chapter 3

Of Names and Address Families

At this point, you know enough to build working TCP clients and servers. However, our examples so far, though useful enough, nevertheless have a couple of features that could be improved. First, *the only way to specify a destination is with an IP address*, such as 169.1.1.1 or FE80:1034::2A97:1001:1. This is a bit painful for most humans, who are—let's face it—not that good at dealing with long strings of numbers that have to be formatted just right. That's why most applications allow the use of *names* like www.mkp.com and server.example.com to specify destinations, in addition to Internet addresses. But the Sockets API, as we've seen, *only* takes numerical arguments, so applications need a way to convert names to the required numerical form.

Another problem with our examples so far is that *the choice of whether to use IPv4 or IPv6 is wired into the code*—each progam we've seen deals with only *one* version of the IP protocol. That was by design—to keep things simple. But wouldn't it be better if we could *hide* this choice from the rest of the code, letting the argument(s) determine whether a socket for IPv4 or IPv6 is created?

It turns out that the API provides solutions to both of these problems—and more! In this chapter we'll see how to (1) access the *name service* to convert between names and numeric quantities; and (2) write code that chooses between IPv4 and IPv6 at runtime.

3.1 Mapping Names to Numbers

Identifying endpoints with strings of dot- or colon-separated numbers is not very user friendly, but that's not the only reason to prefer names over addresses. Another is that a host's Internet

address is tied to the part of the network to which it is connected. This is a source of inflexibility: If a host moves to another network or changes Internet service providers (ISPs), its Internet address generally has to change. Then everybody who refers to the host by that address has to be informed of the change, or they won't be able to access the host! When this book was written, the Web server for the publisher of this text, Morgan Kaufmann, had an Internet address of 129.35.69.7. However, we invariably refer to that Web server as *www.mkp.com*. Obviously, *www.mkp.com* is easier to remember than 129.35.69.7. In fact, this is most likely how you typically think of specifying a host on the Internet, by name. In addition, if Morgan Kaufmann's Web server changes its Internet address for some reason (e.g., new ISP, server moves to another machine), simply changing the mapping of *www.mkp.com* from 129.35.69.7 to the new Internet address allows the change to be transparent to all programs that use the name to identify the Web server.[1]

To solve these problems, most implementations of the Sockets API provide access to a *name service* that maps names to other information, including Internet addresses. You've already seen names that map to Internet addresses (*www.mkp.com*). Names for services (e.g., echo) can also be mapped to port numbers. The process of mapping a name to a numeric quantity (address or port number) is called *resolution*. There are a number of ways to resolve names into binary quantities; your system probably provides access to several of these. Some of them involve interaction with other systems "under the covers"; others are strictly local.

It is critical to remember that **a name service is not required for TCP/IP to work**. Names simply provide a level of indirection, for the reasons discussed above. The host-naming service can access information from a wide variety of sources. Two of the primary sources are the *Domain Name System* (DNS) and local configuration databases. The DNS [8] is a distributed database that maps *domain names* such as *www.mkp.com* to Internet addresses and other information; the DNS protocol [9] allows hosts connected to the Internet to retrieve information from that database using TCP or UDP. Local configuration databases are generally operating-system-specific mechanisms for name-to-Internet-address mappings. Fortunately for the programmer, the details of how the name service is *implemented* are hidden behind the API, so the only thing we need to know is how to ask it to *resolve* a name.

3.1.1 Accessing the Name Service

The preferred interface to the name service interface is through the function `getaddrinfo()`:[2]

---

int `getaddrinfo` (**const char** *`hostStr`, **const char** *`serviceStr`,
const struct addrinfo *`hints`, **struct addrinfo** **`results`)

---

[1] MK's address was actually 208.164.121.48 when we wrote the first edition. Presumably, they changed their address to help us make this point.

[2] Historically, other functions were available for this purpose, and many applications still use them. However, they have several shortcomings and are considered obsolescent as of the POSIX 2001 standard.

The first two arguments to `getaddrinfo()` point to null-terminated character strings representing a host name or address and a service name or port number, respectively. The third argument describes the kind of information to be returned; we discuss it below. The last argument is the location of a **struct addrinfo** pointer, where a pointer to a linked list containing results will be stored. The return value of `getaddrinfo()` indicates whether the resolution was successful (0) or unsuccessful (nonzero error code).

Using `getaddrinfo()` entails using two other auxiliary functions:

---

void `freeaddrinfo(`**struct addrinfo** *`addrList`*`)`
const char `*gai_strerror(`**int** *`errorCode`*`)`

`getaddrinfo()` creates a dynamically allocated linked list of results, which must be deallocated after the caller is finished with the list. Given the pointer to the head of the result list, `freeaddrinfo()` frees all the storage allocated for the list. Failure to call this method can result in a pernicious memory leak. The method **should only be called when the program is finished with the returned information**; no information contained in the list of results is reliable after this function has returned. In case `getaddrinfo()` returns a nonzero (error) value, passing it to `gai_strerror()` yields a string that describes what went wrong.

Generally speaking, `getaddrinfo()` takes the name of a host/service pair as input and returns a linked list of structures containing everything needed to create a socket to connect to the named host/service, including: address/protocol family (v4 or v6), socket type (e.g., stream or datagram), protocol (TCP or UDP for the Internet protocol family), and numeric socket address. Each entry in the linked list is placed into an **addrinfo** structure, declared as follows:

```
struct addrinfo {
  int ai_flags;             // Flags to control info resolution
  int ai_family;            // Family:  AF_INET, AF_INET6, AF_UNSPEC
  int ai_socktype;          // Socket type:  SOCK_STREAM, SOCK_DGRAM
  int ai_protocol;          // Protocol: 0 (default) or IPPROTO_XXX
  socklen_t ai_addrlen;     // Length of socket address ai_addr
  struct sockaddr *ai_addr; // Socket address for socket
  char *ai_canonname;       // Canonical name
  struct addrinfo *ai_next; // Next addrinfo in linked list
};
```

The **ai_sockaddr** field contains a **sockaddr** of the appropriate type, with (numeric) address and port information filled in. It should be obvious which fields contain the address family, socket type, and protocol information. (The flags field is not used in the result; we will discuss its use shortly.) Actually, the results are returned in a pointer to a linked list of **addrinfo** structures; the *ai_next* field contains the pointers for this list.

Why a linked list? There are two reasons. First, for each combination of host and service, there might be several different combinations of address family (v4 or v6) and socket

type/protocol (stream/TCP or datagramUDP) that represent possible endpoints. For example, the host "server.example.net" might have instances of the "spam" service listening on port 1001 on both IPv4/TCP and IPv6/UDP. The `getaddrinfo()` function returns both of these. The second reason is that a hostname can map to multiple IP addresses; `getaddrinfo()` helpfully returns all of these.

Thus, `getaddrinfo()` returns all the viable combinations for a given hostname, service pair. But wait—what if you don't *need* options, and you know exactly what you want in advance? You don't want to have to write code that searches through the returned list for a particular combination—say, IPv4/TCP. That's where the third parameter of `getaddrinfo()` comes in! It allows you to tell the system to *filter the results for you.* We'll see how it is used in our example program, `GetAddrInfo.c`.

GetAddrInfo.c

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <netdb.h>
 5  #include "Practical.h"
 6
 7  int main(int argc, char *argv[]) {
 8
 9    if (argc != 3) // Test for correct number of arguments
10      DieWithUserMessage("Parameter(s)", "<Address/Name> <Port/Service>");
11
12    char *addrString = argv[1];   // Server address/name
13    char *portString = argv[2];   // Server port/service
14
15    // Tell the system what kind(s) of address info we want
16    struct addrinfo addrCriteria;                   // Criteria for address match
17    memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
18    addrCriteria.ai_family = AF_UNSPEC;             // Any address family
19    addrCriteria.ai_socktype = SOCK_STREAM;         // Only stream sockets
20    addrCriteria.ai_protocol = IPPROTO_TCP;         // Only TCP protocol
21
22    // Get address(es) associated with the specified name/service
23    struct addrinfo *addrList; // Holder for list of addresses returned
24    // Modify servAddr contents to reference linked list of addresses
25    int rtnVal = getaddrinfo(addrString, portString, &addrCriteria, &addrList);
26    if (rtnVal != 0)
27      DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
28
29    // Display returned addresses
```

```
30    for (struct addrinfo *addr = addrList; addr != NULL; addr = addr->ai_next) {
31      PrintSocketAddress(addr->ai_addr, stdout);
32      fputc('\n', stdout);
33    }
34
35    freeaddrinfo(addrList); // Free addrinfo allocated in getaddrinfo()
36
37    exit(0);
38  }
```

GetAddrInfo.c

1. **Application setup and parameter parsing:** lines 9–13
2. **Construct address specification:** lines 15–20
   The *addrCriteria* structure will indicate what kinds of results we are interested in.
   - **Declare and initialize addrinfo structure:** lines 16–17
   - **Set address family:** line 18
     We set the family to AF_UNSPEC, which allows the returned address to come from any family (including AF_INET and AF_INET6).
   - **Set socket type:** line 19
     We want a stream/TCP endpoint, so we set this to SOCK_STREAM. The system will filter out results that use different protocols.
   - **Set protocol:** line 20
     We want a TCP socket, so we set this to IPPROTO_TCP. Since TCP is the default protocol for stream sockets, leaving this field 0 would have the same result.
3. **Fetch address information:** lines 22–27
   - **Declare pointer for head of result linked list:** line 23
   - **Call getaddrinfo():** line 25
     We pass the desired hostname, port, and the constraints encoded in the *addrCriteria* structure.
   - **Check return value:** lines 26–27
     `getaddrinfo()` returns 0 if successful. Otherwise, the return value indicates the specific error. The auxiliary function `gai_strerror()` returns a character string error message explaining the given error return value. Note that these messages are different from the normal *errno*-based messages.
4. **Print addresses:** lines 29–33
   Iterate over the linked list of addresses, printing each to the console. The function `PrintSocketAddress()` takes an address to print and the stream on which to print. We present its code, which is in `AddressUtility.c`, later in this chapter.

5. **Free address linked list:** line 35
   The system allocated storage for the linked list of **addrinfo** structures it returned. We must call the auxiliary function `freeaddrinfo()` to free that memory when we are finished with it.

The program `GetAddrInfo.c` takes two command-line parameters, a hostname (or address) and a service name (or port number), and prints the address information returned by `getaddrinfo()`. Suppose you want to find an address for the service named "whois" on the host named "localhost" (i.e., the one you are running on). Here's how you use it:

```
% GetAddrInfo localhost whois
127.0.0.1-43
```

To find the service "whois" on the host "pine.netlab.uky.edu", do this:

```
% GetAddrInfo pine.uky.edu whois
128.163.170.219-43
```

The program can deal with any combination of name and numerical arguments:

```
% GetAddrInfo 169.1.1.100 time
169.1.1.100-37
% GetAddrInfo FE80:0000:0000:0000:0000:ABCD:0001:0002:0003 12345
fe80::abcd:1:2:3-12345
```

These examples all return a single answer. But as we noted above, some names have multiple numeric addresses associated with them. For example, "google.com" is typically associated with a number of Internet addresses. This allows a service (e.g., search engine) to be placed on multiple hosts. Why do this? One reason is robustness. If any single host fails, the service continues because the client can use any of the hosts providing the service. Another advantage is scalability. If clients randomly select the numeric address (and corresponding host) to use, we can spread the load over multiple servers. The good news is that `getaddrinfo()` returns *all* of the addresses to which a name maps. You can experiment with this by executing the program with the names of popular Web sites. (Note that supplying 0 for the second argument results in only the address information being printed.)

3.1.2 Details, Details

As we noted above, `getaddrinfo()` is something of a "Swiss Army Knife" function. We'll cover some of the subtleties of its capabilities here. Beginning readers may wish to skip this section and come back to it later.

The third argument (**addrinfo** structure) tells the system what kinds of endpoints the caller is interested in. In `GetAddrInfo.c`, we set this parameter to indicate that any address family was acceptable, and we wanted a stream/TCP socket. We could have instead specified

datagram/UDP or a particular address family (say, AF_INET6), or we could have set the *ai_socktype* and *ai_protocol* fields to zero, indicating that we wanted to receive *all* possibilities. It is even possible to pass a NULL pointer for the third argument; the system is supposed to treat this case as if an **addrinfo** structure had been passed with *ai_family* set to AF_UNSPEC and everything else set to 0.

The *ai_flags* field in the third parameter provides additional control over the behavior of `getaddrinfo()`. It is an integer, individual bits of which are interpreted as boolean variables by the system. The meaning of each flag is given below; flags can be combined using the bitwise OR operator "|" (see Section 5.1.8 for how to do this).

AI_PASSIVE If *hostStr* is NULL when this flag is set, any returned **addrinfo**s will have their addresses set to the appropriate "any" address constant—INADDR_ANY (IPv4) or IN6ADDR_ANY_INIT (IPv6).

AI_CANONNAME Just as one name can resolve to many numeric addresses, multiple names can resolve to the same IP address. However, one name is usually defined to be the official ("canonical") name. By setting this flag in *ai_flags*, we instruct `getaddrinfo()` to return a pointer to the canonical name (if it exists) in the **ai_canonname** field of the first **struct addrinfo** of the linked list.

AI_NUMERICHOST This flag causes an error to be returned if *hostStr* does not point to a string in valid numeric address format. Without this flag, if the *hostStr* parameter points to something that is not a valid string representation of a numeric address, an attempt will be made to resolve it via the name system; this can waste time and bandwidth on useless queries to the name service. If this flag is set, a given valid address string is simply converted and returned, a la `inet_pton()`.

AI_ADDRCONFIG If set, `getaddrinfo()` returns addresses of a particular family only if the system has an interface configured for that family. So an IPv4 address would be returned only if the system has an interface with an IPv4 address, and similarly for IPv6.

AI_V4MAPPED If the *ai_family* field contains AF_INET6, and no matching IPv6 addresses are found, then `getaddrinfo()` returns IPv4-mapped IPv6 addresses. This technique can be used to provide limited interoperation between IPv4-only and IPv6 hosts.

3.2 Writing Address-Generic Code

A famous bard once wrote "To v6 or not to v6, that is the question." Fortunately, the Socket interface allows us to postpone answering that question until execution time. In our earlier TCP client and server examples, we specified a particular IP protocol version to both the socket creation and address string conversion functions using AF_INET or AF_INET6. However, `getaddrinfo()` allows us to write code that works with either address family, without having to duplicate steps for each version. In this section we'll use its capabilities to modify our version-specific client and server code to make them generic.

Before we do that, here's a handy little method that prints a socket address (of either flavor). Given a **sockaddr** structure containing an IPv4 or IPv6 address, it prints the address to the given output stream, using the proper format for its address family. Given any other kind of address, it prints an error string. This function takes a generic **struct sockaddr** pointer and prints the address to the specified stream. You can find our implementation of `PrintSocketAddr()` in `AddressUtility.c` with the function prototype included in `Practical.h`.

PrintSocketAddr()

```
 1  void PrintSocketAddress(const struct sockaddr *address, FILE *stream) {
 2    // Test for address and stream
 3    if (address == NULL || stream == NULL)
 4      return;
 5
 6    void *numericAddress; // Pointer to binary address
 7    // Buffer to contain result (IPv6 sufficient to hold IPv4)
 8    char addrBuffer[INET6_ADDRSTRLEN];
 9    in_port_t port; // Port to print
10    // Set pointer to address based on address family
11    switch (address->sa_family) {
12    case AF_INET:
13      numericAddress = &((struct sockaddr_in *) address)->sin_addr;
14      port = ntohs(((struct sockaddr_in *) address)->sin_port);
15      break;
16    case AF_INET6:
17      numericAddress = &((struct sockaddr_in6 *) address)->sin6_addr;
18      port = ntohs(((struct sockaddr_in6 *) address)->sin6_port);
19      break;
20    default:
21      fputs("[unknown type]", stream);    // Unhandled type
22      return;
23    }
24    // Convert binary to printable address
25    if (inet_ntop(address->sa_family, numericAddress, addrBuffer,
26        sizeof(addrBuffer)) == NULL)
27      fputs("[invalid address]", stream); // Unable to convert
28    else {
29      fprintf(stream, "%s", addrBuffer);
30      if (port != 0)                // Zero not valid in any socket addr
31        fprintf(stream, "-%u", port);
32    }
33  }
```

3.2.1 Generic TCP Client

Using `getaddrinfo()`, we can write clients and servers that are not specific to one IP version or the other. Let's begin by converting our TCP client to make it version-independent; we'll drop the version number and call it `TCPEchoClient.c`. The general strategy is to set up arguments to `getaddrinfo()` that make it return both IPv4 and IPv6 addresses and use the first address that works. Since our address search functionality may be useful elsewhere, we factor out the code responsible for creating and connecting the client socket, placing it in a separate function, `SetupTCPClientSocket()`, in TCPClientUtility.c. The setup function takes a host and service, specified in a string, and returns a connected socket (or -1 on failure). The host or service may be specified as NULL.

TCPClientUtility.c

```
 1  #include <string.h>
 2  #include <unistd.h>
 3  #include <sys/types.h>
 4  #include <sys/socket.h>
 5  #include <netdb.h>
 6  #include "Practical.h"
 7
 8  int SetupTCPClientSocket(const char *host, const char *service) {
 9    // Tell the system what kind(s) of address info we want
10    struct addrinfo addrCriteria;                   // Criteria for address match
11    memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
12    addrCriteria.ai_family = AF_UNSPEC;             // v4 or v6 is OK
13    addrCriteria.ai_socktype = SOCK_STREAM;         // Only streaming sockets
14    addrCriteria.ai_protocol = IPPROTO_TCP;         // Only TCP protocol
15
16    // Get address(es)
17    struct addrinfo *servAddr; // Holder for returned list of server addrs
18    int rtnVal = getaddrinfo(host, service, &addrCriteria, &servAddr);
19    if (rtnVal != 0)
20      DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
21
22    int sock = -1;
23    for (struct addrinfo *addr = servAddr; addr != NULL; addr = addr->ai_next) {
24      // Create a reliable, stream socket using TCP
25      sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
26      if (sock < 0)
27        continue;  // Socket creation failed; try next address
28
29      // Establish the connection to the echo server
30      if (connect(sock, addr->ai_addr, addr->ai_addrlen) == 0)
```

```
31        break;     // Socket connection succeeded; break and return socket
32
33      close(sock); // Socket connection failed; try next address
34      sock = -1;
35    }
36
37    freeaddrinfo(servAddr); // Free addrinfo allocated in getaddrinfo()
38    return sock;
39  }
```

TCPClientUtility.c

1. **Resolve the host and service:** lines 10–20
   The criteria we pass to `getaddrinfo()` specifies that we don't care which protocol is used (AF_UNSPEC), but the socket address is for TCP (SOCK_STREAM/IPPROTO_TCP).
2. **Attempt to create and connect a socket from the list of addresses:** lines 22–35
   - **Create appropriate socket type:** lines 25–27
     `getaddrinfo()` returns the matching domain (AF_INET or AF_INET6) and socket type/protocol. We pass this information on to `socket()` when creating the new socket. If the system cannot create a socket of the specified type, we move on to the next address.
   - **Connect to specified server:** lines 30–34
     We use the address obtained from `getaddrinfo()` to attempt to connect to the server. If the connection succeeds, we exit the address search loop. If the connection fails, we close the socket and try the next address.
3. **Free address list:** line 37
   To avoid a memory leak, we need to free the address linked list created by `getaddrinfo()`.
4. **Return resulting socket descriptor:** line 38
   If we succeed in creating and connecting a socket, return the socket descriptor. If no addresses succeeded, return −1.

Now we are ready to see the generic client.

TCPEchoClient.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <netdb.h>
8  #include "Practical.h"
```

```
 9
10  int main(int argc, char *argv[]) {
11
12    if (argc < 3 || argc > 4) // Test for correct number of arguments
13      DieWithUserMessage("Parameter(s)",
14          "<Server Address/Name> <Echo Word> [<Server Port/Service>]");
15
16    char *server = argv[1];     // First arg: server address/name
17    char *echoString = argv[2]; // Second arg: string to echo
18    // Third arg (optional): server port/service
19    char *service = (argc == 4) ? argv[3] : "echo";
20
21    // Create a connected TCP socket
22    int sock = SetupTCPClientSocket(server, service);
23    if (sock < 0)
24      DieWithUserMessage("SetupTCPClientSocket() failed", "unable to connect");
25
26    size_t echoStringLen = strlen(echoString); // Determine input length
27
28    // Send the string to the server
29    ssize_t numBytes = send(sock, echoString, echoStringLen, 0);
30    if (numBytes < 0)
31      DieWithSystemMessage("send() failed");
32    else if (numBytes != echoStringLen)
33      DieWithUserMessage("send()", "sent unexpected number of bytes");
34
35    // Receive the same string back from the server
36    unsigned int totalBytesRcvd = 0; // Count of total bytes received
37    fputs("Received: ", stdout);     // Setup to print the echoed string
38    while (totalBytesRcvd < echoStringLen) {
39      char buffer[BUFSIZE]; // I/O buffer
40      // Receive up to the buffer size (minus 1 to leave space for
41      // a null terminator) bytes from the sender
42      numBytes = recv(sock, buffer, BUFSIZE - 1, 0);
43      if (numBytes < 0)
44        DieWithSystemMessage("recv() failed");
45      else if (numBytes == 0)
46        DieWithUserMessage("recv()", "connection closed prematurely");
47      totalBytesRcvd += numBytes; // Keep tally of total bytes
48      buffer[numBytes] = '\0';    // Terminate the string!
49      fputs(buffer, stdout);      // Print the buffer
50    }
51
52    fputc('\n', stdout); // Print a final linefeed
53
```

```
54    close(sock);
55    exit(0);
56  }
```

TCPEchoClient.c

After socket creation, the remainder of TCPEchoClient.c is identical to the version-specific clients. There is one caveat that must be mentioned with respect to this code. In line 25 of SetupTCPClientSocket(), we pass the *ai_family* field of the returned **addrinfo** structure as the first argument to socket(). Strictly speaking, this value identifies an *address family* (AF_XXX, whereas the first argument of socket indicates the desired *protocol family* of the socket (PF_XXX). In all implementations with which we have experience, these two families are interchangeable—in particular AF_INET and PF_INET are defined to have the same value, as are PF_INET6 and AF_INET6. **Our generic code depends on this fact.** The authors contend that these definitions will not change, but feel that full disclosure of this assumption (which allows more concise code) is important. Elimination of this assumption is straightforward enough to be left as an exercise.

3.2.2 Generic TCP Server

Our protocol-independent TCP echo server uses similar adaptations to those in the client. Recall that the typical server binds to *any* available local address. To accomplish this, we (1) specify the AI_PASSIVE flag and (2) specify NULL for the hostname. Effectively, this gets an address suitable for passing to bind(), including a wildcard for the local IP address—INADDR_ANY for IPv4 or IN6ADDR_ANY_INIT for IPv6. For systems that support *both* IPv4 and IPv6, IPv6 will generally be returned first by getaddrinfo() because it offers more options for interoperability. Note, however, that the problem of which options should be selected to maximize connectivity depends on the particulars of the environment in which the server operates—from its name service to its Internet Service Provider. **The approach we present here is essentially the simplest possible, and is likely not adequate for production servers that need to operate across a wide variety of platforms.** See the next section for additional information.

As in our protocol-independent client, we've factored the steps involved in establishing a socket into a separate function, SetupTCPServerSocket(), in TCPServerUtility.c. This setup function iterates over the addresses returned from getaddrinfo(), stopping when it can successfully bind and listen or when it's out of addresses.

SetupTCPServerSocket()

```
1  static const int MAXPENDING = 5; // Maximum outstanding connection requests
2
3  int SetupTCPServerSocket(const char *service) {
```

```
4   // Construct the server address structure
5   struct addrinfo addrCriteria;                   // Criteria for address match
6   memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
7   addrCriteria.ai_family = AF_UNSPEC;             // Any address family
8   addrCriteria.ai_flags = AI_PASSIVE;             // Accept on any address/port
9   addrCriteria.ai_socktype = SOCK_STREAM;         // Only stream sockets
10  addrCriteria.ai_protocol = IPPROTO_TCP;         // Only TCP protocol
11
12  struct addrinfo *servAddr; // List of server addresses
13  int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
14  if (rtnVal != 0)
15    DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
16
17  int servSock = -1;
18  for (struct addrinfo *addr = servAddr; addr != NULL; addr = addr->ai_next) {
19    // Create a TCP socket
20    servSock = socket(servAddr->ai_family, servAddr->ai_socktype,
21        servAddr->ai_protocol);
22    if (servSock < 0)
23      continue;       // Socket creation failed; try next address
24
25    // Bind to the local address and set socket to list
26    if ((bind(servSock, servAddr->ai_addr, servAddr->ai_addrlen) == 0) &&
27        (listen(servSock, MAXPENDING) == 0)) {
28      // Print local address of socket
29      struct sockaddr_storage localAddr;
30      socklen_t addrSize = sizeof(localAddr);
31      if (getsockname(servSock, (struct sockaddr *) &localAddr, &addrSize) < 0)
32        DieWithSystemMessage("getsockname() failed");
33      fputs("Binding to ", stdout);
34      PrintSocketAddress((struct sockaddr *) &localAddr, stdout);
35      fputc('\n', stdout);
36      break;       // Bind and list successful
37    }
38
39    close(servSock);  // Close and try again
40    servSock = -1;
41  }
42
43  // Free address list allocated by getaddrinfo()
44  freeaddrinfo(servAddr);
45
46  return servSock;
47 }
```

SetupTCPServerSocket()

We also factor out accepting client connections into a separate function, `AcceptTCPConnection()`, in `TCPServerUtility.c`.

AcceptTCPConnection()

```
 1  int AcceptTCPConnection(int servSock) {
 2    struct sockaddr_storage clntAddr; // Client address
 3    // Set length of client address structure (in-out parameter)
 4    socklen_t clntAddrLen = sizeof(clntAddr);
 5
 6    // Wait for a client to connect
 7    int clntSock = accept(servSock, (struct sockaddr *) &clntAddr, &clntAddrLen);
 8    if (clntSock < 0)
 9      DieWithSystemMessage("accept() failed");
10
11    // clntSock is connected to a client!
12
13    fputs("Handling client ", stdout);
14    PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
15    fputc('\n', stdout);
16
17    return clntSock;
18  }
```

AcceptTCPConnection()

Note that we use `getsockname()` to print the local socket address. When you execute `TCPEchoServer.c`, it will print the wildcard local network address. Finally, we use our new functions in our protocol-independent echo server.

TCPEchoServer.c

```
 1  #include <stdio.h>
 2  #include "Practical.h"
 3  #include <unistd.h>
 4
 5  int main(int argc, char *argv[]) {
 6
 7    if (argc != 2) // Test for correct number of arguments
 8      DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
 9
10    char *service = argv[1]; // First arg:  local port
11
12    // Create socket for incoming connections
```

```
13    int servSock = SetupTCPServerSocket(service);
14    if (servSock < 0)
15      DieWithUserMessage("SetupTCPServerSocket() failed", service);
16
17    for (;;) { // Run forever
18      // New connection creates a connected client socket
19      int clntSock = AcceptTCPConnection(servSock);
20
21      HandleTCPClient(clntSock); // Process client
22      close(clntSock);
23    }
24    // NOT REACHED
25  }
```

TCPEchoServer.c

3.2.3 IPv4–IPv6 Interoperation

Our generic client and server are oblivious to whether they are using IPv4 or IPv6 sockets. An obvious question is, "What if one is using IPv4 and the other IPv6?" The answer is that if (and only if) the program using IPv6 is a *dual-stack* system—that is, supports *both* verson 4 and version 6—they should be able to interoperate. The existence of the special "v4-to-v6-mapped" address class makes this possible. This mechanism allows an IPv6 socket to be connected to an IPv4 socket. A full discussion of the implications of this and how it works is beyond the scope of this book, but the basic idea is that the IPv6 implementation in a dual-stack system recognizes that communication is desired between an IPv4 address and an IPv6 socket, and translates the IPv4 address into a "v4-to-v6-mapped" address. Thus, each socket deals with an address in its own format.

For example, if the client is a v4 socket with address 1.2.3.4, and the server is listening on a v6 socket in a dual-stack platform, when the connection request comes in, the server-side implementation will automatically do the conversion and tell the server that it is connected to a v6 socket with the v4-mapped address ::ffff:1.2.3.4. (Note that there is a bit more to it than this; in particular, the server side implementation will first try to match to a socket bound to a v4 address, and do the conversion only if it fails to find a match; see Chapter 7 for more details.)

If the server is listening on a v4 socket, the client is trying to connect from a v6 socket on a dual-stack platform, *and* the client has not bound the socket to a particular address before calling `connect()`, the client-side implementation will recognize that it is connecting to an IPv4 address and assign a v4-mapped IPv6 address to the socket at `connect()` time. The stack will "magically" convert the assigned address to an IPv4 address when the connection request is sent out. Note that, in both cases, the message that goes over the network is actually an IPv4 message.

While the v4-mapped addresses provide a good measure of interoperability, the reality is that the space of possible scenarios is very large when one considers v4-only hosts, v6-only hosts, hosts that support IPv6 but have no configured IPv6 addresses, and hosts that support IPv6 and use it on the local network, but have no wide-area IPv6 transport available (i.e., their providers do not support IPv6). Although our example code—a client that tries all possibilities returned by `getaddrinfo()`, and a server that sets AI_PASSIVE and binds to the first address returned by `getaddrinf()`—covers the most likely possibilities, **production code needs to be very carefully designed to maximize the likelihood that clients and servers will find each other under all conditions**. The details of achieving this are beyond the scope of this book; the reader should refer to RFC 4038 [11] for more details.

3.3 Getting Names from Numbers

So we can get an Internet address from a hostname, but can we perform the mapping in the other direction (hostname from an Internet address)? The answer is "usually." There is an "inverse" function called `getnameinfo()`, which takes a **sockaddr** address structure (really a **struct sockaddr_in** for IPv4 and **struct sockaddr_in6** for IPv6) and the address length. The function returns a corresponding node and service name in the form of a null-terminated character string—if the mapping between the number and name is stored in the name system. Callers of `getnameinfo()` must preallocate the space for the node and service names and pass in a pointer and length for the space. The maximum length of a node or service name is given by the constants NI_MAXHOST and NI_MAXSERV, respectively. If the caller specifies a length of 0 for node and/or service, `getnameinfo()` does not return the corresponding value. This function returns 0 if successful and a nonzero value if failure. The nonzero failure return code can again be passed to `gai_strerror()` to get the corresponding error text string.

---

int `getnameinfo` (**const struct sockaddr** *`address`, **socklen_t** `addressLength`,
char *`node`, **socklen_t** `nodeLength`, **char** * `service`,
socklen_t `serviceLength`, **int** `flags`)

As with `getaddrinfo()`, several flags control the behavior of `getnameinfo()`. They are described below; as before, some of them can be combined using bitwise OR ("|").

NI_NOFQDN Return only the hostname, not FQDN (Fully Qualified Domain Name), for local hosts. (The FQDN contains all parts, for example, protocols.example.com, while the hostname is only the first part, for example, "protocols".)

NI_NUMERICHOST Return the numeric form of the address instead of the name. This avoids potentially expensive name service lookups if you just want to use this service as a substitute for `inet_ntop()`.

NI_NUMERICSERV Return the numeric form of the service instead of the name.
NI_NAMEREQD Return an error if a name cannot be found for the given address. Without this option, the numeric form of the address is returned.
NI_DGRAM Specifies datagram service; the default behavior assumes a stream service. In some cases, a service has different port numbers for TCP and UDP.

What if your program needs its own host's name? `gethostname()` takes a buffer and buffer length and copies the name of the host on which the calling program is running into the given buffer.

---

int `gethostname`(**char** *`nameBuffer`, **size_t** `bufferLength`)

Exercises

1. `GetAddrInfo.c` requires two arguments. How could you get it to resolve a service name if you don't know any hostname?
2. Modify `GetAddrInfo.c` to take an optional third argument, containing "flags" that are passed to `getaddrinfo()` in the *ai_flags* field of the *addrinfo* argument. For example, passing "-n" as the third argument should result in the *ai_numerichost* flag being set.
3. Does `getnameinfo()` work for IPv6 addresses as well as IPv4? What does it return when given the address ::1?
4. Modify the generic `TCPEchoClient` and `TCPEchoServer` to eliminate the assumption mentioned at the end of Section 3.2.1.