Tutoral 7 – Exchange any size files between client and server

Contents:

1. Introduction

It is a common task to write code for exchanging files between client and server. SocketPro has a file streaming service implemented at server side so that you can reuse the service for downloading and uploading files in streaming style at the fastest speed. Since both client and server codes are extremely simple and understandable, you can easily extend them for your complex needs.

In addition, we are going to discuss SocketPro client and server threading models so that you can easily remember calling threads for various events and virtual functions.

The tutorial sample projects are located at the directory socketpro/tutorials/(csharp|vbnet|cplusplus|java\src|ce|python)/remote_file. The directory socketpro/samples/module_sample/remotefile/(test_client|test_java|test_sharp|test_python) contains test codes for exchanging multiple files in streaming style.
The Node.js sample is at the file ../socketpro/src/njadapter/remote_file.js.

2. SocketPro client and server threading models

SocketPro is created so that your developments are easy especially for dealing with threading. SocketPro tries its best to manage creating and killing as automatically as possible. As a developer, you are never required to manage various threads at both client and server side.

Server threading model: As described at the [tutorial one: *Hello world for a simple client/server application*](), SocketPro server has one of main threads that create and manage all worker thread on the fly at a proper time. In addition, the main thread dispatches all of client requests to obey the flow chart of the below Figure 1.

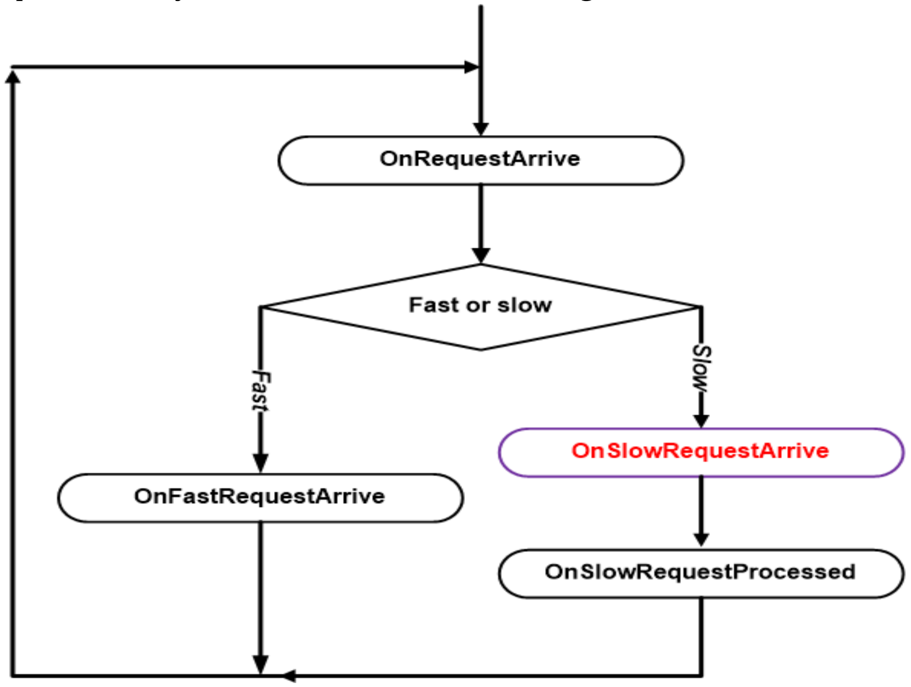


Figure 1: Dispatch chart of requests processing at server side

As shown in Figure 1 above, SocketPro server main thread only dispatches a slow request onto a worker thread for processing. How does the SocketPro server know if a request is a slow one? The answer is simple: when you call *AddSlowRequest* in C++ or set the *SlowRequest* attribute to true (for example, *[RequestAttr(hwConst.idSleepHelloWorld, true)])* right before a function, you tell the SocketPro server that the request is going to take a long time to process.

Another key point is that **all** virtual functions starting with *OnXXX* within whole name space *SocketProAdapter.ServerSide*, **except** the virtual function *OnSlowRequestArrive*, are processed within one of main threads. Only one, *OnSlowRequestArrive*, is called within a worker thread.

All the requests from one socket connection are always processed one-by-one sequentially at server side. From the view of a SocketPro server, all fast requests from all clients are processed within one of main threads.

Client threading model: Client threading model can be described with the below Figure 2.

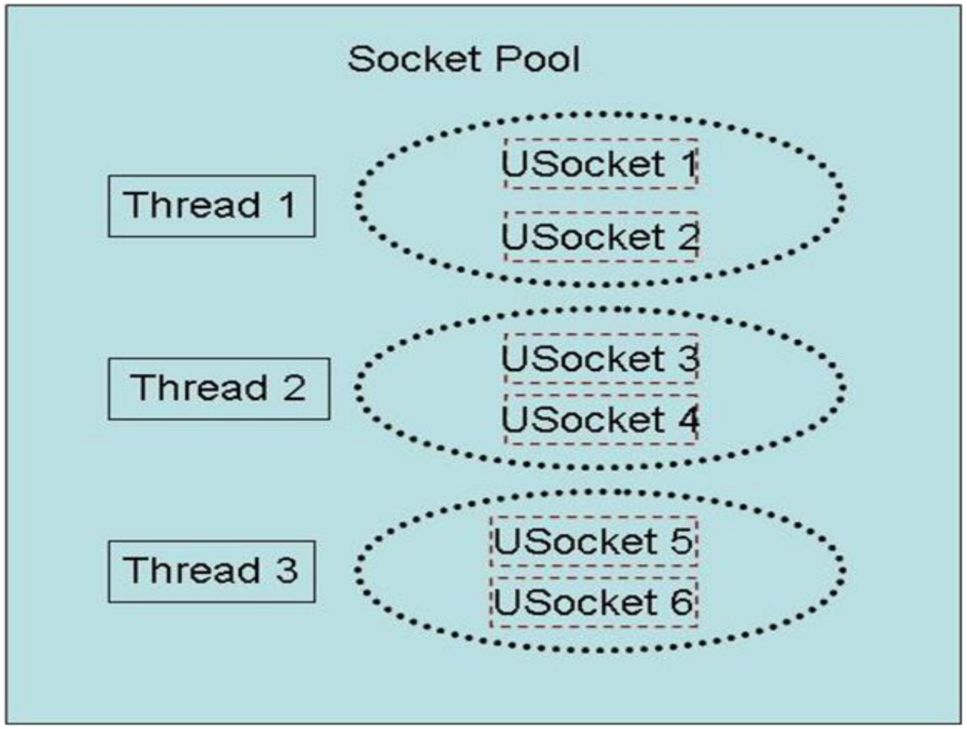
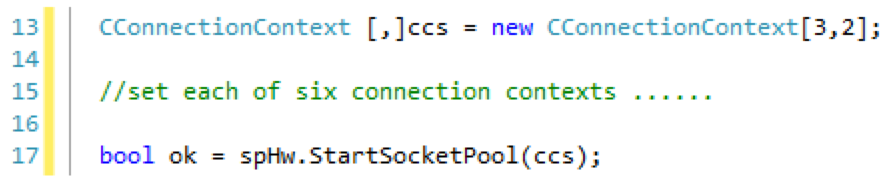


Figure 2: Socket pool having three worker threads hosting six non-blocking sockets

SocketPro client uses socket pool concept to manage threads and sockets driven by its hosting threads. Note that all sockets are using non-blocking styles for communicating data between client and server as depicted above. To create a pool of sockets as shown in Figure 2, you can do so using the follow code in C# shown below.



```
CConnectionContext [,]ccs = new CConnectionContext[3,2];

//set each of six connection contexts ......

bool ok = spHw.StartSocketPool(ccs);
```

Figure 3: Start a pool of sockets with three threads and two sockets per thread

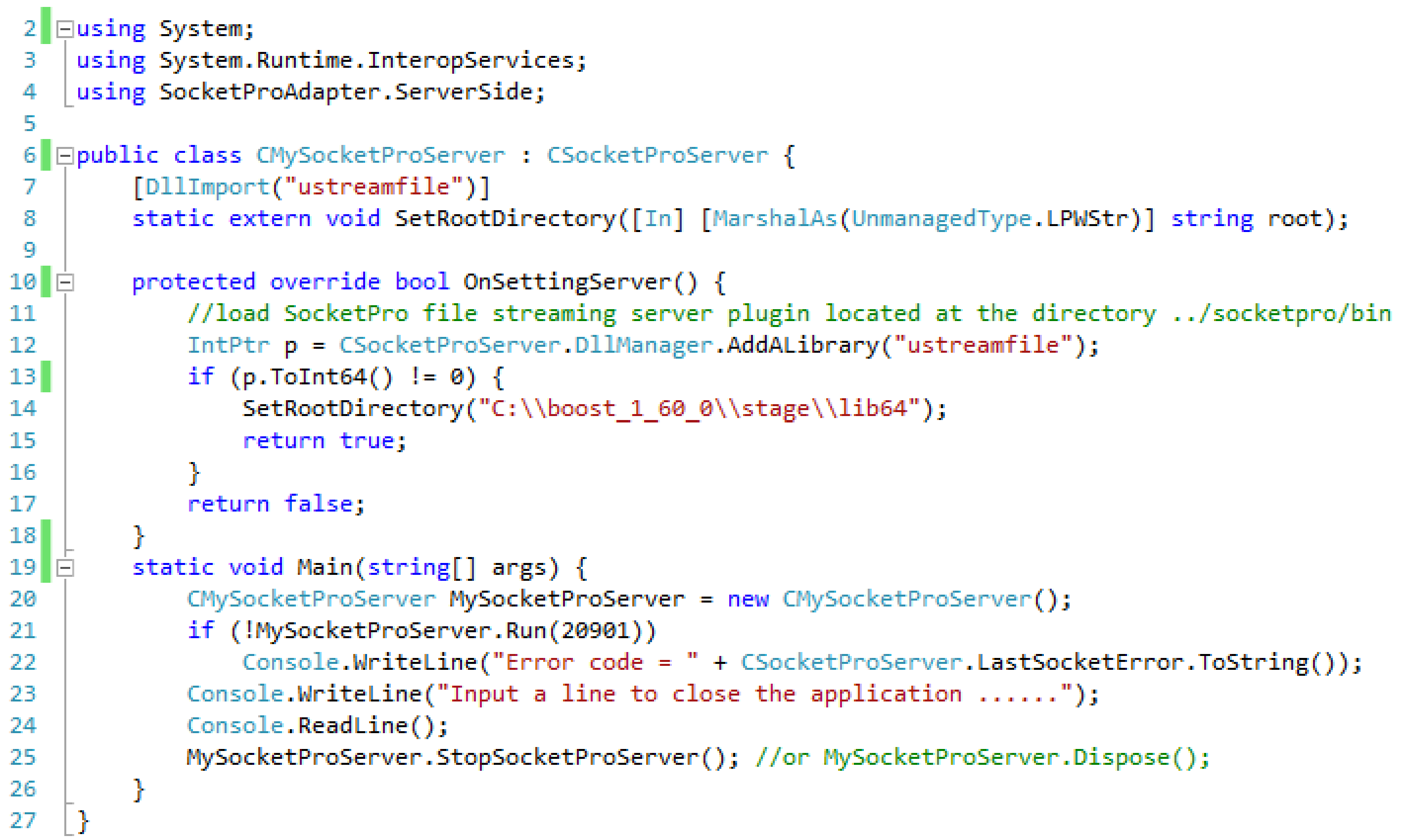
Note that each of the socket connection contexts can point to either the same server or different servers. When you destroy a socket pool, you will kill both their threads and sockets.

One key point is that **all** the socket events, SSL/TLS events, various events within the whole name space *SocketProAdapter.ClientSide* and returning data from server all originate from socket pool threads **except** the SocketPool events for *speCreatingThread*, *speThreadCreated*, *speKillingThread*, *speThreadKilled*, *speUSocketCreated*, *speUSocketKilled*, *speLocked* and *speUnlocked*. Theses eight events originate from your calling thread instead.

3. Start a SocketPro server with file streaming service

UDAParts has developed a file streaming service (ustreamfile.dll for windows or libustreamfile.so for Linux) to exchange files at the fastest speed between two machines in streaming style. Its C/C++ implementation codes can be found at the directory of socketpro/include/file/server_impl. It is noted that you can easily customize or extend it for your needs as both server and client implementations are extremely simple for you to understand. At client side, there is a simple utility class CStreamingFile inside adapter.

The code snippet in Figure 4 shows you how to start a SocketPro server with file streaming service. As shown in the Figure 4, you can load the file streaming service library into a SocketPro server by DllManager as shown at line 12 before starting a listening socket. Afterwards, you may want to configure a root directory like HTTP server root directory as shown at line 14. To make test successfully, you are required to modify the root directory, before compiling the test project and running the server application.



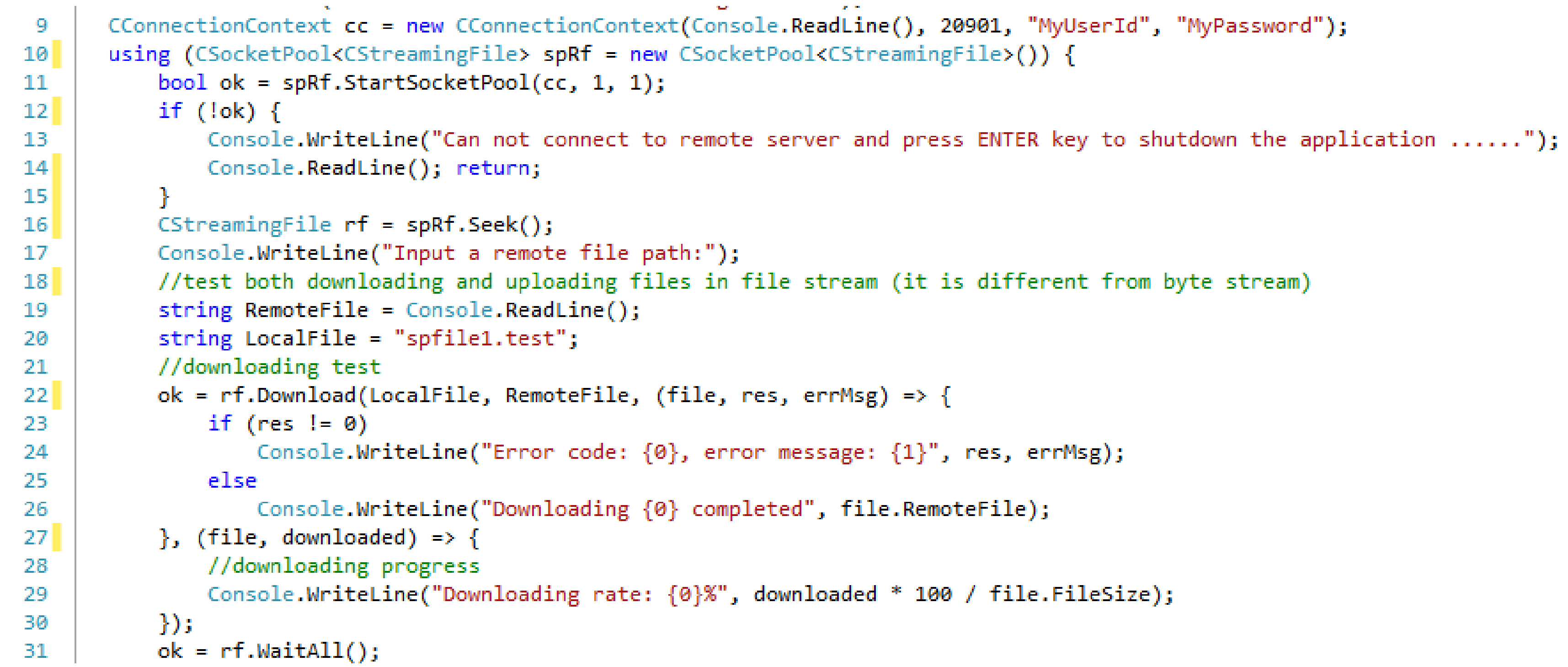
```
using System;
using System.Runtime.InteropServices;
using SocketProAdapter.ServerSide;

public class CMySocketProServer : CSocketProServer {
    [DllImport("ustreamfile")]
    static extern void SetRootDirectory([In] [MarshalAs(UnmanagedType.LPWStr)] string root);

    protected override bool OnSettingServer() {
        //load SocketPro file streaming server plugin located at the directory ../socketpro/bin
        IntPtr p = CSocketProServer.DllManager.AddALibrary("ustreamfile");
        if (p.ToInt64() != 0) {
            SetRootDirectory("C:\\boost_1_60_0\\stage\\lib64");
            return true;
        }
        return false;
    }
    static void Main(string[] args) {
        CMySocketProServer MySocketProServer = new CMySocketProServer();
        if (!MySocketProServer.Run(20901))
            Console.WriteLine("Error code = " + CSocketProServer.LastSocketError.ToString());
        Console.WriteLine("Input a line to close the application ......");
        Console.ReadLine();
        MySocketProServer.StopSocketProServer(); //or MySocketProServer.Dispose();
    }
}
```

Figure 4: Start a SocketPro server with file streaming service

4. Exchange a file between client and server

It is time to look at client side code. All SocketPro adapters come with helper class CStreamingFile for helping client side development.

First, let’s see how to download one file from a remote SocketPro server as shown at line 22 of the below Figure 5.

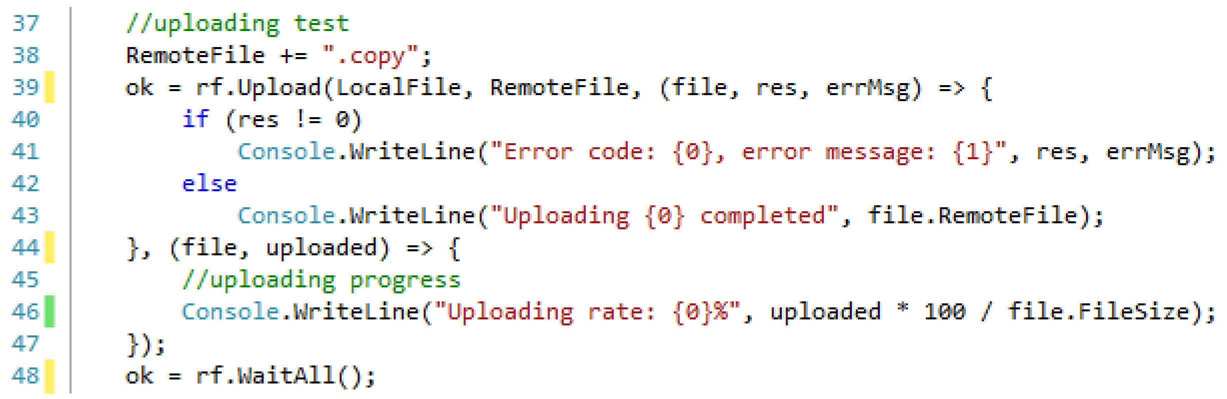


```
 9  CConnectionContext cc = new CConnectionContext(Console.ReadLine(), 20901, "MyUserId", "MyPassword");
10  using (CSocketPool<CStreamingFile> spRf = new CSocketPool<CStreamingFile>()) {
11      bool ok = spRf.StartSocketPool(cc, 1, 1);
12      if (!ok) {
13          Console.WriteLine("Can not connect to remote server and press ENTER key to shutdown the application ......");
14          Console.ReadLine(); return;
15      }
16      CStreamingFile rf = spRf.Seek();
17      Console.WriteLine("Input a remote file path:");
18      //test both downloading and uploading files in file stream (it is different from byte stream)
19      string RemoteFile = Console.ReadLine();
20      string LocalFile = "spfile1.test";
21      //downloading test
22      ok = rf.Download(LocalFile, RemoteFile, (file, res, errMsg) => {
23          if (res != 0)
24              Console.WriteLine("Error code: {0}, error message: {1}", res, errMsg);
25          else
26              Console.WriteLine("Downloading {0} completed", file.RemoteFile);
27      }, (file, downloaded) => {
28          //downloading progress
29          Console.WriteLine("Downloading rate: {0}%", downloaded * 100 / file.FileSize);
30      });
31      ok = rf.WaitAll();
```

Figure 5: Download a file from server to client with given local and server file paths

In addition to local and server file paths, we can also input two optional callbacks or Lambda expressions to catch downloading error message from line 23 through 26 and file downloading progress at line 29, respectively. It cannot be simpler!

For file upload, its code is nearly the same as file download as shown at line 39 in the below Figure 6.



```
//uploading test
RemoteFile += ".copy";
ok = rf.Upload(LocalFile, RemoteFile, (file, res, errMsg) => {
    if (res != 0)
        Console.WriteLine("Error code: {0}, error message: {1}", res, errMsg);
    else
        Console.WriteLine("Uploading {0} completed", file.RemoteFile);
}, (file, uploaded) => {
    //uploading progress
    Console.WriteLine("Uploading rate: {0}%", uploaded * 100 / file.FileSize);
});
ok = rf.WaitAll();
```

Figure 6: Upload a file from client to server with given local and server file paths

5. Exchanging files in SocketPro streaming style

The above Figures 5 and 6 shows how to download and upload just one file only for demonstration code clarity. In case you want to download, upload or download/upload many files, you can call the methods Download and Upload ***continuously*** before a previous file is completely exchanged so that all files are flowed into the other side in streaming style. This is different from traditional approach.

Typically, for example, you could send a request from a client to a remote server to check if a file exists at server side first. Afterwards, if the file indeed exists, you send new requests to download chunks of data back and forth until all data are copied into a client side. Once the first file is completed, you repeat doing the same things for the other files.

SocketPro doesn't use the above approach. Instead, we call the method Download continuously to queue requests (downloading all files) without waiting any returning data from server to client. Once a SocketPro server receives download requests in stream, the server will first check if a file exists according to each context of streamed download requests. If the file doesn't exist, SocketPro server returns an error message (an error code and error text message). If the file does exist, SocketPro server reads the file data in chunks, and push these chunks onto its client continuously until all chunked data are completed. There is no request from client to server for chunk downloading during pushing data, although the client can still send other requests to the server. SocketPro file streaming style reduces not only the number of ***data packet trips*** but also degrades the bad ***influence of latency*** on network throughput. If you understand SocketPro file stream style clearly, you will find it will be extremely efficient for exchanging ***small files on wide area network***. SocketPro is still able to easily improve large file exchange speed by 50% on local area networks.

To know SocketPro file streaming style more, you can use the source code at the directory socketpro/samples/module_sample/remotefile/(test_client|test_java|test_sharp|test_python) for experimenting. For Node.js, see the file ../socketpro/src/njadapter/remote_file.js.