A "Host" Wrapper Class for Providing Telnet Services under Win32

Tom Wheeler, NØGSG
twheeler@kc.devry.edu

Introduction

Remote access to an application and its data can be an important feature, but the reality of providing this access has in the past been fairly painful. The `CHost` class eliminates most of the hassle and lets you provide a simple, streamlined command-line (Telnet) interface within your Win32 application. In less than 50 lines of code, you can craft a very functional remote access routine! As a bonus, you automatically get multi-client connectivity. You decide whether only one or 10,000 clients may connect simultaneously with your software. `CHost` takes care of all the low-level management functions (such as threads) for you.

Basis of Operation

In order to use the `CHost` class, you simply provide one or two functions that CHost will call when needed. These functions do the specific I/O you require. Here's how it works:

- You construct a `CHost` object, passing pointers to the two functions, as well as some other information (such as how many clients will be allowed to connect at one time). One of the functions, the `TalkToClient()` function, is required; the `OnIdle()` function is optional, and you may pass `NULL` if you don't need this feature.

- You set the `m_nServerSocketTimeout` variable in the `CHost` object to the number of milliseconds between calls to `OnIdle()`. (This defaults to 500 ms, or two `OnIdle()` calls per second).

- You then call the `StartService()` method on the `CHost` object, and check the return code to see that it starts successfully.

- Then continue your software as before. (You don't have to do anything else at this point, as `CHost` runs in its own thread to take care of the client hosting requirements.)

- Periodically, `CHost` will call your `OnIdle()` method. (For example, if you want to perform an advertisement or some other fixed-time function, do it here.) You don't have to use this feature.

- When a client connects, `CHost` will create a new thread (if needed) and call your `TalkToClient()` method. Your supplied method interacts with the remote user or agent. When this method returns, the client is automatically disconnected.

  If you allow multiple clients, each client will call a new instance of the `TalkToClient()` method you provide, each in a new thread. (It is still your responsibility to properly manage this method in a thread-safe manner, should you support multiple clients.)

- CHost maintains a `nRunFlag` variable that is passed by reference to every instance of `TalkToClient()`. This flag, when non-zero, gives permission for the `TalkToClient()` function to continue running. When zeroed, the `CHost` service is being terminated, and the `TalkToClient()` function must gracefully return to allow this to happen.

Dependencies

The `CHost` class utilizes the `Socket` and `ServerSocket` wrapper classes; they're provided in the sample Visual C project, and are documented at:

http://faculty.kc.devry.edu/twheeler/projects/socketclass.pdf.

Sample Project

A sample Visual C project contains the CHost class as well as a short demonstration program. This project can be downloaded from:

http://faculty.kc.devry.edu/twheeler/projects/tcphost.zip.

Sample Project Analysis

Very few lines of code are used in the sample project. Within the main() function, here's what takes place:

First, a `CHost` object is created, and its `m_nServerSocketTimeout` value is adjusted to 1000 ms:

```
pHost = new CHost(TalkToClient,       // Address of Client interactiion routine
                       OnIdle,        // Address of OnIdle routine
                       10,            // Number of clients (>1 means multi-threaded)
                       nPort);        // TCP port number to bind & listen on


pHost->m_nServerSocketTimeout = 1000;
```

The service is then started:

```
nResult = pHost->StartService();              // Start the service

if (nResult)
        {
        printf("Failed to start service, exiting.\n");
        return 1;
        }
```

While the service is running, our program doesn't have any other processing to do, so it simply waits for the ENTER key, then stops the service:

```
printf("Started on port %d\n\n", nPort );

// Service started; terminate it when ENTER is pressed.

printf("Press ENTER to terminate service.\n\n");
fflush(stdin);
gets(buf);

printf("\nStopping Service...");
pHost->StopService();
printf("Stopped.\n\n");

delete pHost;          // Clean up, exit
return 0;
}
```

The `TalkToClient()` method is also very straightforward. It is called by the CHost framework automatically when someone connects to your service:

```
void TalkToClient(Socket& s, int& nRunFlag)
```

`CHost` passes two parameters to this method, a reference to a `Socket` object (which is connected to the remote user), and a reference to the `nRunFlag`. For example, to send a logon message to the remote user, `TalkToClient` simply uses a `Write()` instruction on the `Socket`:

```
s.Write("\r\nHello, TCP World.\r\n");
s.Write("Enter your name > ");
```

The `CHost` class provides handy utilities, such as `LineInput()`, for interacting with remote callers:

```
nResult = aHost.LineInput(buf,32,1,&s,30000);
```

In this `LineInput` call, the user's text (up to 32 characters) is typed into `buf`, with normal echo (`1`), using the I/O of the passed socket (`&s`), with a time-out of 30 seconds (`30000` ms).

Remember that `TalkToClient()` runs in its own thread (or threads, if you allow more than one client), and that it will be running asynchronous to your program's main process. (Be careful; you may need use synchronization methods, such as critical sections, events, or mutexes, to coordinate the `TalkToClient()` method and your main code.)

Summary

Providing a Telnet interface in your software is now easy and very convenient. With just a few lines of code, you can implement both single and multi-user hosts to give your application the power of Internet connectivity.