CHAPTER 9

Case Study: A Networked Application

In this chapter, we will bring together everything we've talked about in Parts One and Two and build a chat application. It will be written in C++ and consist of both a server and a GUI client. The server will be multithreaded as described in Chapter 5 and will use the chat protocol developed in Chapter 6.

The Server

The server has no user interface and would be a good candidate for a daemon, although we won't complicate it with our daemonize function from Chapter 7 here. The basic architecture of the server is that it will spawn a thread for each client that connects. It will have a global list of the connected clients, and each thread will relay server messages to that list. The list will be protected from simultaneous modification by means of a mutex. The main loop for each thread will first check to see if the client has sent a command. If it has, it is dealt with and a return code is sent. Then the thread checks to see if there are any pending messages to be sent to the client. This loop continues until the client disconnects or the user has been kicked out of the room.

First, we include the system headers and signify that we will be using the Standard Template Library (STL) namespace.

/* chatsrv.cpp */
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <pthread.h>
#include <pthread.h>
using namespace std;
Next, we define a couple of constants. First, we signify that the server will listen on port 5296, and we set the maximum line length to be 1024 bytes.

/* define's */
#define LISTEN_PORT 5296
#define MAX_LINE_BUFF 1024

Here we define two structures. The first will help us keep track of each connected client. We have two flags: one to signify whether the user is a room operator and the other to signify that the user was kicked out of the room. Finally, we have a vector of strings. This will contain a list of messages that are being relayed to the client.

/* Structures */
struct client_t {
    bool opstatus;
    bool kickflag;
    vector<string> outbound;
};

The second structure is for holding the client commands. We will parse the command into this structure for easier handling. The maximum number of operands for any command is two.

struct cmd_t {
    string command;
    string op1;
    string op2;
};

Now we declare some global variables. First, we have a mutex to synchronize access to the room topic. Then, we have the room topic variable. Next, we have a mutex to synchronize access to the client list. We are using an STL map keyed on username to hold our list of connected users. Each entry is a client_t structure.

/* Globals */
pthread_mutex_t room_topic_mutex;
string room_topic;
pthread_mutex_t client_list_mutex;
map<string, client_t> client_list;

Here are our forward declarations for the functions we will use: