chapter 8

Socket Programming in C++*

This book is for people who want to understand sockets. It's for people who want to know not only how to get a couple of programs to communicate over a network but also how and why the Sockets API works like it does. Or course, lots of developers use sockets all the time without really understanding these details. It's common to use sockets via a library that offers a simplified interface to socket creation, name resolution, and message transmission. This is particularly common in object-oriented languages like C++ and Java, where it's easy to wrap socket functionality in a collection of related classes.

The PracticalSocket library was developed to help expose students to the basics of socket programming without requiring a complete understanding of some of the material covered elsewhere in this book. This library is typical of object-oriented wrappers around socket functionality; it tries to offer a simple interface to the most commonly used functionality. The PracticalSocket library provides portability between Windows and UNIX platforms, and it can serve an instructional purpose since its source code is readily available.

A reader who is more comfortable in C or who prefers to start by understanding what's going on underneath should save this chapter for last. It can serve as a summary and application of many of the concepts introduced earlier. For a reader who is an experienced C++ programmer or who prefers to learn about sockets more selectively, this chapter can be read earlier and can serve as an overview of concepts covered in much more detail in earlier chapters. The examples presented here include many pointers to appropriate sections earlier in the text, so this chapter can serve as an entry point for many other sections of the book.

---

*Contributed by David Sturgill

In this chapter, we introduce the PracticalSocket library and demonstrate its use in a simple application. Through a series of more sophisticated applications, we expose additional features of the library and demonstrate how PracticalSockets or a similar library might be used in practice. Both the PracticalSocket library and the example programs used in this chapter to demonstrate it are available from the Web site for this text.

8.1 PracticalSocket Library Overview

Figure 8.1 illustrates the classes in PracticalSockets and their inheritance relationships. All the classes ending in "Socket" serve as wrappers around TCP or UDP sockets and provide a simple interface for creating a socket and using it for communication. The **SocketException** class provides support for error handling, and **SocketAddress** serves as a wrapper around an address and port number.

We can get started using this library without understanding everything about its classes and methods all at once. We'll start by covering just enough to write a simple application. From there, we can introduce new features gradually.

The **TCPSocket** class is the basic mechanism for communication over TCP. It is implemented as a wrapper around a TCP socket and serves as an endpoint in a bidirectional communication channel. If two applications want to communicate, they can each obtain an instance of **TCPSocket**. Once these two sockets are connected, sequences of bytes sent from one can be received at the other.

Functionality for **TCPSocket** is distributed across the class itself and its two base classes, **CommunicatingSocket** and **Socket**. The **Socket** class is at the top of the inheritance hierarchy, and contains only functionality common to all socket wrappers. Among other things, it has the job of keeping up with the underlying socket descriptor, and it automatically closes its descriptor when it is destroyed.

The **CommunicatingSocket** class is an abstraction for a socket that, once connected, can exchange data with a peer socket. It provides `send()` and `recv()` methods that are wrappers around the `send()` and `recv()` calls for the underlying socket descriptor. A successful call to the `send()` method of a **CommunicatingSocket** will send the first *`bufferLen`* bytes

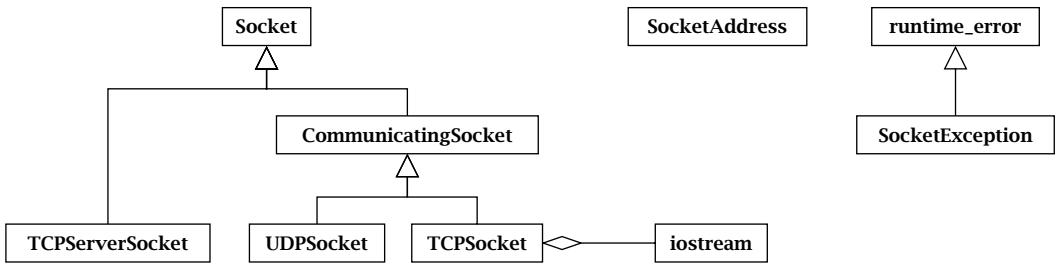

Figure 8.1: PracticalSockets class diagram.

pointed to by *buffer* to the peer **CommunicatingSocket**. A call to the `recv()` method of a **CommunicatingSocket** will attempt to read up to *bufferLen* bytes of data from the peer and store the result in the memory pointed to by *buffer*. The `recv()` method will block until data is available on the socket, and it will return the number of bytes received on the socket and written into *buffer*. After a socket is closed, `recv()` will return zero to indicate that no more data can be received.

---

void CommunicatingSocket::`send`(**const void** **buffer*, **int** *bufferLen*) throw(**SocketException**)
int CommunicatingSocket::`recv`(**void** **buffer*, **int** *bufferLen*) throw(**SocketException**)
int CommunicatingSocket::`recvFully`(**void** **buffer*, **int** *bufferLen*) throw(**SocketException**)

The stream of bytes transmitted between sockets may be fragmented into packets and reconstituted by buffering on its way from the sender to the receiver. **A group of bytes sent in a single call to** `send()` **may not all be received in a single, corresponding call to** `recv()`.  The `recvFully()` method is intended to help with this. It works just like `recv()`, except that it blocks until either exactly *bufferLen* bytes are received or the socket is closed. The return value from `recvFully()` reports the number of bytes received. Ordinarily, this will be the same as *bufferLen*. However, if the socket is closed before all the requested bytes are transmitted, a value less than *bufferLen* may be returned.

The PracticalSocket library uses C++ exceptions to report when something goes wrong. This is evident in the prototypes above. An instance of **SocketException** is thrown whenever an error occurs in the library. This exception object inherits from **runtime_error**, so it can be caught as an instance of **SocketException** by error-handling code specific to the communications portion of an application. A **SocketException** may be caught as a more general exception type by more general error-handling code. The `what()` method of a **SocketException** returns a string with a short description of the particular error that occurred.

The way to obtain an instance of **TCPSocket** depends on the role of the application. To establish a pair of connected **TCPSocket** peers, one application must function as a server and the other as a client. The server listens for new connections by creating an instance of **TCPServerSocket**. The other creates a **TCPSocket** directly. The **TCPServerSocket** class is derived from **Socket**, but not **CommunicatingSocket**. It is used to establish new TCP socket connections with client applications, but it is not itself used to send and receive bytes. The server must construct a **TCPServerSocket** with an application-defined port number. Afterward, a call to `accept()` will block until a client application attempts to connect by creating a **TCPSocket** with the same port number. When this happens, `accept()` will return a pointer to a new instance of **TCPSocket** that is connected to the **TCPSocket** peer on the client.

`TCPServerSocket`(**in_port_t** localPort) throw(**SocketException**)
TCPSocket ***TCPServerSocket**::`accept`() throw(**SocketException**)

The client application creates its end of the socket connection by simply constructing an instance of **TCPSocket** and providing the name or address of the server's host and the same port number. Once a pair of connected **TCPSocket** objects have been created, client and server can communicate using the `send()` and `recv()` methods until one of the endpoints closes its connection via the destructor or the `close()` method.

`TCPSocket(`**const char** *`foreignAddress`*, **in_port_t** *`foreignPort`*`)` throw(*`SocketException`*)
void Socket::`close()`

8.2 Plus One Service

The few classes and methods introduced so far are enough to let us implement a simple client and server similar to the ones in previous chapters. Here, accessing sockets through the PracticalSocket classes yields somewhat shorter source code that hides many of the details of the underlying API. The "plus one" service is a client-server application that performs the increment operation. The client sends an unsigned integer to the server, and the server sends back a value that is one greater.

8.2.1 Plus One Server

`PlusOneServer.cpp` is the server portion of the application. It accepts client connections, reads a 32-bit unsigned integer from each client, increments it, and then sends it back.

PlusOneServer.cpp

```
 1  #include <iostream>
 2  #include "PracticalSocket.h"
 3
 4  using namespace std;
 5
 6  int main(int argc, char *argv[]) {
 7    try {
 8      // Make a socket to listen for SurveyClient connections.
 9      TCPServerSocket servSock(9431);
10
11      for (;;) {                          // Repeatedly accept connections
12        TCPSocket *sock = servSock.accept(); // Get next client connection
13
14        uint32_t val;                     // Read 32-bit int from client
15        if (sock->recvFully(&val, sizeof(val)) == sizeof(val)) {
```

```
16          val = ntohl(val);               // Convert to local byte order
17          val++;                          // Increment the value
18          val = htonl(val);               // Convert to network byte order
19          sock->send(&val, sizeof(val));  // Send value back to client
20        }
21
22        delete sock;                      // Close and delete TCPSocket
23      }
24    } catch (SocketException &e) {
25      cerr << e.what() << endl;           // Report errors to the console
26    }
27
28    return 0;
29  }
```

PlusOneServer.cpp

1. **Application setup:** lines 1–6
   Access to the sockets API is hidden behind the PracticalSocket classes. The application only needs to include the header for the library and any calls we use directly.
2. **Error handling:** lines 7, 24–26
   Error handling is via exceptions. If an error occurs, it is caught at the end of the main function, and the program prints an error message before terminating.
3. **Create a server socket:** line 9
   The server creates a **TCPServerSocket** that listens for connections on port 9431. When constructed like this, the **TCPServerSocket** automatically makes the calls to `socket()`, `bind()` and `listen()` described in Sections 2.5, 2.6, and 2.7. If anything goes wrong in one of these steps, an exception is thrown.

   To keep the example simple, the server's port number is hard-coded in the constructor call. Of course, it would be more maintainable to use a named constant in a header file as the port number.
4. **Repeatedly accept client connections:** lines 11–12
   The server repeatedly calls the `accept()` method to wait for a new client connection. When a client connects, this method returns a pointer to a new **TCPSocket** for communicating with the client. This method is a wrapper around the `accept()` call described in Section 2.7. If something goes wrong in `accept()`, the catch block reports the error and the program terminates.
5. **Read an integer from the client and convert byte order:** lines 14–16
   Client and server have been written to exchange fixed-sized messages in the form of unsigned 32-bit integers. The server uses a stack-allocated integer, *val*, to hold the received message. Since the server is expecting a 4-byte message, it uses the `recvFully()`

method of **TCPSocket** to read the message directly into the storage for *val*. If the client connection is closed before the entire message arrives, `recvFully()` returns fewer than the expected number of bytes and the server ignores the message.

Although client and server agree on the size of a message, if they are running on different hosts, they may represent integers using different byte orders. To take care of this possibility, we agree to only transmit values that are in big-endian byte order. The server uses `ntohl()` to convert the received integer to local byte order if necessary.

Concerns over reading entire messages and adjusting byte order are really just the issues of framing and encoding. Chapter 5 focuses specifically on these topics and offers a variety of techniques for handling framing and encoding.

6. **Increment the client integer and send it back:** lines 17–19
   The server adds one to the client-provided value and then converts it back to network byte order. The server sends the value back to the client by sending a copy of the 4 bytes of memory used to represent *val*.
7. **Close client connection:** line 22
   When the server destroys its instance of **TCPSocket**, the socket connection is closed. If the server had neglected to destroy this object, it would have not only leaked memory but also leaked an underlying socket descriptor with each client connection. A server with this type of bug could very quickly reach an operating-system-enforced limit on per-process resources.

8.2.2 Plus One Client

`PlusOneClient.cpp` is the client counterpart of the plus one server. It connects to the server, sends it a copy of an integer value given on the command line, and prints out the value the server sends back.

PlusOneClient.cpp

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  #include "PracticalSocket.h"
 4
 5  using namespace std;
 6
 7  int main(int argc, char *argv[]) {
 8    if (argc != 3) {                       // Check number of parameters
 9      cerr << "Usage: PlusOneClient <server host> <starting value>" << endl;
10      return 1;
11    }
12
13    try {
```

```
14      TCPSocket sock(argv[1], 9431);      // Connect to the server.
15
16      uint32_t val = atoi(argv[2]);       // Parse user-suppled value
17      val = htonl(val);                   // Convert to network byte order
18      sock.send(&val, sizeof(val));       // Send to server.
19
20      // Read the server's response, convert to local byte order and print it
21      if (sock.recvFully(&val, sizeof(val)) == sizeof(val)) {
22        val = ntohl(val);
23        cout << "Server Response: " << val << endl;
24      }
25      // Socket is closed when it goes out of scope
26    } catch(SocketException &e) {
27      cerr << e.what() << endl;
28    }
29
30    return 0;
31  }
```

PlusOneClient.cpp

1. **Application setup and parameter checking:** lines 1–11
   The client uses the PracticalSocket classes along with support from a few other header files. On the command line, the client expects an address or hostname for the server and an integer value to send to the server. A usage message is printed if the wrong number of arguments is given.
2. **Error handling:** lines 13, 26–28
   As in the server, a socket-related exception is caught by code at the end of the program, and an error message is printed.
3. **Connect to the server:** line 14
   The client creates an instance of **TCPSocket**, passing in the user-supplied hostname and the hard-coded port number for the server. This constructor hides a lot of detail in the underlying sockets API. First, the given hostname is resolved to one or more addresses. As described in Chapter 3, `getaddrinfo()` returns all matching addresses for a given host and port. Some may be IPv4 addresses, and some may be IPv6. The **TCPSocket** constructor creates a socket for the first address and attempts to connect to it. If this fails, each successive address is tried until a connection can be established. If no addresses will connect, an exception is thrown.
4. **Parse, convert, and send value to server:** lines 16–18
   The client parses the user-provided value from the command line, converts it to network byte order, and sends it to the server by sending the 4 bytes starting at the value's starting address.

5. **Receive incremented value, convert, and print:** lines 21–24
   Using `recvFully()`, the client blocks until it either receives a 4-byte integer from the server or the socket connection is closed. If all 4 bytes arrive, the received value is converted to host byte order and printed.
6. **Close the socket:** line 25
   Since the client's **TCPSocket** is allocated on the stack, it is automatically closed and destroyed when the socket goes out of scope.

8.2.3 Running Server and Client

The executable `PlusOneServer` requires no command-line arguments. Once compiled, it should be started from the command prompt and permitted to run for as long as you want to try it out. While the server is running, you should run a copy of `PlusOneClient` from a different command prompt, possibly on a different host. Supply the name of the server's host and an integer value on the command line, and it should, with a little help from the server, produce a value one larger than the command-line argument. For example, if the server is running on a host named "venus," you should be able to run the client as follows:

PlusOneClient connecting to a server on host venus

```
% PlusOneClient venus 345318
Server Response: 345319
```

Exercises

1. Instead of simply incrementing the value from a single client, modify the server so that it will send back the sum of two values supplied by different clients. After a first client connects, the server will wait for a second client. The server will read integer values from both clients and send back the sum of these two values to both.
2. The client and server communicate using binary-encoded, fixed-sized messages. Modify these programs so that they use variable-length, text-encoded messages as described in Section 5.2.2. The sender will write its integer value to a character array as a sequence of ASCII-encoded digits. The receiver will read a copy of this array and parse out the integer.

8.3 Survey Service

Building on the concepts demonstrated in the plus one service and using techniques presented in Chapters 5 and 6, we can create a distributed application that does something a little more useful. The survey client and server implement a simple, distributed survey application. At start-up, the server reads a list of survey questions and responses from a text file. When a

client connects, the server sends it copies of the questions and response options. The client prints each question and sends the user's response back to the server, where it is recorded.

The file, `survey1.txt`, is an example survey file the server might use. The first line gives the number of questions. This is followed by a description for each question. A question is described by a one-line prompt, followed by a line giving the number of responses and a line of text for each response. For example, the first question on the survey below is, "What is your favorite flavor of ice cream?" The three possible responses are "Vanilla," "Chocolate," or "Strawberry." As users take the survey, the server will keep up with how many selected each of these responses.

survey1.txt

```
 1  2
 2  What is your favorite flavor of ice cream?
 3  3
 4  Vanilla
 5  Chocolate
 6  Strawberry
 7  Socket programming is:
 8  4
 9  Surprisingly easy
10  Empowering
11  Not for the faint of heart
12  Fun for the whole family
```

survey1.txt

8.3.1 Survey Support Functions

The client and server depend on common functionality implemented in `SurveyCommon.h` and `SurveyCommon.cpp`. The header file provides a named constant for the server's port number, a **Question** type for representing survey questions and prototypes for provided functions.

SurveyCommon.h

```
 1  #ifndef __SURVEYCOMMON_H__
 2  #define __SURVEYCOMMON_H__
 3
 4  #include "PracticalSocket.h"
 5  #include <string>
 6  #include <vector>
 7
 8  /** Port number used by the Survey Server */
```

```
 9  const in_port_t SURVEY_PORT = 12543;
10
11  /** Write an encoding of val to the socket, sock. */
12  void sendInt(CommunicatingSocket *sock, uint32_t val) throw(SocketException);
13
14  /** Write an encoding of str to the socket, sock. */
15  void sendString(CommunicatingSocket *sock, const std::string &str)
16    throw(SocketException);
17
18  /** Read from sock an integer encoded by sendInt() and return it */
19  uint32_t recvInt(CommunicatingSocket *sock) throw(std::runtime_error);
20
21  /** Read from sock a string encoded by sendString() and return it */
22  std::string recvString(CommunicatingSocket *sock) throw(std::runtime_error);
23
24  /** Representation for a survey question */
25  struct Question {
26    std::string qText;                    // Text of the question.
27    std::vector<std::string> rList;       // List of response choices.
28  };
29
30  /** Read survey questions from the given stream and store them in qList. */
31  bool readSurvey(std::istream &stream, std::vector<Question> &qList);
32
33  #endif
```

SurveyCommon.h

SurveyCommon.cpp

```
 1  #include "SurveyCommon.h"
 2
 3  using namespace std;
 4
 5  void sendInt(CommunicatingSocket *sock, uint32_t val) throw(SocketException) {
 6    val = htonl(val);                     // Convert val to network byte order
 7    sock->send(&val, sizeof(val));        // Send the value through the socket
 8  }
 9
10  void sendString(CommunicatingSocket *sock, const string &str)
11    throw(SocketException) {
12    sendInt(sock, str.length());            // Send the length of string
13    sock->send(str.c_str(),  str.length()); // Send string contents
14  }
15
```

```
16 uint32_t recvInt(CommunicatingSocket *sock) throw(runtime_error) {
17   uint32_t val;                          // Try to read a 32-bit int into val
18   if (sock->recvFully(&val, sizeof(val)) != sizeof(val))
19     throw runtime_error("Socket closed while reading int");
20
21   return ntohl(val);                     // Convert to host byte order, return
22 }
23
24 string recvString(CommunicatingSocket *sock) throw(runtime_error) {
25   uint32_t len = recvInt(sock);          // Read string length
26   char *buffer = new char [len + 1];     // Temp buffer to hold string
27   if (sock->recvFully(buffer, len) != len) {  // Try to read whole string
28     delete [] buffer;
29     throw runtime_error("Socket closed while reading string");
30   }
31
32   buffer[len] = '\0';                    // Null terminate the received string
33   string result(buffer);                 // Convert to an std::string
34   delete [] buffer;                      // Free temporary buffer
35   return result;
36 }
37
38 bool readSurvey(istream &stream, std::vector<Question> &qList) {
39   int count = 0;
40   stream >> count;                       // See how many questions there are
41   stream.ignore();                       // Skip past newline
42   qList = vector< Question >(count);
43
44   // Parse each question.
45   for (unsigned int q = 0; q < qList.size(); q++ ) {
46     getline(stream, qList[q].qText);     // Get the text of the question
47
48     count = 0;
49     stream >> count;                     // Read number of responses
50     stream.ignore();                     // Skip past newline
51
52     // Initialize the response list and populate it.
53     qList[q].rList = vector< string >(count);
54     for (unsigned int r = 0; r < qList[q].rList.size(); r++)
55       getline(stream, qList[q].rList[r]);
56   }
57
58   return stream;                         // Return true if stream is still good
59 }
```

SurveyCommon.cpp

1. **Integer encode and decode:** lines 5–8, 16–22
   In this application, client and server communicate by exchanging values of type integer and string. Integers are handled much like they are in the plus one service. To encode a 32-bit integer, it is first converted to network byte order and then sent out over a socket. To receive an integer, we attempt to read 4 bytes from the socket and, if successful, convert them to host byte order and return the result. If 4 bytes can't be read, an exception is thrown. Since this type of error is not produced in the PracticalSocket library itself, the exception is reported as a **runtime_error** rather than a **SocketException**.
2. **String encode and decode:** lines 10–14, 24–36
   Encoding and decoding of strings is more complicated because they can be of arbitrary length. Strings are encoded by first sending the string length and following this by the content of the string. The decoder tries to read both of these values and, if successful, converts the received string contents to a **string** object and returns it.
3. **Parse survey:** lines 38–59
   The survey is stored in a text file. The `readSurvey()` function reads the survey from the given input stream and fills in the given *qList* parameter with the sequence of questions.

8.3.2 Survey Server

The survey server is responsible for maintaining the list of survey questions, keeping up with user response totals, and interacting with the client.

The plus one server was able to handle only one client connection at a time. If multiple clients wanted to use the service, they would have to take turns. This makes sense for a simple application, where we can expect very short exchanges with each client. However, it may not work for the survey service. Here, users may deliberate for as long as they want over each question. If two users want to take the survey at the same time, it isn't reasonable to make one wait until the other is finished.

To interact with more than one client at the same time, the survey server creates a separate thread to handle interaction with each client. As in Section 6.4.2, the new thread manages the session with the client. Each time the server receives a response, it tallies it in its response count. Mutual exclusion helps the server to make sure two threads don't modify the response totals at the same time.

SurveyServer.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <pthread.h>
4  #include "PracticalSocket.h"
5  #include "SurveyCommon.h"
6
```

```
 7  using namespace std;
 8
 9  static vector<Question> qList;          // List of survey questions
10  static pthread_mutex_t lock;            // Mutex to Protect critical sections
11  static vector<vector<int> > rCount;     // Response tallies for each question
12
13  /** Thread main function to administer a survey over the given socket */
14  static void *conductSurvey(void *arg);
15
16  int main(int argc, char *argv[]) {
17    if (argc != 2) {
18      cerr << "Usage: SurveyServer <Survey File>" << endl;
19      return 1;
20    }
21
22    ifstream input(argv[1]);              // Read survey from given file
23    if (!input || !readSurvey(input, qList) ) {
24      cerr << "Can't read survey from file: " << argv[1] << endl;
25      return 1;
26    }
27
28    // Initialize response tally for each question/response.
29    for (unsigned int i = 0; i < qList.size(); i++)
30      rCount.push_back(vector<int>(qList[i].rList.size(), 0));
31    pthread_mutex_init(&lock, NULL);      // Initialize mutex
32
33    try {
34      // Make a socket to listen for SurveyClient connections.
35      TCPServerSocket servSock(SURVEY_PORT);
36
37      for (;;) {    // Repeatedly accept connections and administer the survey.
38        TCPSocket *sock = servSock.accept();
39
40        pthread_t newThread;              // Give survey in a separate thread
41        if (pthread_create(&newThread, NULL, conductSurvey, sock) != 0) {
42          cerr << "Can't create new thread" << endl;
43          delete sock;
44        }
45      }
46    } catch (SocketException &e) {
47      cerr << e.what() << endl;           // Report errors to the console.
48    }
49
50    return 0;
51  }
```

```
52
53  static void *conductSurvey(void *arg) {
54    TCPSocket *sock = (TCPSocket *)arg;   // Argument is really a socket
55    try {
56      sendInt(sock, qList.size());        // Tell client no of questions
57
58      for (unsigned int q = 0; q < qList.size(); q++) {
59        // For each question, send the question text and list of responses
60        sendString(sock, qList[q].qText);
61        sendInt(sock, qList[q].rList.size());
62        for (unsigned int r = 0; r < qList[q].rList.size(); r++)
63          sendString(sock, qList[q].rList[r]);
64
65        // Get the client's response and count it if it's in range
66        unsigned int response = recvInt(sock);
67        if (response >= 0 && response < rCount[q].size()) {
68          pthread_mutex_lock(&lock);      // Lock the mutex
69          rCount[q][response]++;          // Increment count for chosen item
70          pthread_mutex_unlock(&lock);    // Release the lock
71        }
72      }
73    } catch (runtime_error e) {
74      cerr << e.what() << endl;           // Report errors to the console.
75    }
76
77    delete sock;  // Free the socket object (and close the connection)
78    return NULL;
79  }
```

SurveyServer.cpp

1. **Access to library functions:** lines 1–5
   The server uses standard C++ I/O facilities and POSIX threads for interacting with multiple concurrent clients.
2. **Survey representation:** lines 9–11
   The survey is represented as a **vector** of **Question** instances. The variable, *rCount*, keeps up with user response counts, with each element of *rCount* corresponding to a survey question. Since each question has several possible responses, each element of *rCount* is a **vector** of totals, one for each response. Each time a user selects a response, the count for that question and response is incremented.

   If multiple clients connect to the server at the same time, it's possible that two or more of them will try to increment a counter at the same time. Depending on how this increment is performed by the hardware, this could leave the count in an unknown state. For example, an increment performed in one thread might overwrite the result of an

increment that was just completed in another thread. The chances of this happening are remote, but this possibility should not be left to chance. The *lock* variable is a mutex that is used to manage concurrent access to *rCount*. By using this synchronization object, it will be easy to make sure only one thread at a time tries to modify *rCount*.

The server uses file-scoped variables to keep up with the survey and responses. This makes it easy to access these data structures from any of our threads. Using the static modifier prevents these variables from polluting the global namespace, since they are only visible to a single implementation file. However, this organization would be an impediment to code reuse if, say, we wanted to run multiple surveys from the same application.

3. **Initialize server state:** lines 17–31
   The server parses the survey text from a user-supplied file. If anything goes wrong during this process, an error message is printed and the server exits. After the **Question** list has been successfully read, we know how many questions and responses there are, so we initialize the parallel response count representation. The mutex, *lock*, must also be initialized before it can be used.
4. **Create server socket and repeatedly accept clients:** lines 35–45
   The server creates a **TCPServerSocket** listening on an agreed-upon port number. It repeatedly waits for client connections and for each connection creates a new thread to handle interaction with the client. When creating the thread, the server gives a pointer to the `conductSurvey` function as the starting point for the new thread and a pointer to the new **TCPSocket** as the thread's argument. From this point on, the new thread is responsible for the socket object and for interacting with the client. Of course, if a new thread can't be created, the server prints an error message and cleans up by deleting the socket immediately.
5. **Survey client handler:** lines 53–79
   - **Thread start-up:** lines 53–54
     The `conductSurvey` function serves as the main function for each new thread. The thread will execute this function once it starts up, and it will terminate once it returns out of the function. The parameter and return types of this function are determined by the Pthreads API. The void pointers are intended to let us pass pointers to any type of data we need. Here, we know we passed in a pointer to a **TCPSocket** instance, so the first thing we do is cast it to a more specific type to make it easier to use.

     Depending on the number of clients connecting, there may be several copies of `conductSurvey` running at the same time, each on its own thread. While all of these threads may be running in the same function, each has its own local variables, and each is communicating over the instance of **TCPSocket** that was provided at thread start-up.
   - **Send each question to client:** lines 56–63
     The server uses functions provided by `SurveyCommon.cpp` to simplify interaction with the client. It sends the client the number of questions in the survey and

then sends each question and its response list, waiting for a response between questions.

- **Get client response and tally it:** lines 66–71
  The client indicates the user's chosen response by sending its index back to the server. Before tallying the response, the server makes sure it's in the range of legitimate responses. This check is very important. If the server incremented *rCount* using an arbitrary client-supplied index, it would be giving the client permission to make changes at unknown memory locations. A malicious client could use this to try to crash the server or, worse, take control of the server's host. Of course, we wrote the client and the server code. You will see that we perform a similar check on the client before we even send a response, so what's the point of also performing the check on the server? We're not concerned about the behavior of properly functioning clients. We're more concerned about what will happen if a malicious user creates a client that doesn't behave as nicely as the one we wrote.

  If a legitimate response is received, the client thread locks the mutex to make sure no other threads are trying to modify *rCount* at the same time. It then increments the appropriate count and unlocks the mutex right away to let other threads make changes to *rCount* as needed. This is typical of the operation of a multithreaded application. In general, we want to lock out other threads for as short a period as possible. Consider what would happen if threads locked the mutex once at the start of `conductSurvey` and then released it when they were done. This would eliminate the need to lock and unlock the mutex on every response, but it would completely suppress concurrency; only one thread at a time could interact with its client.

- **Error handling:** lines 55,73–75
  An exception occurring in the client thread will terminate the thread, but the server will continue to run and accept new connections. This makes sense as such an exception may simply result from a client terminating unexpectedly. If an exception is caught during client interaction, execution falls through to the thread exit code. Note that the exception is caught as the more general type, **runtime_exception**, since it may be thrown either by PracticalSockets or by our own `recvInt()` or `recvString()` functions.

- **Close socket and exit:** lines 77–78
  Since the thread in `conductSurvey` is responsible for its dynamically allocated **TCPClient** instance, it must free the instance before it exits. Like a C++ stream, deleting the object automatically closes the underlying socket.

8.3.3 Survey Client

The survey client is not as complicated as its server. Here, we don't have to worry about multithreading, concurrency, or maintaining response counts.

SurveyClient.cpp

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include "PracticalSocket.h"
 4  #include "SurveyCommon.h"
 5
 6  using namespace std;
 7
 8  int main(int argc, char *argv[]) {
 9    if (argc != 2) {                      // Make sure the user gives a host
10      cerr << "Usage: SurveyClient <Survey Server Host>" << endl;
11      return 1;
12    }
13
14    try {
15      // Connect to the server.
16      TCPSocket sock(argv[1], SURVEY_PORT);
17
18      // Find out how many questions there are.
19      int qCount = recvInt(&sock);
20      for (int q = 0; q < qCount; q++) {
21        // Show each the question to the user and print the list of responses.
22        cout << "Q" << q << ":  " << recvString(&sock) << endl;
23        int rCount = recvInt(&sock);
24        for (int r = 0; r < rCount; r++)
25          cout << setw(2) << r << " "  << recvString(&sock) << endl;
26
27        // Keep prompting the user until we get a legal response.
28        int response = rCount;
29        while (response < 0 || response >= rCount) {
30          cout << "> ";
31          cin >> response;
32        }
33
34        // Send the server the user's response
35        sendInt(&sock, response);
36      }
37    } catch(runtime_error &e) {
38      cerr << e.what() << endl;           // Report errors to the console.
39      return 1;
40    }
41
42    return 0;
43  }
```

1. **Access to library functions:** lines 1–4
2. **Connect to server:** lines 9–16
   The client expects the hostname of the server on the command line. If it's not given, an error message is printed and the client exits. The client attempts to create a **TCPSocket** object using the hostname and the server's port number defined in `SurveyCommon.h`. If a connection can't be established, the exception-handling code will report an error and exit.
3. **Receive and print survey questions:** lines 19–25
   Using functions provided by `SurveyCommon.cpp`, the client reads the number of questions and then reads each question and its list of responses.
4. **Read user responses and send them to the server:** lines 28–35
   For each question, the user is prompted for a response. The client checks to make sure the response is in the proper range before sending it to the server.

8.3.4 Running Server and Client

The `SurveyServer` requires one command-line argument, the name of the file containing the survey questions. For a survey file like the one above, we can run the server like:

SurveyServer offering questions from `survey1.txt`

```
% SurveyServer survey1.txt
```

The `SurveyClient` requires one command-line argument, the server's hostname or address. If the server is running on a system named "earth," we could run the client as:

SurveyClient connecting to a server on host earth

```
% SurveyClient earth
Q0: What is your favorite flavor of ice cream?
0 Vanilla
1 Chocolate
2 Strawberry
>
```

From this point, the user can respond to questions by entering the index of the desired response. The client terminates once all questions have been answered.

8.4 Survey Service, Mark 2

Although it performs a modestly useful function as it is, the survey service is ripe for enhancement. In this section, we add an administrative interface to report on response counts,

we have the server keep a list of IP addresses of clients completing the survey, and we explore an alternative technique for encoding and decoding messages between client and server. A few additional features of the PracticalSocket library are needed to make these enhancements to the survey service.

8.4.1 Socket Address Support

The PracticalSocket library provides a **SocketAddress** class that encapsulates an IP address and a port number. It's actually just a wrapper around the **sockaddr_storage** structure introduced in Section 2.4. Objects of this type are handy for keeping up with the endpoints of a socket connection, and the library uses them in many different places.

Through its constructors and static `lookupAddresses()` methods, **SocketAddress** provides a simple interface to name resolution. An instance of **SocketAddress** can be constructed with either an address or a hostname and a port number. Alternatively, the port can be described using a service name instead of a number. Both of these constructors are wrappers around the `getaddrinfo()` function described in Chapter 3. The underlying `getaddrinfo()` actually returns a list of matching addresses, and these constructors simply use the first address that's returned. The static `lookupAddresses()` methods take the same parameters as these constructors and return a list of all matching addresses in an STL **vector**. This could be useful for an application that needs to choose between IPv4 or IPv6 or when multiple network interfaces are available on a single host.

`SocketAddress`(**const char** **host*, **in_port_t** *port*) throw(**SocketException**)
`SocketAddress`(**const char** **host*, **const char** **service*) throw(**SocketException**)
static **std::vector**⟨**SocketAddress**⟩ **SocketAddress**::`lookupAddresses`(**const char** **host*, **in_port_t** *port*) throw(**SocketException**)
static **std::vector**⟨**SocketAddress**⟩ `lookupAddresses`(**const char** **host*, **const char** **service*) throw(**SocketException**)

Instances of **SocketAddress** can also be obtained by querying the addresses at the ends of a socket connection. The base class, **Socket**, provides a `getLocalAddress()` method that returns a **SocketAddress** for the local address to which a socket is bound. In the case of a **TCPServerSocket**, this will give the address on which the server is listening, and in the case of a **TCPSocket** this will give the address at the local end of the connection. The **CommunicatingSocket** class provides a `getForeignAddress()` method that returns a **SocketAddress** for the remote end of a connection. Once obtained, the host address and port number of a SocketAddress can be examined via the `getAddress()` and `getPort()` methods.

SocketAddress Socket::`getLocalAddress`() throw(**SocketException**)
SocketAddress CommunicatingSocket::`getForeignAddress`() throw(**SocketException**)
std::string SocketAddress::`getAddress`() const throw(**SocketException**)
in_port_t SocketAddress::`getPort`() const throw(**SocketException**)

For the user who wants to take more control over how a socket is set up, `bind()` and `connect()` methods are provided that take an instance of **SocketAddress** as a parameter. An application can create a **TCPSocket** or a **TCPServerSocket** using the default constructor. The socket can then be bound to a local address with the `bind()` method and, in the case of **TCPSocket**, connected to a remote server with the `connect()` method. This lets the programmer manage socket creation with a level of control more similar to what's described in Chapter 2.

void TCPSocket::`bind`(**const SocketAddress &***localAddress*) throw(**SocketException**)
void TCPSocket::`connect`(**const SocketAddress &***foreignAddress*) throw(**SocketException**)
void TCPServerSocket::`bind`(**const SocketAddress &***localAddress*) throw(**SocketException**)

8.4.2 Socket iostream Interface

The `send()` and `recv()` methods provided by **CommunicatingSocket** serve as a very thin veneer over the `send()` and `recv()` functions for the underlying socket. This kind of interface is natural for some types of messages, but it isn't the most familiar for handling text-encoded messages, as described in Section 5.2.2.

The `getStream()` method of **CommunicatingSocket** returns a reference to an **iostream** that's backed by the socket. For a C++ programmer, this can serve as a very convenient interface for reading and writing text-encoded information. It's a C++ analog to the `fdopen()` mechanism described in Section 5.1.5. Character sequences written to the **iostream** will be sent over the socket, and the sequence of characters received from the network can be read from the **iostream**. With this interface, the programmer can use the familiar techniques for working with **istream** and **ostream** to encode and decode messages.

std::iostream &CommunicatingSocket::`getStream`() throw(**SocketException**)

The **iostream** interface to a **CommunicatingSocket** provides fixed-sized buffers for storing part of the character sequence being sent and received. Text written to the **iostream**

will be held in memory until the buffer is full or until it is flushed. This can offer some performance advantages since a message can be written to the **iostream** one field at a time and then pushed out to the network after it's complete. However, this level of buffering makes it problematic to mix I/O operations via the **iostream** with direct calls to the `send()` and `recv()` methods. Consider, for example, some message *A* written to the socket via its **iostream**. If the stream isn't subsequently flushed, part of message *A* may remain buffered in the **iostream**. If message *B* is then sent directly via the socket's `send()` method, *B* will be pushed out to the network before the buffered portions of *A*.

8.4.3 Enhanced Survey Server

The enhanced server uses **SocketAddress** objects to keep up with a list of addresses for clients completing the survey. For each client, the server records the host address by saving a copy of the **SocketAddress** representing the remote end of the socket.

The server also provides an administrative socket interface for reporting the current state of the survey. The server restricts access to the administrative interface by only permitting connections from clients running on the same host. To do this, the server cannot use the convenience constructor for **TCPServerSocket** used in previous examples. The server must create a **TCPServerSocket** in the unbound state and then explicitly bind it to a **SocketAddress** for the loopback interface.

The enhanced survey server makes extensive use of the **iostream** interface of **CommunicatingSocket**. This makes the sending and receiving of messages look a lot like reading and writing a file. It also permits some additional code reuse between the client and server. Instead of using a new format to send survey questions to the client, the server encodes them in the same format as the survey file it reads at start-up time. The client can use the same `readSurvey()` function provided by `SurveyCommon.cpp` to read the list of survey questions all at once.

SurveyServer2.cpp

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include <fstream>
 4  #include <pthread.h>
 5  #include "PracticalSocket.h"
 6  #include "SurveyCommon.h"
 7
 8  using namespace std;
 9
10  static vector<Question> qList;          // List of survey questions
11  static pthread_mutex_t lock;            // Mutex to Protect critical sections
12  static vector<vector<int> > rCount;     // Response tallies for each question
13  static vector<SocketAddress> addrList;  // Address list for client history
```

```
14
15  /** Thread main function to administer a survey over the given socket */
16  void *conductSurvey(void *arg);
17
18  /** Thread main function to monitor an administrative connection and
19      give reports over it. */
20  void *adminServer(void *arg);
21
22  int main(int argc, char *argv[]) {
23    if (argc != 2) {
24      cerr << "Usage: SurveyServer <Survey File>" << endl;
25      return 1;
26    }
27
28    ifstream input(argv[1]);              // Read survey from given file
29    if (!input || !readSurvey(input, qList) ) {
30      cerr << "Can't read survey from file: " << argv[1] << endl;
31      return 1;
32    }
33
34    // Initialize response tally for each question/response.
35    for (unsigned int i = 0; i < qList.size(); i++)
36      rCount.push_back(vector<int>(qList[i].rList.size(), 0));
37    pthread_mutex_init(&lock, NULL);      // Initialize mutex
38
39    try {
40      pthread_t newThread;
41
42      // Make a thread to provide the administrative interface.
43      if (pthread_create(&newThread, NULL, adminServer, NULL) != 0) {
44        cerr << "Can't create administrative thread" << endl;
45        return 1;
46      }
47
48      // Make a socket to listen for SurveyClient connections.
49      TCPServerSocket servSock(SURVEY_PORT);
50
51      for (;;) {    // Repeatedly accept connections and administer the survey.
52        TCPSocket *sock = servSock.accept();
53        if (pthread_create(&newThread, NULL, conductSurvey, sock) != 0) {
54          cerr << "Can't create new thread" << endl;
55          delete sock;
56        }
57      }
58    } catch (SocketException &e) {
```

```
59      cerr << e.what() << endl;           // Report errors to the console.
60    }
61
62    return 0;
63  }
64
65  void *conductSurvey(void *arg) {
66    TCPSocket *sock = (TCPSocket *)arg;   // Argument is really a socket.
67    try {
68      // Write out the survey in the same format as the input file.
69      iostream &stream = sock->getStream();
70      stream << qList.size() << "\n";
71
72      for (unsigned int q = 0; q < qList.size(); q++) {
73        stream << qList[q].qText << "\n";
74        stream << qList[q].rList.size() << "\n";
75        for (unsigned int r = 0; r < qList[q].rList.size(); r++)
76          stream << qList[q].rList[r] << "\n";
77      }
78      stream.flush();
79
80      // Read client responses to questions and record them
81      for (unsigned int q = 0; q < qList.size(); q++) {
82        unsigned int response;
83        stream >> response;
84        if (response >= 0 && response < rCount[q].size()) {
85          pthread_mutex_lock(&lock);      // Lock the mutex
86          rCount[q][response]++;          // Increment count for chosen item
87          pthread_mutex_unlock(&lock);    // Release the lock
88        }
89      }
90
91      // Log this client as completing the survey.
92      pthread_mutex_lock(&lock);
93      addrList.push_back(sock->getForeignAddress());
94      pthread_mutex_unlock(&lock);
95    } catch (runtime_error e) {
96      cerr << e.what() << endl;           // Report errors to the console.
97    }
98
99    delete sock;  // Free the socket object (and close the connection)
100   return NULL;
101 }
102
103 void *adminServer(void *arg) {
```

```
104    try {
105      // Make a ServerSocket to listen for admin connections
106      TCPServerSocket adminSock;
107      adminSock.bind(SocketAddress("127.0.0.1", SURVEY_PORT + 1));
108
109      for (;;) {   // Repeatedly accept administrative connections
110        TCPSocket *sock = adminSock.accept();
111        iostream &stream = sock->getStream();
112
113        try {
114          // Copy response counts and address lists
115          pthread_mutex_lock(&lock);
116          vector<vector<int> > myCount = rCount;
117          vector<SocketAddress> myList = addrList;
118          pthread_mutex_unlock(&lock);
119
120          for (unsigned int q = 0; q < qList.size(); q++) {
121            // Give a report for each question.
122            stream << "Q" << q << ":  " << qList[q].qText << "\n";
123            for (unsigned int r = 0; r < qList[q].rList.size(); r++)
124              stream << setw(5) << myCount[q][r] << " " << qList[q].rList[r]
125                     << "\n";
126          }
127
128          // Report the list of client addresses.
129          stream << "Client Addresses:" << endl;
130          for (unsigned int c = 0; c < myList.size(); c++ )
131            stream << "  " << myList[c].getAddress() << "\n";
132
133          stream.flush();
134        } catch (runtime_error e) {
135          cerr << e.what() << endl;
136        }
137
138        delete sock;   // Free the socket object (and close the connection)
139      }
140    } catch (SocketException e) {
141      cerr << e.what() << endl;           // Report errors to the console.
142    }
143    return NULL;                          // Reached only on error
144  }
```

SurveyServer2.cpp

1. **Access to library functions:** lines 1–6
2. **Survey representation:** lines 10–13
   This version of the server adds a **vector** of **SocketAddress** objects to keep up with the list of clients completing the survey. Since multiple client threads may attempt to access this list at the same time, use of the list is protected by the mutex, *lock*.
3. **Initialize server state:** lines 23–37
4. **Create a thread for the administrative interface:** lines 43–46
   The server must accept connections over its primary **TCPServerSocket** and its administrative **TCPServerSocket**. If the same thread tried to serve both of these sockets, calling `accept()` on one of them would leave it blocked and unable to accept connections from the other. To serve both, the server creates a new thread to handle connections over its administrative interface. The `adminServer()` function implements this interface.
5. **Create server socket and handle each client with a new thread:** lines 49–57
6. **Survey client handler:** lines 65–101
   - **Encode survey and send to client:** lines 69–78
     A client thread first writes a copy of the whole survey to the client, using the same format as the original input file. It uses `getStream()` to obtain a copy of the **iostream** for its **TCPSocket** and then writes out the survey as if it were writing to a file. Instead of using *endl* to end each line, we use the newline character. Using *endl* would flush the stream after each line. As much as possible, we would like to buffer all or large portions of the message and then send when the message is complete. The server writes the entire message and then calls the stream's `flush()` method to initiate the sending of any buffered data.
   - **Get client responses and tally them:** lines 81–89
     The client has the entire survey, so the server only needs to read its responses one after another.
   - **Log the client:** lines 92–94
     Once the client has completed the survey, the server records a copy of its address.
7. **Administrative client handler:** lines 103–144
   - **Create administrative socket:** lines 106–107
     The administrative interface uses a port number offset by one from the survey port. Since the server is intended to only accept administrative connections from the local host, we have to go through more explicit steps to construct the **TCPServerSocket** and bind it to a local address.
   - **Repeatedly accept administrative connections:** lines 109–110
     Administrative clients are served with a single thread. If two administrative clients try to connect, the second will have to wait for the first to be served.

- **Snapshot reported structures:** lines 115–118
  Before sending a report, the server makes a copy of the client address history and the response count lists. This frees the server from having to lock these structures repeatedly as it writes out the report, but, more importantly, it provides a consistent snapshot of the state. The reported list of client addresses won't grow while the server is printing it out. If the server had simply locked the mutex for the entire report generation instead of copying these lists, all client threads would have been blocked while the server tried to send the report to the administrative client. A malicious administrative client could fail to read from its socket and suspend the conducting of surveys indefinitely.
- **Write administrative report:** lines 120–133
  We make the server responsible for formatting the administrative report. It writes a tally for each question and response to the socket's **iostream**, followed by the host addresses for every client completing the survey.

8.4.4 Enhanced Survey Client

To communicate with the server using text-encoded messages, the client requires a few changes. First, instead of reading survey questions individually, it obtains the socket's **iostream** and uses `readSurvey()` to read the whole survey at once. The client offers the survey to the user one question at a time and sends corresponding responses by writing to the stream.

SurveyClient2.cpp

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include "PracticalSocket.h"
 4  #include "SurveyCommon.h"
 5
 6  using namespace std;
 7
 8  int main(int argc, char *argv[]) {
 9    if (argc != 2) {                      // Make sure the user gives a host
10      cerr << "Usage: SurveyClient <Survey Server Host>" << endl;
11      return 1;
12    }
13
14    try {
15      // Connect to the server.
16      TCPSocket sock(argv[1], SURVEY_PORT);
17      iostream &stream = sock.getStream();
18      vector<Question> qList;            // Read the whole survey
19      readSurvey(stream, qList);
```

```
20
21      for (unsigned int q = 0; q < qList.size(); q++) {
22        // Show each the question to the user and print the list of responses.
23        cout << "Q" << q << ":  " << qList[q].qText << endl;
24        for (unsigned int r = 0; r < qList[q].rList.size(); r++)
25          cout << setw(2) << r << " "  << qList[q].rList[r] << endl;
26
27        // Keep prompting the user until we get a legal response.
28        unsigned int response = qList[q].rList.size();
29        while (response < 0 || response >= qList[q].rList.size()) {
30          cout << "> ";
31          cin >> response;
32        }
33
34        stream << response << endl;       // Send user response to server
35        stream.flush();
36      }
37    } catch(SocketException &e) {
38      cerr << e.what() << endl;           // Report errors to the console.
39      return 1;
40    }
41
42    return 0;
43  }
```

SurveyClient2.cpp

8.4.5 Administrative Client

A very simple client is sufficient to access the administrative interface. Since the server writes out response counts and address list in a human-readable format, the client can just copy each byte it receives from the server to the console. The administrative client first attempts to connect to a server running on the local host. Since the client is attempting to connect using the administrative port number, the server will automatically send it a report and then close the socket connection.

The client creates a fixed-sized buffer and then copies one buffer full after another to the console. Even though the server writes to the socket using its **iostream** interface, the socket simply conveys the sequence of bytes written, and the client is free to read the sequence using the `recv()` method. In this case, `recv()` is probably a more convenient interface than **iostream** for echoing the server's output to the console. The client doesn't have to worry about how the server's report is formatted, how many lines it contains, or how long each line is.

The server's administrative report is just a sequence of printable characters, with no null terminators anywhere in the report. To print out a buffer full of the report as if it was a string, the client must mark the end of the buffer with a null terminator. Since the `recv()` method

returns the number of bytes successfully saved in *buffer*, it's easy to fill in a null terminator at the end. Of course, the client has to be certain to leave room for this extra character each time a buffer is read. That's why the client's call to `recv()` advertises the buffer as one byte shorter than its actual capacity.

AdminClient2.cpp

```
 1  #include <iostream>
 2  #include "PracticalSocket.h"
 3  #include "SurveyCommon.h"
 4
 5  using namespace std;
 6
 7  int main(int argc, char *argv[]) {
 8    try {
 9      // Connect to the server's administrative interface.
10      TCPSocket sock("localhost", SURVEY_PORT + 1);
11
12      // Read the server's report a block at a time.
13      char buffer[ 1025 ];
14      int len;
15      while ((len = sock.recv(buffer, sizeof(buffer) - 1)) != 0) {
16        buffer[len] = '\0';               // Null terminate the sequence
17        cout << buffer;                   // And print it like as a string
18      }
19    } catch(SocketException &e) {
20      cerr << e.what() << endl;           // Report errors to the console.
21      exit(1);
22    }
23
24    return 0;
25  }
```

AdminClient2.cpp

8.4.6 Running Server and Clients

The enhanced survey server and its survey client are run with the same arguments as the original. The administrative client doesn't require any arguments since it automatically tries to connect to an instance of the survey server running on the local host at a known port number.

Exercises

1. In the server's code for the administrative interface, we made an effort to get a consistent snapshot of the response tally and the client address list before we started generating the report. However, since the server's `conductSurvey()` function records client responses as soon as they are received, the administrative report may still include results for partially completed surveys. Modify the server so that responses are tallied only after the client completes the survey. In this way, both the response tallies and the list of client addresses will reflect only clients that actually completed the survey.
2. As it is written, it's possible for a client to cause a server crash. If the client closes its socket connection while the server is sending it a message, the server will receive a SIGPIPE signal. Consult Section 6.2 and modify the server so that it will not terminate under these circumstances.
3. The survey server maintains an open socket and a thread on behalf of each client taking a survey. If a client never completes the survey, the server never reclaims these resources. Extend the server so that sockets are closed and client threads are permitted to terminate automatically five minutes after the client begins taking the survey. When a client thread starts up, you will also start up another thread that we'll call the sleeper. The sleeper will simply wait for five minutes and then check to see if the client thread is still running. If it is, the sleeper will close the client's socket, which will unblock the server thread and give it a chance to exit. Doing this the right way will require the server to maintain some additional per-client state and to perform some additional synchronization. For example, a sleeper thread should not make a call to `close()` if its client thread has already finished and deleted the socket.