12.5 Source Code Portability

Most existing network applications are written assuming IPv4. `sockaddr_in` structures are allocated and filled in and the calls to `socket` specify `AF_INET` as the first argument. We saw in the conversion from Figure 1.5 to Figure 1.6 that these IPv4 applications could be converted to use IPv6 without too much effort. Many of the changes that we showed could be done automatically using some editing scripts. Programs that are more dependent on IPv4, using features such as multicasting, IP options, or raw sockets, will take more work to convert.

If we convert an application to use IPv6 and distribute it in source code, we now have to worry about whether or not the recipient's system supports IPv6. The typical way to handle this is with `#ifdefs` throughout the code, using IPv6 when possible (since we have seen in this chapter that an IPv6 client can still communicate with IPv4 servers, and vice versa). The problem with this approach is that the code becomes littered with `#ifdefs` very quickly, and is harder to follow and maintain.

A better approach is to consider the move to IPv6 as a chance to make the program protocol-independent. The first step is to remove calls to `gethostbyname` and `gethostbyaddr` and use the `getaddrinfo` and `getnameinfo` functions that we described in the previous chapter. This lets us deal with socket address structures as opaque objects, referenced by a pointer and size, which is exactly what the basic socket functions do: `bind`, `connect`, `recvfrom`, and so on. Our `sock_`*XXX* functions from Section 3.8 can help manipulate these, independent of IPv4 or IPv6. Obviously these functions contain `#ifdefs` to handle IPv4 and IPv6, but hiding all of this protocol dependency in a few library functions makes our code simpler. We will develop a set of `mcast_`*XXX* functions in Section 21.7 that can make multicast applications independent of IPv4 or IPv6.

Another point to consider is what happens if we compile our source code on a system that supports both IPv4 and IPv6, distribute either executable code or object files (but not the source code), and someone runs our application on a system that does not support IPv6? There is a chance that the local name server supports AAAA records and returns both AAAA records and A records for some peer with which our application tries to connect. If our application, which is IPv6-capable, calls `socket` to create an IPv6 socket, it will fail if the host does not support IPv6. We handle this in the helper functions described in the previous chapter by ignoring the error from `socket` and trying the next address on the list returned by the name server. Assuming the peer has an A record, and that the name server returns the A record in addition to any AAAA records, the creation of an IPv4 socket will succeed. This is the type of functionality that belongs in a library function, and not in the source code of every application.

To enable passing socket descriptors to programs that were IPv4-only or IPv6-only, RFC 2133 [Gilligan et al. 1997] introduced the `IPV6_ADDRFORM` socket option, which could return or potentially change the address family associated with a socket. However, the semantics were never completely described, and it was only useful in very specific cases, so it was removed in the next revision of the API.