher level protocols, like HTTP or FTP, take precedence over SOCKS settings. So, in that particular case, when establishing a HTTP connection, the SOCKS proxy settings will be ignored and the HTTP proxy will be contacted. Let's look at an example:

```
$ java -Dhttp.proxyHost=webcache.mydomain.com -Dhttp.proxyPort=8080
-DsocksProxyHost=socks.mydomain.com GetURL
```

Here, an http URL will go through `webcache.mydomain.com:8080` because the http settings take precedence. But what about an ftp URL? Since no specific proxy settings were assigned for FTP, and since FTP is on top of TCP, then FTP connections will be attempted through the SOCKS proxy server at `socks.mydomsain.com:1080`. If an FTP proxy had been specified, then that proxy would have been used instead.

3) Proxy class

As we have seen, the system properties are powerful, but not flexible. The "all or nothing" behavior was justly deemed too severe a limitation by most developers. That's why it was decided to introduce a new, more flexible, API in J2SE 5.0 so that it would be possible to have connection based proxy settings.

The core of this new API is the Proxy class which represents a proxy definition, typically a type (http, socks) and a socket address. There are, as of J2SE 5.0, 3 possible types:

- `DIRECT` which represents a direct connection, or absence of proxy.
- `HTTP` which represents a proxy using the HTTP protocol.
- `SOCKS` which represents proxy using either SOCKS v4 or v5.

So, in order to create an HTTP proxy object you would call:

```
SocketAddress addr = new
InetSocketAddress("webcache.mydomain.com", 8080);
Proxy proxy = new Proxy(Proxy.Type.HTTP, addr);
```

Remember, this new proxy object represents a proxy **definition**, nothing more. How do we use such an object? A new `openConnection()` method has been added to the URL class and takes a Proxy as an argument, it works the same way as `openConnection()` with no arguments, except it forces the connection to be established through the specified proxy, ignoring all other settings, including the system properties mentioned above.

So completing the previous example, we can now add:

```
URL url = new URL("http://java.sun.com/");
URConnection conn = url.openConnection(proxy);
```

Simple, isn't it?

The same mechanism can be used to specify that a particular URL has to be reached directly, because it's on the intranet for example. That's where the DIRECT type comes into play. But, you don't need to create a proxy instance with the DIRECT type, all you have to do is use the NO_PROXY static member:

```
URL url2 = new URL("http://infos.mydomain.com/");
URLConnection conn2 = url2.openConnection(Proxy.NO_PROXY);
```

Now, this guarantees you that this specific URL will be retrieved though a direct connection bypassing any other proxy settings, which can be convenient.

Note that you can force a URLConnection to go through a SOCKS proxy as well:

```
SocketAddress addr = new InetSocketAddress("socks.mydomain.com", 1080);
Proxy proxy = new Proxy(Proxy.Type.SOCKS, addr);
URL url = new URL("ftp://ftp.gnu.org/README");
URLConnection conn = url.openConnection(proxy);
```

That particular FTP connection will be attempted though the specified SOCKS proxy. As you can see, it's pretty straightforward.

Last, but not least, you can also specify a proxy for individual TCP sockets by using the newly introduced socket constructor:

```
SocketAddress addr = new InetSocketAddress("socks.mydomain.com", 1080);
Proxy proxy = new Proxy(Proxy.Type.SOCKS, addr);
Socket socket = new Socket(proxy);
InetSocketAddress dest = new InetSocketAddress("server.foo.com", 1234);
socket.connect(dest);
```

Here the socket will try to connect to its destination address (server.foo.com:1234) through the specified SOCKS proxy.

As for URLs, the same mechanism can be used to ensure that a direct (i.e. not through any proxy) connection should be attempted no matter what the global settings are:

```
Socket socket = new Socket(Proxy.NO_PROXY);
socket.connect(new InetAddress("localhost", 1234));
```

Note that this new constructor, as of J2SE 5.0, accepts only 2 types of proxy: SOCKS or DIRECT (i.e. the NO_PROXY instance).

4) ProxySelector

As you can see, with J2SE 5.0, the developer gains quite a bit of control and flexibility when it comes to proxies. Still, there are situations where one would like to decide which proxy to use dynamically, for instance to do some load balancing between proxies, or depending on the destination, in which case the API described so far would be quite cumbersome. That's where the ProxySelector comes into play.

In a nutshell the ProxySelector is a piece of code that will tell the protocol handlers which proxy to use, if any, for any given URL. For example, consider the following code:

```
URL url = new URL("http://java.sun.com/index.html");
URLConnection conn = url.openConnection();
InputStream in = conn.getInputStream();
```

At that point the HTTP protocol handler is invoked and it will query the proxySelector. The dialog might go something like that:

> **Handler**: Hey dude, I'm trying to reach java.sun.com, should I use a proxy?

ProxySelector: Which protocol do you intend to use?
Handler: http, of course!
ProxySelector: On the default port?
Handler: Let me check.... Yes, default port.
ProxySelector: I see. Then you shall use webcache.mydomain.com on port 8080 as a proxy.
Handler: Thanks. <pause> Dude, webcache.mydomain.com:8080 doesn't seem to be responding! Any other option?
ProxySelector: Dang! OK, try webcache2.mydomain.com, on port 8080 as well.
Handler: Sure. Seems to be working. Thanks.
ProxySelector: No sweat. Bye.

Of course I'm embellishing a bit, but you get the idea.

The best thing about the ProxySelector is that it is plugable! Which means that if you have needs that are not covered by the default one, you can write a replacement for it and plug it in!

So what is a ProxySelector? Let's take a look at the class definition:

```
public abstract class ProxySelector {
        public static ProxySelector getDefault();
        public static void setDefault(ProxySelector ps);
        public abstract List<Proxy> select(URI uri);
        public abstract void connectFailed(URI uri,
                SocketAddress sa, IOException ioe);
}
```

As we can see, ProxySelector is an abstract class with 2 static methods to set, or get, the default implementation, and 2 instance methods that will be used by the protocol handlers to determine which proxy to use or to notify that a proxy seems to be unreachable. If you want to provide your own ProxySelector, all you have to do is extend this class, provide an implementation for these 2 instance methods then call ProxySelector.setDefault() passing an instance of your new class as an argument. At this point the protocol handlers, like http or ftp, will query the new ProxySelector when trying to decide what proxy to use.

Before we see in details how to write such a ProxySelector, let's talk about the default one. J2SE 5.0 provides a default implementation which enforces backward compatibility. In other terms, the default ProxySelector will check the system properties described earlier to determine which proxy to use. However, there is a new, optional feature: On recent Windows systems and on Gnome 2.x platforms it is possible to tell the default ProxySelector to use the system proxy settings (both recent versions of Windows and Gnome 2.x let you set proxies globally through their user interface). If the system property `java.net.useSystemProxies` is set to true (by default it is set to false for compatibility sake), then the default ProxySelector will try to use these settings. You can set that system property on the command line, or you can edit the JRE installation file `lib/net.properties`, that way you have to change it only once on a given system.

Now let's examine how to write, and install, a new ProxySelector.

Here is what we want to achieve: We're pretty happy with the default ProxySelector behavior, except when it comes to http and https. On our network we have more than one possible proxy for these protocols and we we'd like our application to try them in sequence (i.e.: if the 1st one doesn't respond, then try the second one and so on). Even more, if one of them fails too many time, we'll remove it from the list in order to optimize things a bit.

All we need to do is subclass `java.net.ProxySelector` and provide implementations for both the `select()` and `connectFailed()` methods.

The `select()` method is called by the protocol handlers before trying to connect to a destination. The argument passed is a URI describing the resource (protocol, host and port number). The method will then return a List of Proxies. For instance the following code:

```
URL url = new URL("http://java.sun.com/index.html");
InputStream in = url.openStream();
```

will trigger the following pseudo-call in the protocol handler:

```
List<Proxy> l = ProxySelector.getDefault().select(new URI("http://java.sun.com/"));
```

In our implementation, all we'll have to do is check that the protocol from the URI is indeed http (or https), in which case we will return the list of proxies, otherwise we just delegate to the default one. To do that, we'll need, in the constructor, to store a reference to the old default, because ours will become the default.

So it is starting to look like this:

```
public class MyProxySelector extends ProxySelector {
        ProxySelector defsel = null;
        MyProxySelector(ProxySelector def) {
                defsel = def;
        }

        public java.util.List<Proxy> select(URI uri) {
                if (uri == null) {
                        throw new IllegalArgumentException("URI can't be null.");
                }
                String protocol = uri.getScheme();
                if ("http".equalsIgnoreCase(protocol) ||
                        "https".equalsIgnoreCase(protocol)) {
```

```
                        ArrayList<Proxy> l = new ArrayList<Proxy>();
                        // Populate the ArrayList with proxies
                        return l;
                }
                if (defsel != null) {
                        return defsel.select(uri);
                } else {
                        ArrayList<Proxy> l = new ArrayList<Proxy>();
                        l.add(Proxy.NO_PROXY);
                        return l;
                }
        }
}
```

First note the constructor that keeps a reference to the old default selector. Second, notice the check for illegal argument in the select() method in order to respect the specifications. Finally, notice how the code defers to the old default, if there was one, when necessary. Of course, in this example, I didn't detail how to populate the ArrayList, as it not of particular interest, but the complete code is available in the appendix if you're curious.

As it is, the class is incomplete since we didn't provide an implementation for the `connectFailed()` method. That's our very next step.

The `connectFailed()` method is called by the protocol handler whenever it failed to connect to one of the proxies returned by the `select()` method. 3 arguments are passed: the URI the handler was trying to reach, which should be the one used when `select()` was called, the `SocketAddress` of the proxy that the handler was trying to contact and the IOException that was thrown when trying to connect to the proxy. With that information, we'll just do the following: If the proxy is in our list, and it failed 3 times or more, we'll just remove it from our list, making sure it won't be used again in the future. So the code is now:

```
public void connectFailed(URI uri, SocketAddress sa, IOException ioe) {
        if (uri == null || sa == null || ioe == null) {
                throw new IllegalArgumentException("Arguments can't be null.");
        }
        InnerProxy p = proxies.get(sa);
        if (p != null) {
                if (p.failed() >= 3)
                        proxies.remove(sa);
        } else {
                if (defsel != null)
                        defsel.connectFailed(uri, sa, ioe);
        }
}
```

Pretty straightforward isn't it. Again we have to check the validity of the arguments (specifications again). The only thing we do take into account here is the SocketAddress, if it's one of the proxies in our list, then we do deal with it, otherwise we defer, again, to the default selector.

Now that our implementation is, mostly, complete, all we have to do in the application is to register it and we're done:

```
public static void main(String[] args) {
        MyProxySelector ps = new MyProxySelector(ProxySelector.getDefault());
        ProxySelector.setDefault(ps);
        // rest of the application
}
```

Of course, I simplified things a bit for the sake of clarity, in particular you've probably noticed I didn't do much Exception catching, but I'm confident you can fill in the blanks.

It should be noted that both Java Plugin and Java Webstart do replace the default ProxySelector with a custom one to integrate better with the underlying platform or container (like the web browser). So keep in mind, when dealing with ProxySelector, that the default one is typically specific to the underlying platform and to the JVM implementation. That's why it is a good idea, when providing a custom one, to keep a reference to the older one, as we've done in the above example, and use it when necessary.

5) Conclusion

As we have now established J2SE 5.0 provides quite a number of ways to deal with proxies. From the very simple (using the system proxy settings) to the very flexible (changing the ProxySelector, albeit for experienced developers only), including the per connection selection courtesy of the Proxy class.

Appendix

Here is the full source of the ProxySelector we developed in this paper. Keep in mind that this was written for educational purposes only, and was therefore kept pretty simple on purpose.

```
import java.net.*;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.io.IOException;

public class MyProxySelector extends ProxySelector {
        // Keep a reference on the previous default
    ProxySelector defsel = null;
```

```
/*
 * Inner class representing a Proxy and a few extra data
 */
class InnerProxy {
Proxy proxy;
        SocketAddress addr;
        // How many times did we fail to reach this proxy?
        int failedCount = 0;

        InnerProxy(InetSocketAddress a) {
                addr = a;
                proxy = new Proxy(Proxy.Type.HTTP, a);
        }

        SocketAddress address() {
                return addr;
        }

        Proxy toProxy() {
                return proxy;
        }

        int failed() {
                return ++failedCount;
        }
}

/*
 * A list of proxies, indexed by their address.
 */
HashMap<SocketAddress, InnerProxy> proxies = new HashMap<SocketAddress, InnerProxy>();

MyProxySelector(ProxySelector def) {
  // Save the previous default
  defsel = def;

  // Populate the HashMap (List of proxies)
  InnerProxy i = new InnerProxy(new InetSocketAddress("webcache1.mydomain.com", 8080));
  proxies.put(i.address(), i);
  i = new InnerProxy(new InetSocketAddress("webcache2.mydomain.com", 8080));
  proxies.put(i.address(), i);
  i = new InnerProxy(new InetSocketAddress("webcache3.mydomain.com", 8080));
  proxies.put(i.address(), i);
  }

  /*
   * This is the method that the handlers will call.
   * Returns a List of proxy.
   */
  public java.util.List<Proxy> select(URI uri) {
        // Let's stick to the specs.
        if (uri == null) {
                throw new IllegalArgumentException("URI can't be null.");
        }

        /*
         * If it's a http (or https) URL, then we use our own
         * list.
         */
        String protocol = uri.getScheme();
        if ("http".equalsIgnoreCase(protocol) ||
                "https".equalsIgnoreCase(protocol)) {
                ArrayList<Proxy> l = new ArrayList<Proxy>();
                for (InnerProxy p : proxies.values()) {
                  l.add(p.toProxy());
                }
                return l;
        }

        /*
         * Not HTTP or HTTPS (could be SOCKS or FTP)
         * defer to the default selector.
         */
        if (defsel != null) {
                return defsel.select(uri);
        } else {
                ArrayList<Proxy> l = new ArrayList<Proxy>();
                l.add(Proxy.NO_PROXY);
                return l;
```

```
                }
        }

        /*
         * Method called by the handlers when it failed to connect
         * to one of the proxies returned by select().
         */
        public void connectFailed(URI uri, SocketAddress sa, IOException ioe) {
                // Let's stick to the specs again.
                if (uri == null || sa == null || ioe == null) {
                        throw new IllegalArgumentException("Arguments can't be null.");
                }

                /*
                 * Let's lookup for the proxy
                 */
                InnerProxy p = proxies.get(sa);
                        if (p != null) {
                                /*
                                 * It's one of ours, if it failed more than 3 times
                                 * let's remove it from the list.
                                 */
                                if (p.failed() >= 3)
                                        proxies.remove(sa);
                        } else {
                                /*
                                 * Not one of ours, let's delegate to the default.
                                 */
                                if (defsel != null)
                                  defsel.connectFailed(uri, sa, ioe);
                        }
    }
}
```

Feedback to:jean-christophe.collet@sun.com