

Sample Code for Dual Stack Applications Using Non-Blocking Sockets

Lawrence E. Hughes

CTO, InfoWeapons Corp / DualStak Networks Sdn. Bhd.

There are several sources of sample code for making connections from a dual stack client, and additional ones for accepting connections on a dual stack server. Many of these are suboptimal for various reasons, but a lot of current applications have used the code (or at least the techniques illustrated by them) from these samples. Because of this, some major content providers (e.g. Google) are unwilling to have their primary URL (e.g. www.google.com) publish both IPv4 and IPv6 records in DNS. This has had a chilling effect on adoption and rollout of IPv6 (it “sends the wrong signals” to other providers).

Dual Stack Client Issues

In a traditional “single stack” client (IPv4-only or IPv6-only), you typically use DNS to translate a Fully Qualified Domain Name (e.g. www.google.com) into a *single* IP address. You connect to the only address returned. Everything works fine.

In some cases (e.g. with www.google.com) you might get back *several* IP addresses (in that case, all IPv4). Some sites publish multiple IPv4 addresses to do a simple form of *load balancing* (distributing the overall load among many servers). Most DNS servers will rotate the configured IPv4 addresses in *round robin* fashion. Existing clients will use the first one returned, so in general, if there are n published addresses, $1/n$ of the clients will use each address.

In a dual stack client (IPv4 and IPv6), when you query DNS, you may get back only one IPv4 address, only one IPv6 address, or if the server is dual stack, you may get back *both* an IPv4 and an IPv6 address (or possibly even more than one of each). The current code samples (assuming IPv6 connectivity is available) will try IPv6 first, and if that doesn’t work, fail back to IPv4. In the simple case of one IPv6 and one IPv4 address, the client will first try the IPv6 address. If it works, all is fine. If for some reason that fails, there may be a 30 to 70 second timeout, then it will try the IPv4 address, which usually works. The user doesn’t understand that the fault lies in his client software and thinks there is a problem with the content provider’s website (and then complain to the content provider or worse yet, use another content provider).

Existing code samples connect *serially* to the addresses returned from DNS, as described above. A better approach is to connect in *parallel* to at least one IPv4 and one IPv6 address (optionally to all addresses returned from DNS) simultaneously. This involves obtaining the list of all addresses published for a given Fully Qualified Domain Name, and using *non-blocking connects* together with a scheme to determine when connections have succeeded. If it doesn’t matter which IP version is used (IPv4 or IPv6), then you can simply accept the first connection. If IPv6 is preferred, you can wait for a working IPv6 connection for some number of milliseconds, and accept the first one that happens. If no IPv6 connection succeeds within that time, chances are an IPv4 connection has already succeeded, which can then immediately be used. All other connections can then be discarded.

The net effect is that even in a mis-configured network, client applications using this approach will not experience long timeouts that will annoy users. If even a short “try for IPv6” time is provided, most of the time, an IPv6 connection will be obtained. The longer you are willing to wait for an IPv6 connection,

the more likely one is to work. Once most client software uses this approach, content providers should be more willing to publish both IPv4 and IPv6 addresses for their main URLs in DNS.

Normal socket operations work in *blocking* mode. In blocking mode, when you call a system routine (e.g. connect), execution stops until that operation is complete, at which point it will resume execution with the statement following the system call. This is simple to code, but does not allow concurrency (doing multiple things at the same time). In *non-blocking* mode, execution does not stop in such a system call – it starts the operation and resumes execution immediately. That means you can do other things while waiting for it to complete, but at some point you must check for completion before you can use the result of that system call (e.g. before you can use the connection). This is rather more complicated to code, but supports concurrency. Both of the techniques presented here depend on using the TCP/IP socket system calls in non-blocking mode, at least during part of the process, so that multiple things can be “kept in the air” at the same time, like a juggler’s balls.

The parallel approach is rather more complex than the serial approach, but I have created not only an example of how to do this, but an entire royalty free library for you to use as is, or to modify as needed, in C. This provides buffered I/O with getline/putline capability, and both serial and parallel versions of the connect code. All the tricky code with non-blocking sockets is hidden from users of the library – by the time the connection is made, sockets are in normal blocking mode. A simple test program is included to demonstrate connection to a dual stack email server, with simple interaction (display greeting, process EHLO and QUIT commands, then close the connection).

This code has been tested on both FreeBSD and Ubuntu Linux. I would appreciate anyone that tests it on or modifies it for other platforms to share their results with me, and if possible, release their code for posting on my site.

Many people have talked about the “long timeout” problem, and why it is slowing support of IPv6 by content providers. This code is InfoWeapons’ contribution to helping make World IPv6 Day work better.

Dual Stack Server Issues

The use of non-blocking sockets can also be applied to servers. Previously, many servers (such as Apache 2.2) have implemented dual stack support through use of IPv6 sockets that use “IPv4-mapped IPv6 addresses” (e.g. `::ffff:123.45.67.89`). You can recognize such servers by use of the “netstat -na” command on most platforms (e.g. FreeBSD or Linux). An IPv4 stream socket listening only on IPv4 is indicated with “tcp4”. An IPv6 stream socket listening only on IPv6 is indicated with “tcp6”. An IPv6 stream socket that accepts connections on either IPv4 or IPv6 through use of “IPv4 mapped IPv6 addresses” is indicated by “tcp46”. If you have either two instances of an application, one listening only on IPv4, and one listening only over IPv6, the same port (e.g. 22 for SSH) is listed twice, once with “tcp4” and once with “tcp6”. It is also possible to create a single process that creates *two* listening sockets, one for IPv4-only and one for IPv6-only, and will accept connections over either of them. Again, the port listed twice, once with “tcp4” and once with “tcp6”, but only once server process is running.

There are numerous problems with IPv4-mapped IPv6 addresses, and these have been deprecated. This approach should not be used in new code, and should be updated in any existing code.

Note that there is a significant difference between FreeBSD and Linux when working with IPv6 sockets. By default, IPv6 sockets in Linux support “IPv4 mapped IPv6 addresses” which allows you to use a single IPv6 socket for both IPv6 and IPv4 connections. In comparison, by default FreeBSD does not enable “IPv4 mapped IPv6 addresses” on IPv6 sockets. So, the server code forces the use of “IPv6_V6ONLY” mode in IPv6 sockets (no “IPv4 mapped IPv6 addresses”). This is not needed in FreeBSD (it is already disabled), but without this option, the Linux version will fail (when you try to listen with the IPv6 socket, it will complain that it is already listening to that port – it is trying to listen to the specified port over both IPv4 and IPv6 with the IPv6 socket, and we already are listening on IPv4 with the IPv4 socket). As written, this code will work without modification on both FreeBSD 8.2 and Ubuntu Linux 10.10.

The normal use of blocking sockets makes it difficult to listen for a connection on either of two sockets. One approach would be to spawn two *parent* processes, one with an IPv4-only listening socket and one with an IPv6-only listening socket, but this is inefficient. A better approach can be created again through use of non-blocking sockets, in combination with the `select()` system call. You create two sockets in non-blocking mode, one listening for connections on IPv6, and one listening for connections on IPv4. The `select()` call will block until a connection happens on either or both of them, at which point the child socket can be made blocking, and either a child process or thread can be created to handle the connection(s), as usual.

Again, a demo application is created to illustrate this approach, once again using the SCIO library from the client example. This application is a simple process model server that accepts connections from telnet clients (port 23) over IPv4 or IPv6. The connection handler sends back the address from which the client connected, then terminates the connection. This is a dual stack “telnet autoresponder”, as deployed on v6address.com and v4address.com. This is similar to sites like “whatismyipaddress.com” for web browsers, but allows seeing what address is being used for outgoing connections from a text based interface (e.g. FreeBSD “terminal”, or Windows “command prompt”), by use of telnet. It is not meant to illustrate the best way to create concurrent servers, e.g. using threads or even pre-created processes or threads. It uses a simple process per connection model to make it more obvious how the non-blocking acceptance of connections works. This code creates the sockets and handles acceptance of connections outside of the SCIO library, then sets up the StreamCon data structure to allow the connection handler to use the SCIO `getline()` / `putline()` routines normally.

Again, this code has been tested on both FreeBSD and Linux. You are welcome to use it as is, modify it in any way you wish and even include it in commercial applications. If you do port it to other platforms, I would appreciate your feedback, and if possible, release of the ported code to publish on my site for others, on the same terms.

This code is not intended to show the optimal way to handle large numbers of connections. It spawns a process for each incoming connection, which works, but is not efficient. Using a thread for each connection is more efficient, especially with “pre-created” threads.

We should try to eliminate use of “IPv4-mapped IPv6 addresses” in dual-stack servers for a number of reasons. This code hopefully will help in this process.

Note that connection handler routines can easily determine whether the connection was over IPv4 or IPv6, and what address they connected from. This can be used in various creative ways. In telnet it simply reports to the user (via telnet) what those are. You could provide different functionality based on where that IP address is located geographically (databases exist for both IPv4 and IPv6). You could also provide

more sophisticated functionality (e.g. IPsec, multicast, end-to-end connections) over IPv6, but simpler functionality over IPv4.