

Lab Practice 6

- Multithreaded Socket Server

In this lab, we will create Multithreaded Socket Server.

Exercise 1

- Multithreaded Socket Server

This *Multithreaded Socket Server* is going to echo back the message to the same client which has sent the message.

```
0 //-----
1 // Simple multithreaded socket server.
2 //
3
4 #ifndef WIN32_LEAN_AND_MEAN
5 #define WIN32_LEAN_AND_MEAN
6 #endif
7 #include <winsock2.h>
8 #include <iostream>
9
10 using namespace std;
11
12 /// Link with ws2_32.lib
13 #pragma comment(lib, "Ws2_32.lib")
14
15 #define DEFAULT_BUFSIZE 8192
16 #define DEFAULT_PORT 7890
17 #define MAX_SESSION_LIMIT 100
18
19 struct SessionInfo {
20     SOCKET Socket;           /// Socket Handle
21     char IPAddress[16];    /// IP Address, 'xxx.xxx.xxx.xxx'
22     char MessageBuffer[DEFAULT_BUFSIZE]; // Message buffer
23     int MessageSize;        /// Message size in the buffer
24     HANDLE hThread;         /// Thread handle
25     DWORD dwThreadId;       /// Thread ID
26     CRITICAL_SECTION cs_SessionList;
27     // more information, what you need for your server.
28 } g_SessionList[MAX_SESSION_LIMIT];
29
30 int add_new_session( SOCKET NewSocket );
31 void close_session( int SessionIndex );
32 DWORD WINAPI RecvThread( void *arg );
33
34 int main( void )
35 {
36     WSADATA wsaData;          /// Declare some variables
37     SOCKET ListenSocket = INVALID_SOCKET; // Listening socket to be created
38     SOCKET ClientSocket = INVALID_SOCKET; // Client socket to be created
39     sockaddr_in ServerAddress;      /// Socket address to be passed to bind
40     sockaddr_in ClientAddress;     /// Address of connected socket
41     int ClientAddressLen;         /// Length for sockaddr_in.
42     int Result = 0;               /// used to return function results
43 }
```

```
44     int      SessionIndex;
45
46     //-----
47     /// Initialize the session structure array
48     for( int i = 0; i < MAX_SESSION_LIMIT; i++ )
49     {
50         memset( &g_SessionList[i], '\0', sizeof(struct SessionInfo) );
51     }
52
53     //-----
54     /// 1. Initialize Winsock
55     Result = WSASStartup( MAKEWORD( 2, 2 ), &wsaData );
56     if( NO_ERROR != Result )
57     {
58         printf( "Error at WSASStartup()\n");
59         return 1;
60     }
61     else
62     {
63         printf( "WSASStartup success.\n");
64     }
65
66     //-----
67     /// 2. Create a SOCKET for listening for
68     ///     incoming connection requests
69     ListenSocket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
70     if( INVALID_SOCKET == ListenSocket )
71     {
72         printf( "socket function failed with error: %d\n", WSAGetLastError() );
73         WSACleanup( );
74         return 1;
75     }
76     else
77     {
78         printf( "socket creation success.\n");
79     }
80
81     //-----
82     /// 3-1. The sockaddr_in structure specifies the address family,
83     ///     IP address, and port for the socket that is being bound.
84     ServerAddress.sin_family = AF_INET;
85     ServerAddress.sin_port = htons( DEFAULT_PORT );
86     ServerAddress.sin_addr.s_addr = htonl( INADDR_ANY );
87
88     //-----
89     /// 3-2. Bind the socket.
90     Result = bind( ListenSocket, (SOCKADDR *)&ServerAddress,
91                 sizeof( ServerAddress ) );
92     if( SOCKET_ERROR == Result )
93     {
94         printf( "bind failed with error :%d\n", WSAGetLastError() );
95         closesocket( ListenSocket );
96         WSACleanup( );
97         return 1;
98     }
99     else
100    {
101        printf( "bind returned success\n");
102    }
103
104    //-----
105    /// 4. Listen for incoming connection requests
```

```
106     /// on the created socket
107     if( SOCKET_ERROR == listen( ListenSocket, SOMAXCONN ) )
108     {
109         printf( "listen function failed with error: %d\n", WSAGetLastError() );
110     }
111     printf( "Listening on socket...\\n");
112
113     //-----
114     /// Main loop.
115     do
116     {
117         //-----
118         /// Create a SOCKET for accepting incoming requests.
119         printf( "Waiting for client to connect...\\n");
120
121         //-----
122         /// 5. Accept the connection.
123         ClientAddressLen = sizeof(ClientAddress);
124         ClientSocket = accept( ListenSocket,
125                               (struct sockaddr*)&ClientAddress,
126                               &ClientAddressLen );
127         if( INVALID_SOCKET == ClientSocket )
128         {
129             printf( "accept failed with error: %d", WSAGetLastError() );
130             closesocket( ListenSocket );
131             WSACleanup( );
132             return 1;
133         }
134
135         SessionIndex = add_new_session( ClientSocket );
136         printf( "Client connected. IP Address:%d.%d.%d.%d, Port Number:%d,
SessionIndex:%d\\n",
137                 (ClientAddress.sin_addr.S_un.S_un_b.s_b1),
138                 (ClientAddress.sin_addr.S_un.S_un_b.s_b2),
139                 (ClientAddress.sin_addr.S_un.S_un_b.s_b3),
140                 (ClientAddress.sin_addr.S_un.S_un_b.s_b4),
141                 ntohs( ClientAddress.sin_port ),
142                 SessionIndex);
143
144         //-----
145         /// Create thread for new connected session
146         g_SessionList[SessionIndex].hThread = CreateThread( NULL, 0,
147                                               RecvThread,           /* Thread function */
148                                               (LPVOID)SessionIndex, /* Passing Argument */
149                                               0, &g_SessionList[SessionIndex].dwThreadID );
150     } while( 1 );
151
152     //-----
153     /// Closes an existing socket
154     closesocket( ListenSocket );
155
156     //-----
157     /// 8. Terminate use of the Winsock 2 DLL (Ws2_32.dll)
158     WSACleanup( );
159
160     return 0;
161 }
162
163 int add_new_session( SOCKET NewSocket )
164 { // Let's just make it easy for this time!
165     int SessionIndex;
```

```

167     for( SessionIndex = 0; SessionIndex < MAX_SESSION_LIMIT; SessionIndex++ )
168     {
169         if( 0 == g_SessionList[SessionIndex].Socket )
170         { // find the empty slot!
171             //Initialize the critical section
172             InitializeCriticalSection(&g_SessionList[SessionIndex].cs_SessionList);
173             g_SessionList[SessionIndex].Socket = NewSocket;
174             g_SessionList[SessionIndex].MessageSize = 0;
175             // More information for Session.
176
177             return SessionIndex;
178         }
179     }
180
181     return -1; // g_SessionList is FULL!
182 }
183
184 void close_session( int SessionIndex )
185 {
186     // Closes an existing socket
187     closesocket( g_SessionList[SessionIndex].Socket );
188     // Delete critical Section
189     DeleteCriticalSection( &g_SessionList[SessionIndex].cs_SessionList );
190
191     printf( "Connection Closed. SessionIndex:%d, hThread:%d, ThreadID:%d\n",
192             SessionIndex, g_SessionList[SessionIndex].hThread,
193             g_SessionList[SessionIndex].dwThreadID );
194
195     // Reset the structure data.
196     memset( &g_SessionList[SessionIndex], '\0', sizeof(struct SessionInfo) );
197 }
198
199 DWORD WINAPI RecvThread( void *arg )
200 {
201     char RecvBuffer[DEFAULT_BUFLEN];
202     int RecvResult;
203     int SessionIndex;
204
205     SessionIndex = (int)arg;
206
207     printf( "New thread created for SessionIndex:%d, hThread:%d, ThreadID:%d\n",
208             SessionIndex, g_SessionList[SessionIndex].hThread,
209             g_SessionList[SessionIndex].dwThreadID );
210
211     while( 1 ) {
212         //-----
213         /// Receive and echo the message until the peer closes the connection
214         RecvResult = recv( g_SessionList[SessionIndex].Socket,
215                           RecvBuffer, DEFAULT_BUFLEN, 0 );
216         if( 0 < RecvResult )
217         {
218             // Copy the RecvBuffer into Session Buffer
219             EnterCriticalSection( &g_SessionList[SessionIndex].cs_SessionList );
220             memcpy( &g_SessionList[SessionIndex].MessageBuffer[0],
221                     &RecvBuffer, RecvResult );
222             g_SessionList[SessionIndex].MessageSize = RecvResult;
223             LeaveCriticalSection( &g_SessionList[SessionIndex].cs_SessionList );
224             printf( "Received from SessionIndex:%d, hThread:%d, ThreadID:%d\n",
225                     SessionIndex, g_SessionList[SessionIndex].hThread,
226                     g_SessionList[SessionIndex].dwThreadID );
227             printf( "Bytes received : %d\n",
228                     g_SessionList[SessionIndex].MessageSize );

```

```

229         printf( "Buffer received : %s\n",
230                 g_SessionList[SessionIndex].MessageBuffer );
231
232         /// TODO : Process the packet.
233     }
234     else if( 0 == RecvResult )
235     {
236         printf( "Connection closed\n");
237         break;
238     }
239     else
240     {
241         printf( "Recv failed: %d\n", WSAGetLastError() );
242         break;
243     }
244
245     /// Echo same message to client
246     RecvResult = send( g_SessionList[SessionIndex].Socket,
247                         g_SessionList[SessionIndex].MessageBuffer,
248                         g_SessionList[SessionIndex].MessageSize,
249                         0 );
250     if( SOCKET_ERROR == RecvResult )
251     {
252         printf( "Send failed: %d\n", WSAGetLastError() );
253     }
254 }
255
256     /// Close the session
257     close_session( SessionIndex );
258
259     return 0;
260 }
```

Exercise 2

- Multithreaded Socket Client

```

0 //-----
1 /// Multithread Socket Client
2
3 #ifndef WIN32_LEAN_AND_MEAN
4 #define WIN32_LEAN_AND_MEAN
5 #endif
6
7 #include <winsock2.h>
8 #include <stdio.h>
9
10 /// Link with ws2_32.lib
11 #pragma comment(lib, "Ws2_32.lib")
12
13 #define DEFAULT_BUFLEN 1024
14 #define DEFAULT_PORT    7890
15
16 SOCKET g_ConnectSocket = INVALID_SOCKET; // Socket to connect to server
17
18 DWORD WINAPI RecvThread( void *arg )
19 {
20     char      MessageBuffer[DEFAULT_BUFLEN]; // Buffer to recv from socket
21     int       Result;                      // Return value for function result
```

```
22
23     do
24     {
25         memset( MessageBuffer, '\0', DEFAULT_BUFLEN );
26         Result = recv( g_ConnectSocket, MessageBuffer, DEFAULT_BUFLEN, 0 );
27         if( 0 < Result )
28         {
29             printf( "\nBytes received : %d\n", Result );
30             printf( "Message received : %s\n", MessageBuffer );
31         }
32         else if( 0 == Result )
33         {
34             printf( "Connection closed\n" );
35             break;
36         }
37         else
38         {
39             printf( "Recv failed: %d\n", WSAGetLastError( ) );
40             break;
41         }
42         printf( "Enter messages : " );
43     } while( 1 );
44
45     return 0;
46 }
47
48 int main( void )
49 {
50     //-----
51     /// Declare and initialize variables.
52     WSADATA      wsaData;           /// Variable to initialize Winsock
53     sockaddr_in  ServerAddress;    /// Socket address to connect to server
54     HANDLE        hThread;
55     DWORD         dwThreadID;
56     char          MessageBuffer[DEFAULT_BUFLEN]; /// Buffer to recv from socket
57     int           BufferLen;        /// Length of the message buffer
58     int           Result;          /// Return value for function result
59     int           i;
60
61     //-----
62     /// 1. Initiate use of the Winsock Library by a process
63     Result = WSAStartup( MAKEWORD( 2, 2 ), &wsaData );
64     if( NO_ERROR != Result )
65     {
66         printf( "WSAStartup failed: %d\n", Result );
67         return 1;
68     }
69
70     //-----
71     /// 2. Create a new socket for application
72     g_ConnectSocket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
73     if( INVALID_SOCKET == g_ConnectSocket )
74     {
75         printf( "Error at socket(): %ld\n", WSAGetLastError( ) );
76         WSACleanup( );
77         return 1;
78     }
79
80     //-----
81     /// The sockaddr_in structure specifies the address family,
82     /// IP address, and port of the server to be connected to.
83     ServerAddress.sin_family = AF_INET;
```

```
84 // Connecting to local machine. "127.0.0.1" is the loopback address.  
85 ServerAddress.sin_addr.s_addr = inet_addr( "127.0.0.1" );  
86 ServerAddress.sin_port = htons( DEFAULT_PORT );  
87  
88 //-----  
89 // 3. Establish a connection to a specified socket  
90 if( SOCKET_ERROR == connect( g_ConnectSocket, (SOCKADDR*)&ServerAddress,  
91 sizeof(ServerAddress) ) )  
92 {  
93     closesocket( g_ConnectSocket );  
94     printf( "Unable to connect to server: %ld\n", WSAGetLastError( ) );  
95     WSACleanup( );  
96     return 1;  
97 }  
98 printf( "Connected to the server.\n" );  
99  
100 //-----  
101 // Create thread for receive message  
102 hThread = CreateThread( NULL,  
103     0,  
104     RecvThread, // Thread function  
105     0,           // Passing Argument  
106     0,  
107     &dwThreadID );  
108 printf("RecvThread created. Handle[%d], ThreadID[%d]\n", hThread, dwThreadID);  
109  
110 // Receive until the peer closes the connection  
111 while( 1 )  
112 {  
113     printf( "Enter messages : " );  
114  
115     for( i = 0; i < (DEFAULT_BUFLEN - 1); i++ )  
116     {  
117         MessageBuffer[i] = getchar( );  
118         if( MessageBuffer[i] == '\n' )  
119         {  
120             MessageBuffer[i++] = '\0';  
121             break;  
122         }  
123     }  
124     BufferLen = i;  
125  
126     // 4. Send & receive the data on a connected socket  
127     if( SOCKET_ERROR == (Result = send( g_ConnectSocket,  
128                               MessageBuffer,  
129                               BufferLen, 0 )) )  
130     {  
131         printf( "Send failed: %d\n", WSAGetLastError( ) );  
132         closesocket( g_ConnectSocket );  
133         WSACleanup( );  
134         return 1;  
135     }  
136     printf( "Bytes sent: %ld\n", Result );  
137 }  
138  
139 // 5. close & cleanup  
140 closesocket( g_ConnectSocket );  
141 WSACleanup( );  
142  
143 return 0;  
144 }
```

Exercise 3

- Thread synchronization

If you read the Exercise 1. Multithreaded Socket Server, there is thread synchronization with CRITICAL_SECTION. However, if you read the Exercise 2. Multithreaded Socket Client, there is not of any thread synchronization.

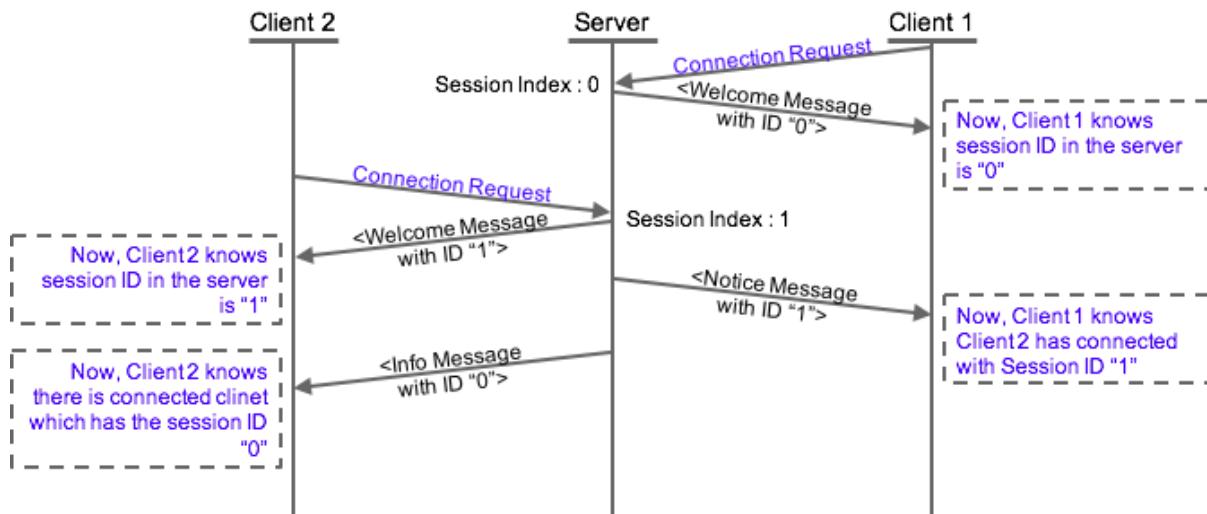
Why Client doesn't need thread synchronization?

Exercise 4

- Send the Welcome Message with the Session ID.

1. After your server accepts the client connection request, immediately send back the *Welcome Message* to the client with *Session ID* information. *The Session ID* is the index number of the *g_SessionList* array.
 - Welcome message : “<Welcome to my Multithreaded server! Your Session ID is ####>” (#### is the number of *Session ID*)
2. After sending *Welcome Message*, the server will send the *Info Message* immediately to the connected client. The *Info Message* has the information with the *Session ID* of all client which is already connected to the server.
 - Info message : “<Already connected client with session ID ####>” (#### is the number of *Session ID*)
3. And send the *Notice Message* below with the connected client session ID to the other connected clients which were already connected.
 - Notice message : “<New client has connected. Session ID is ####>” (#### is the number of *Session ID*)

※ Flow diagram is like below;



Exercise 5

- Send a message to one specific client.

Send the message to one specified client. The message needs to be changed with information of the sender. If send has success, the server will return back the success message to the message sender.

- Message input from the sender client :

“</### Message>”

- The server will convert the message and send to specific client (receiver) :

“<Sender @@@ sent : Message>”

- If success, the server will send the message to the sender :

“<Message sent to ###>”

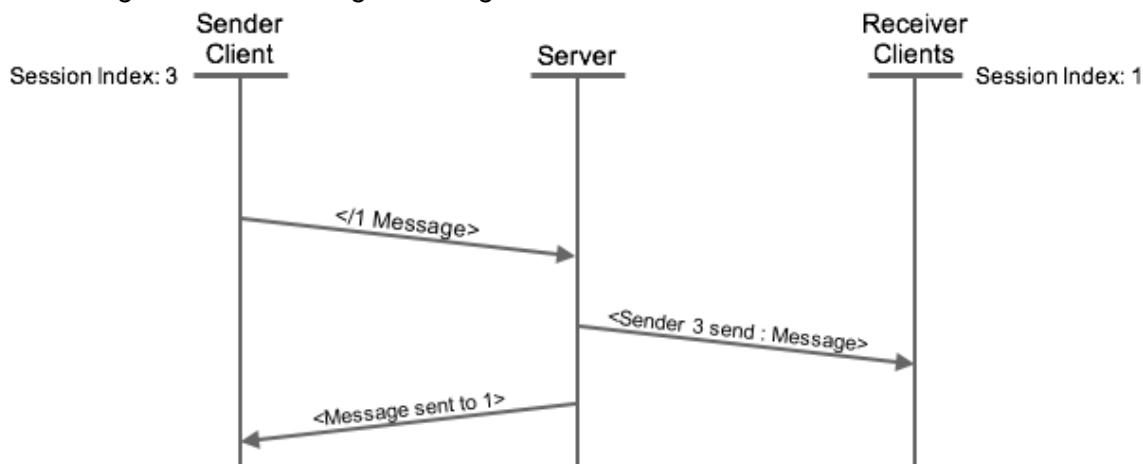
- If fail or error :

“<Fail to send to ###>”

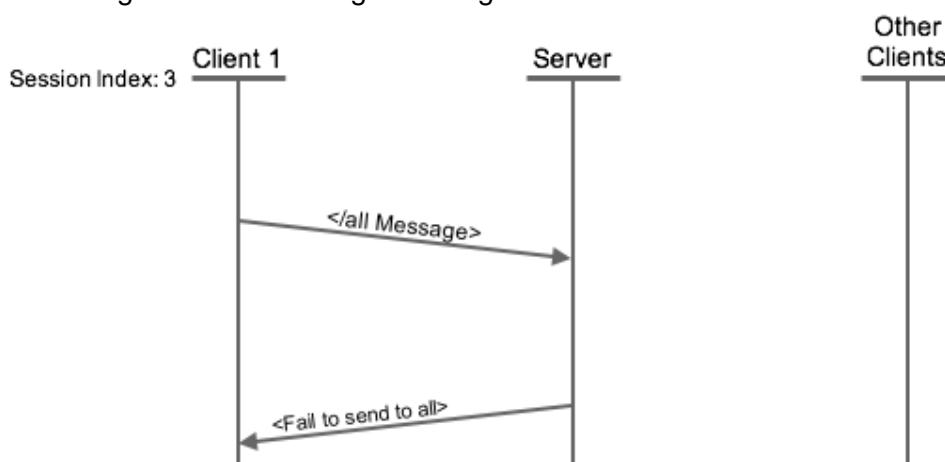
※ “###” is the Session ID of receiver client of SessionInfo structure.

※ “@@@” is the Session ID of sender client of SessionInfo structure.

※ Flow diagram if the message sending has success



※ Flow diagram if the message sending has failed



Exercise 6

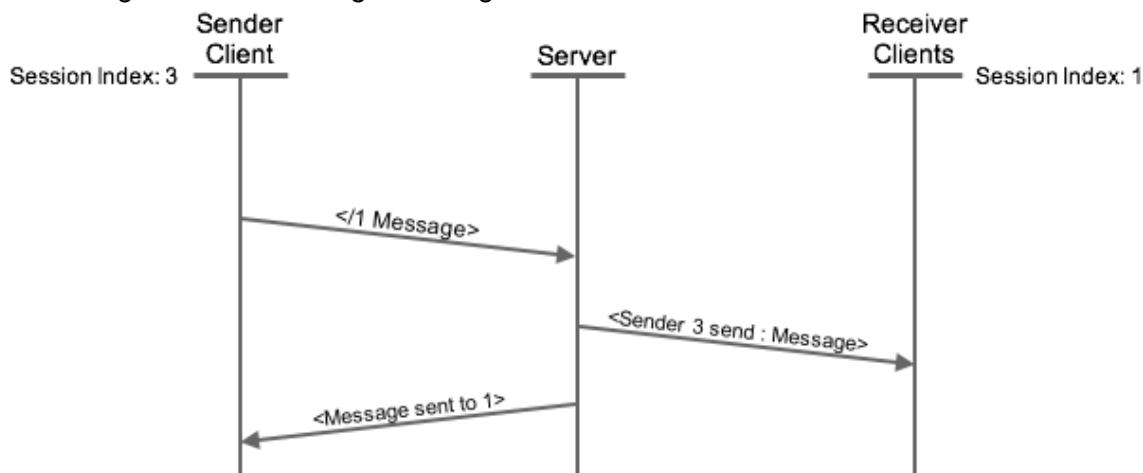
- Send a message to all connected clients.

Now it's time to send your message to everyone! Send your message to the server to announce to everyone already connected into the server.

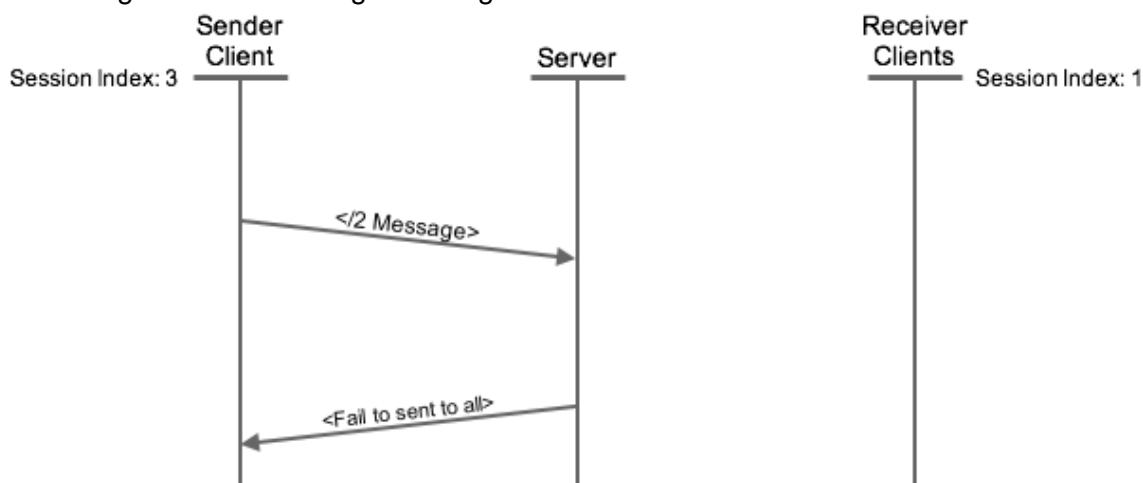
- Message input from the sender client :
“</all Message>”
- The server will convert the message and send to all clients :
“<Sender @@@ sent to everyone : Message>”
- If success, the server will send the message to the sender :
“<Message sent to all>”
- If fail or error :
“<Fail to send to all>”

※ “@@@” is the number of socket descriptor number (Session ID) of sender client in SessionInfo structure.

※ Flow diagram if the message sending has success



※ Flow diagram if the message sending has failed



Extra Exercises

You don't need to submit these exercises.

Extra Exercise 1

- *Thread synchronization with Mutex*

Update the Exercise 1. Multithreaded Socket Server with Mutex instead of CRITICAL_SECTION to synchronize multithread.

Extra Exercise 2

- *Thread synchronization in Client*

Without thread synchronization, this can be happen;

- during typing in the message from keyboard, if there is any message received from network, RecvThread() function will pick up the message and going to print on the screen. So the screen can be messed up.

Add thread synchronization into Exercise 2. Multithreaded Socket Client to separate the process of “receiving message from network & printing out one the screen” and “type in message from keyboard & send the message to network”.