|  |
| --- |
| UDAParts |
| Development guide for SocketPro communication framework |
|  |

|  |
| --- |
| Yuancai (Charlie) Ye  10/25/2016 |

Table of Contents

[1. Version history 4](#_Toc465247280)

[2. Audiences 4](#_Toc465247281)

[2.1 Basic knowledge requirements 4](#_Toc465247282)

[2.2 Audience types 4](#_Toc465247283)

[3. Introduction 4](#_Toc465247284)

[4. SocketPro data transferring pattern 5](#_Toc465247285)

[4.1 One request for one response or result 5](#_Toc465247286)

[4.2 Multiple requests for multiple responses or results 6](#_Toc465247287)

[5. Implementation approach 7](#_Toc465247288)

[5.1 Must-have requirements 7](#_Toc465247289)

[5.2 Optional but highly recommended features 7](#_Toc465247290)

[5.3 Technologies used for windows implementation 7](#_Toc465247291)

[5.4 Technologies used for non-windows implementation 7](#_Toc465247292)

[6. Client and server communication agreements 8](#_Toc465247293)

[6.1 Internal communication protocol 8](#_Toc465247294)

[5.1.1 Request or command structure 8](#_Toc465247295)

[5.1.2 Reserved request Ids 8](#_Toc465247296)

[6.2 idAuthentication and idAuthenticationReserved 9](#_Toc465247297)

[6.3 idServerException 9](#_Toc465247298)

[6.4 idHeartBeat 9](#_Toc465247299)

[6.5 Windows implementation approach 9](#_Toc465247300)

[7. Defines, structures and interfaces shared by both client and server 10](#_Toc465247301)

[7.1 Error codes 10](#_Toc465247302)

[7.2 Session states 10](#_Toc465247303)

[7.3 Interface ISession 10](#_Toc465247304)

[7.4 Interface ICertificate 11](#_Toc465247305)

[7.5 Callback definitions 11](#_Toc465247306)

[7.6 Session callback structure 11](#_Toc465247307)

[8. Server side 12](#_Toc465247308)

[8.1 Goals 12](#_Toc465247309)

[8.2 Achievements 12](#_Toc465247310)

[8.3 Interface IServerSession and one callback 12](#_Toc465247311)

[8.4 Three C functions 12](#_Toc465247312)

[9. Client side 13](#_Toc465247313)

[9.1 Goals 13](#_Toc465247314)

[9.2 Achievements 13](#_Toc465247315)

[9.3 Interface IClientSession 13](#_Toc465247316)

[9.4 Three C functions 14](#_Toc465247317)

[10. Client and server adapters as well as unit test applications 14](#_Toc465247318)

[10.1 Purposes 14](#_Toc465247319)

[10.2 Server adapter and client unit test code snippet 14](#_Toc465247320)

[10.3 Client adapter and server unit test code snippet 15](#_Toc465247321)

# Version history

The table below records major changes to this document

|  |  |
| --- | --- |
| **Date** | **Comment** |
| 2016-05-13 | Initial |
|  |  |
|  |  |

# Audiences

## Basic knowledge requirements

Audiences are not expected to be professionals to the below terminologies, but it is expected that all of audiences should have general knowledge about them:

* Client and server communication architecture
* Persistent message queue architecture
* Online message publish-subscribe architecture
* Message replication
* Request load balancing and routing
* HTTP and web socket protocols
* Socket, latency, throughput and network bandwidth
* TCP/IP protocol for plain text communication
* SSL/TLSv1 protocol for encryption and decryption
* SSL/TLSv1 certification verification at client side
* C API functions and interfaces

## Audience types

* Software architect
* Software developer
* Software development manager

# Introduction

Existing AutoSave communication was implemented by heavy use of old MFC socket classes at both client and server sides long time ago. It has a number of the following fundamental flaws from today’s view:

* No support on SSL.
* No support on Unicode string.
* Poor code readability and maintenance with too old code style.
* Having troubles to support .NET code in the future because of MFC dependency.
* No support on x64 applications.
* No support on direct running client applications right after inserting memory stick into a window PC machine.
* No support on IPv6.

The coming new communication libraries are expected to solve all the above problems. New libraries should have clear threading models with simplicity for easy reuse by other libraries.

There will be two highly reusable libraries to be created. One is running at client side; and the other at server side as they will be addressed at the following section 4.

# SocketPro data transferring pattern

SocketPro framework is created on non-blocking TCP/IP socket communication to support concurrently transferring data between two end points with best network efficiency by continuous inline data batching at both client and server sides. To help you understand SocketPro data transferring pattern more clearly, please pay close attention to the below picture with your careful analyses.

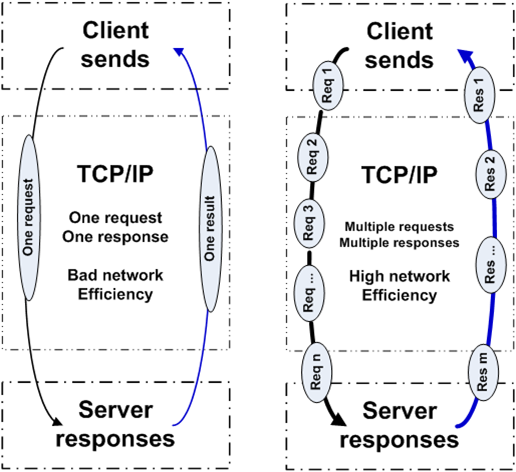


Figure 1: Two data transferring patterns

## 4.1 One request for one response or result

The above picture shows two types of data transferring patterns. First of all, let’s have a close the left one. It actually represents a typical synchronous communication pattern by use of blocking TCP/IP socket, which is most widely used by most of distributed application systems.

It is noted that many applications may use a worker thread at client side to send a request and wait for a result from a server to fake an asynchronous communication. However, it is not considered by UDAParts as a true asynchronous communication because the communication pattern is just a typical pattern of one-request for one-result. Therefore, UDAParts thinks this fake asynchronous communication is still a synchronous one, whose communication pattern is actually the above left one. In comparison to the above first scenario, this one does not block this calling thread as it is freed by a worker thread although the approach has flaws of extra thread context switches and worker thread manipulations. The two scenarios are very popular as they are easy to be understood and implemented by all software engineers, architectures and project managers.

Currently, a few distributed application systems do use non-blocking TCP/IP sockets without use of any worker thread for asynchronous communications. However, most of them still follow one-request for one-result for processing. Each of many requests is one-by-one sent from client, processed at server side, and finally returned from server to client with an expected result. In comparison to the above two scenarios, this approach has two advantages, no calling thread blocking and no expensive thread-context switch. However, it still follows the pattern of one-request for one-result.

All the above scenarios have two fundamental issues, low network efficiency and lack of concurrency in client requests sending and processing. First of all, LAN (local area network) has a typical latency between 0.1 and 0.4 millisecond. In other words, one TCP/IP socket will never exceed 10,000 or 2,500 requests per second no matter either 1 or 10 GB switches are used. Further, most of requests and their corresponding results are smaller than 1460 in bytes, which leads to huge waste of network bandwidth. Assuming the above assumptions are correct and each of requests has a size of 1460 bytes (Note that requests could be around a few bytes in reality under many situations), each of sockets will not have a throughput of 14,600,000 (efficiency = 12% (14.6/120 megabytes) 1 GB switches) or 3,650,000 (efficiency = 0.3% (= 3.65/1200 megabytes) for 10 GB switches) bytes per second. You can see this pattern has extremely low network efficiency from view of a single socket. To solve the low network efficiency, it is very common to open multiple sockets for improving it. Besides, very few modern communication frameworks such as Apache Kafka, Apache Spark, Apache Storm employ batching multiple requests at client side and send them in one shot at a predefine time interval before putting a bigger chunk into a TCP/IP socket. If we consider a data transferring path, it is not difficult for us to find a flaw that devices on the whole path cannot run concurrently at any time on one TCP/IP socket. This is actually an issue as it degrades the total performance of a distributed application system.

As the above fundamental issues are deeply understood by UDAParts, UDAParts has spent a long time to overcome them completely with a much more powerful communication pattern. Please keep on reading and thinking the right side communication pattern of the figure 1.

## 4.2 Multiple requests for multiple responses or results

SocketPro framework is created with

# Implementation approach

## Must-have requirements

* Both client and server libraries must support the communication protocol as described at the sections 5.1 through 5.4.
* Client library must expose a function in some ways to verify certificate sourced from server side before sending any sensitive data. There is no certificate verification at server side at this writing time.
* SSL/TLSv1 version 3.0 or higher is required for both client and server libraries.

## Optional but highly recommended features

* Three communication events, SSL handshake done, data arrive, and session disconnection.
* Non-blocking communication with full support on blocking request at higher level by calling the method like WaitAll at client side.
* Expose session state information such as error code, error message, session connection state and endian-difference as well as others.
* PostClose or Close method at client, server or both sides to close a communication session.
* Both client and server libraries should have dependencies as low as possible.
* Both client and server libraries should be highly reusable friendly within multi-threading environments.
* Auto session reconnection after disconnection.
* Session connecting timeout at client side.
* Both client and server libraries should be converted onto other development environments (for example, .NET).
* Request or command receiving timeout at client side.

## Technologies used for windows implementation

* Meet all requirements listed at the section 4.1.
* Visual C++ will be used for windows environment development tool.
* C++11 new features.
* Openssl libraries for SSL/TLSv1 encryption and description.
* BOOST ASIO communication framework.
* Window socket 32-bit and 64-bit system libraries.
* Both client and server libraries are standard system libraries exposing C structures and functions as well as interfaces only. No classes are permitted to be exposed from the standard system libraries, although implementations at client and server libraries may heavily use C++ classes and C++ 11 new features as well as other advanced features internally.
* All optional features listed in the section 4.2, except the last one, are implemented for window platforms.
* Other minor features.

## Technologies used for non-windows implementation

* Must meet all the basic requirements as listed at the section 4.2.
* Open to any acceptable development environments and tools as well as frameworks
* Open to any acceptable SSL/TLSv1 supported libraries, but Openssl libraries are highly recommended.
* Features listed at the above section 4.3 are highly recommended, although these features are not must-have ones.

# Client and server communication agreements

## Internal communication protocol

### 5.1.1 Request or command structure

Whenever a client sends a request (command), we could use the following eight-byte structure to clearly describe it. We could also use the exactly same structure to describe returning result from server side.

*#pragma pack(push,1)*

*typedef struct CRequestHeader {*

*CRequestHeader()*

*: RequestId(0),*

*Reserved(0),*

*Size(0) {*

*}*

*void Reset() {*

*RequestId = 0;*

*Reserved = 0;*

*Size = 0;*

*}*

*unsigned short RequestId; //A request identification number, which can* ***NOT*** *be zero*

*unsigned short Reserved; //Reserved for the future use (online compression and decompression)*

*unsigned int Size; //Request chunk data size in bytes*

*} RequestHeader;*

*#pragma pack(pop)*

We could use the request id to identify what request is at server side and what result is at client side. The member Reserved is reserved and always set to zero at the moment. We may use it to support online compression and decompression in the future to accelerate file or database record set transferring between client and server with low bandwidth network. The last member Size is used to indicate the size of request data chunk in bytes. As you can see, we could always retrieve the first eight bytes of data for the structure first. Afterwards, we could always retrieve the coming request or result data chunk from the structure member Size at both client and server sides.

Note that request id can NOT be zero. Therefore, the implementation of both client and server libraries should ensure there is no request transferred with request id equal to zero between client and server.

### 5.1.2 Reserved request Ids

To make the protocol work correctly, we also reserved the following four predefined request ids for different purposes:

*static const unsigned short idAuthentication = 1;*

*static const unsigned short idServerException = 2;*

*static const unsigned short idHeartBeat = 3;*

*static const unsigned short idAuthenticationReserved = 256;*

* 1. idAuthentication and idAuthenticationReserved

The id *idAuthentication* is used to ensure that a client must always send the client credentials to server before sending any other requests right after both TCP/ICP and SSL/TLSv1 handshakes are completed. This rule is set for better security purpose

The second id *idAuthenticationReserved* is used to detect endian difference between client and server. When a client sends an authentication request to a server, if the server gets this reserved id instead of *idAuthentication*, we are ensured that the client and server have different endians at server side. Similarly, the client will know that its endian is different from its server endian right after seeing returning authentication result. Note that we may not meet endian-difference situation in the near future, but we consider it ahead. Also, both client and server libraries ensure there is no possibility to send this particular request from other software components. Otherwise, libraries will be confused with endian-difference.

* 1. idServerException

This id is used to transfer the following data from server to client in the following predefined order:

* A two-byte unsigned short for current request id
* A four-byte integer for anyone of error codes
* An ASCII string with a four-byte integer length indicator ahead for error message
* An ASCII string with a four-byte integer length indicator ahead for stack

Therefore, you could use the above information as a base to build error logging file for runtime debugging at client, server or both sides.

* 1. idHeartBeat

This id is used to detect abnormal session disconnection. At client side, client library should always send an empty request having this request id with zero size at a predefine interval (TBD), whenever there is no data transaction between client and server during the interval period. Server library will use this logic to detect abnormal client detection if there is no data received during the predefined interval period. Similarly, client library is able to do the same detection.

Note that both client and server are able to detect normal session disconnection at real time fashion, which is highly preferred, although this is not required for both client and server libraries.

## Windows implementation approach

From now on until the end of this document, it is focused on windows communication implementation only, which is realized from asynchronous communication style with help of boost asio framework for easy and fast development. In regards to other platforms, other implementations may refer to the windows platform implementation, but do not have to follow this implementation exactly as long as they abide by the rules defined at sections 4.1 and 5.1 through 5.4.

It is highly recommended that you should read through the following two boost ASIO doc sites:

* [*http://www.boost.org/doc/libs/1\_55\_0/doc/html/boost\_asio/overview/core/basics.html*](http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/overview/core/basics.html)
* [*http://www.boost.org/doc/libs/1\_55\_0/doc/html/boost\_asio/overview/core/async.html*](http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/overview/core/async.html)

# Defines, structures and interfaces shared by both client and server

All defined constancies, structures and interfaces in this section are declared at the file sccomm.h. They are shared by both client and server core libraries.

## Error codes

There are a few defined constancies with comments, as described at the following.

*//error codes for the method Retrieve of interface ISession*

*//and the method Send of interfaces IClientSession and IServerSession*

*//as well as the method SendException of interface IServerSession*

*static const unsigned int SESSION\_CLOSED = (~0); //session closed*

*static const unsigned int BAD\_REQUEST = SESSION\_CLOSED - 1; //Unexpected request id at client side*

*static const unsigned int BAD\_RETRIEVE\_THREAD = BAD\_REQUEST - 1; //Retrieve data from wrong thread*

*static const unsigned int BAD\_RESULT = BAD\_REQUEST; //Send result with wrong request id*

They are used with the methods *Retrieve* and *Send* as well as *SendException*.

## Session states

There are a few defined session states as described at the following. These sessions are obvious to everyone.

*typedef enum tagSessionState {*

*ssClosed = 0,*

*ssClosing = 1,*

*ssConnecting = 2,*

*ssConnected = 3,*

*ssSslHandshaking = 4,*

*ssSslHandshaked = 5,*

*ssAuthentcated = 6*

*} SessionState;*

Note that session state would be at the session state *ssAuthenticated* after a client request *idAuthentication* is sent and processed.

## Interface ISession

We define a basic interface with a number of methods shared between client and server sides. Its major methods are listed as the below with ignoring obvious ones.

*struct ISession {*

*//Check the data size in bytes to be sent to remote peer.*

*//If it returns a large vaule, it means network bandwidth is not matchable for sending speed*

*virtual unsigned int GetSendingBufferSize() = 0;*

*//Set a buffer recv with length size in bytes to receive error message terminated by null*

*virtual unsigned int GetErrMsg(char \*recv, unsigned int size) = 0;*

*//Set a buffer recv with length size in bytes to receive data*

*//Must call this method within the thread hosting boost asio io service object*

*//The method returns the actually obtained data length or BAD\_RETRIEVE\_THREAD*

*virtual unsigned int Retrieve(unsigned char \*recv, unsigned int size) = 0;*

*};*

Both client and server libraries implements the interface to support reusing it within multi-threading friendly and easily. Note that we may add new methods into the interface without breaking code compatibility.

## Interface ICertificate

We also define an interface to access SSL/TLSv1 certificate for its verification before sending any sensitive data. Note that we may add new methods into the interface without breaking code compatibility.

Its major methods are listed as the below with ignoring obvious ones.

*struct ICertificate {*

*virtual const char\* const Verify(int \*errCode) = 0;*

*virtual const char\* const GetCertPem() = 0;*

*virtual const char\* const GetSessionInfo() = 0;*

*virtual bool IsValid() = 0;*

*virtual const char\* const GetSubject() = 0;*

*virtual const char\* const GetIssuer() = 0;*

*virtual const unsigned char\* const GetPublicKey(unsigned int \*pKeySize) = 0;*

*};*

Note that we have implemented the interface at client side as required at the section 4.1, but not at server side.

## Callback definitions

We also define the following four obvious callbacks to track session events shared by both client and server libraries. Note the namespace SC stands for secured communication just for your information.

*typedef void (CALLBACK \*POnArrive) (SC::ISession \*session, unsigned short RequestId, unsigned int Size);*

*typedef void (CALLBACK \*POnClose) (SC::ISession \*session);*

*typedef void (CALLBACK \*POnLess) (SC::ISession \*session);*

*typedef void (CALLBACK \*POnSslHandshakeDone) (SC::ISession \*session);*

## Session callback structure

At last, we define a structure which will be passed into both client and server core libraries so that its callback functions, which are actually implemented from a calling library, could be correctly called at a proper time.

*struct SessionCallback {*

*//!!!! must initialize the following callbacks*

*POnSslHandshakeDone OnSslHandshakeDone; //SSL handshake done event*

*POnArrive OnArrive; //command or result arrive event*

*POnClose OnClose; //session disconnection event*

*POnLess OnLess; //sending buffer no data event*

*};*

# Server side

## Goals

The server library is a window standard system library with exposing three C functions at this writing time. A server is able to accept and support multiple sessions and make sure these sessions run concurrently without blocking each others from client view. The server library is able to be easily reused from .NET environment.

## Achievements

The server library is created to replace the classes *CListensoc* (*CListen60soc*) and *CConectSoc* of existing project *UcsdServer*. The future class *CListenServer* will not be derived from the *MFC* class *CWinThread*. We’ll totally get rid of *MFC* from both client and server libraries.

## Interface IServerSession and one callback

In addition to the interface ISession, we also define a new interface IServerSession from the interface ISession for specific session methods of server side.

*struct IServerSession : public ISession {*

*//Send a result chunk buffer with length size onto remote client for a request ReqId*

*//It returns the actually length data, SESSION\_CLOSED, or BAD\_RESULT*

*virtual unsigned int Send(unsigned short ReqId, const unsigned char \* const buffer, unsigned int size) = 0;*

*//Send an exception (ec, msg and stack) onto remote client for current request CurReqtId*

*//It returns the actually length data in bytes, SESSION\_CLOSED, or BAD\_RESULT*

*virtual unsigned int SendException(unsigned short CurrReqId, int ec, const char \*msg, const char \*stack) = 0;*

*//Tell the underlying server core library that we are going to use a worker thread for processing*

*//Must call this method within main thread before starting a worker thread. Otherwise, it returns false.*

*virtual bool StartThreadProcessing() = 0;*

*//Tell the underlying server core library that worker thread for processing is completed*

*//Must call this method from a worker thread. Otherwise, it returns false or no processing for this session.*

*virtual bool EndThreadProcessing() = 0;*

*};*

The following callback is called by listening socket when a socket session is initialized at server side. Server side will return a structure for four callbacks, which are implemented from a calling server library or application, so that server will use the four callbacks to notify events at proper times. By the same time, a calling library or application will record an IServerSession interface to access session at server side.

*typedef SC::SessionCallback (CALLBACK \*POnAccepted)(SC::ServerSide::IServerSession \*session);*

## Three C functions

At this writing time, the server library only exposes the following three C functions.

const char\* WINAPI InitializeSecureCommServer(POnAccepted OnAccepted, unsigned int port = 22260, bool v6 = false, unsigned int backlog = 16);

const char\* WINAPI RunSecureCommServer(const char \*certFile, const char \*keyFile, const char \*pwdForPrivateKeyFile, const char \*dhFile = nullptr);

void WINAPI ShutdownSecureCommServer();

The first two functions will return an error message if available.

In short, we need to call the first method *InitializeSecureCommServer* with a callback *OnAccepted* that is implemented within either calling library or application. Afterwards, we call the method *RunSecureCommServer* with SSL-related files to start the secure socket server. When a client establishes a socket connection, the callback *OnAccepted* will be called from the server library. Referring to the definition *POnAccepted* at the section 7.3, you can get an interface *IServerSession* to the newly established session. In addition, you have to return a structure for session callbacks that are implemented within calling library or server application. Internally, the server library will call your codes through the callback structure at proper times.

As expected at the end, call the method *ShutdownSecureCommServer* to kill the internal listening socket and its associated sessions if available.

# Client side

## Goals

The client library is also a window standard system library with exposing three C functions at this writing. The client library also implements a number of callbacks to notify common socket session events to any calling 32-bit and 64-bit libraries or application. Also, the client library is able to be easily reused from .NET environment to take advantage of .NET new key words async and await.

## Achievements

The client library is created to replace the *MFC* class *CSocket*. Like server library, we don’t use *MFC* within the new client library at all.

## Interface IClientSession

Similar to the server interface *IServerSession*, we derive a new interface IClientSession from the base interface ISession for specific methods of client session as described at the following.

*struct IClientSession : public ISession {*

*//Wait until all queued requests are processed, timeout or session closed.*

*//The method returns true only if all queued requests are processed.*

*virtual bool WaitAll(unsigned int ms) = 0;*

*//Send a data chunk buffer with length size onto remote server for a request RequestId*

*//it returns the actually length data, SESSION\_CLOSED, or BAD\_REQUEST*

*virtual unsigned int Send(unsigned short RequestId, const unsigned char \* const buffer, unsigned int size, bool oneWay) = 0;*

*virtual bool IsAutoConnecting() = 0;*

*virtual size\_t GetRequestCountQueued() = 0;*

*virtual ICertificate\* GetCertificate() = 0;*

*};*

The interface is defined with detailed comments for key methods. The method *WaitAll* is used to easy convert asynchronous requests into synchronous if proper for simplifying code logic.

If you set the last input parameter *oneWay* to true for the method *Send*, this request will be one way and ignored at client side if you call the method *WaitAll*.

The method *IsAutoConnecting* is defined here to support auto reconnection.

## Three C functions

Client core library just exposes three C functions only as listed at the below.

*//certificate verification*

*int WINAPI SetCertificateVerifyFile(const char \*caFile);*

*//manage socket life*

*SC::ClientSide::IClientSession\* WINAPI DoClientConnection(SC::SessionCallback sc, const char \*host, unsigned int port, unsigned int timedout, bool autoConnecting, bool v6);*

*void WINAPI DestroyClient(SC::ClientSide::IClientSession \*session);*

The first function is used to set CA (Certificate Authority) certificate file path for certificate verification at client side. The last two functions are used to manage socket session life. When you make a call *DoClientConnection*, you get an interface *IClientSession* after you set a structure for four callbacks that are implemented at calling libraries or application. The other parameters are obvious. When the session is no longer needed, we simply call the method *DestroyClient* to kill it.

# Client and server adapters as well as unit test applications

## Purposes

After reading through the sections 7.4 and 8.4, you may get ideas how the two core libraries work in general. However, you may still have some difficulties to use the two core libraries as we implement the two standard window system libraries exposing C functions. To reduce these difficulties, we create adapters to make reusing the two libraries easier.

Note the adapters don’t belong to the core communication libraries at all.

## Server adapter and client unit test code snippet

Here is a sample test code snippet for server side:

*int \_tmain(int argc, \_TCHAR\* argv[]) {*

*string errMsg = InitializeSecureCommServer(CServerHandlerBase::OnAccepted<CRequestHandler>, 20901);*

*if (!errMsg.size())*

*errMsg = ::RunSecureCommServer("server.pem", "server.pem", "test", "dh512.pem");*

*if (errMsg.size())*

*cout << "Error message = " << errMsg <<endl;*

*cout << "Press any key to shutdown the application ......" << endl;*

*getchar();*

*ShutdownSecureCommServer();*

*return 0;*

*}*

Referring to the section 7.4, you can quickly understand these calls. The input *CServerHandlerBase::OnAccepted<CRequestHandler>* actually is a callback for *POnAccepted*. It will become very clear to you after reading through the template function and its inside comments.

## Client adapter and server unit test code snippet

Here is a sample client code snippet.

*int \_tmain(int argc, \_TCHAR\* argv[]) {*

*int errCode = ::SetCertificateVerifyFile("ca.pem");*

*//Call DoClientConnection within constructor*

*//Call DestroyClient within destructor*

*CRequestHandler handler("localhost", 20901);*

*SC::ICertificate \*cert = handler.GetCertificate();*

*if (!cert) {*

*std::cout << "No connection! error code = " << handler.GetErrCode()*

*<< ", error message = " << handler.GetErrMsg() << std::endl;*

*return 1;*

*}*

*//do ceritificate verification before sending any sensitive data*

*const char \*str = cert->Verify(&errCode);*

*std::cout << "Cert verification result = " << str*

*<< " with error code = " << errCode << std::endl;*

*std::cout << "SSL session info: " << cert->GetSessionInfo() << std::endl;*

*std::cout << "Cert perm :\r\n" << cert->GetCertPem() << std::endl;*

*//do authentication first, which is required from server side for better security*

*if (!handler.DoAuthentication("MDTUser", "SomePassword", "charliedev-1")) {*

*std::cout << "Authentication failed" << std::endl;*

*return 1;*

*}*

*//set a callback for returning result*

*SC::ClientSide::ResultHandler rh = [](SC::ClientSide::CAsyncResult &ar) {*

*std::string res;*

*ar >> res;*

*std::cout << res << std::endl;*

*};*

*unsigned int res = handler.SendRequest(idSayHelloWorld, "Philip", "Dalrymple", rh);*

*res = handler.SendRequest(idSleep, (unsigned int)5000, [](SC::ClientSide::CAsyncResult &ar){});*

*res = handler.SendRequest(idSayHelloWorld, "Bob", "Gargan", rh);*

*//bool ok = handler.WaitAll();//convert asynchronous computations into synchrnous ones*

*std::cout << "Press any key to shutdown the application ......" << std::endl;*

*::getchar();*

*return 0;*

*}*

The above code snippet demonstrates typical usage cases of client core library with help of client adapter with sending a few requests. Note you can convert all asynchronous requests into synchronous ones by calling the method *WaitAll* at your will anytime after authentication.

At last, asynchronous computation style works greatly for window .NET form application as .NET version 4.5 starts supporting key words *async* and *await* for better UI response. In regards to window C++/MFC development environment, you can use window system function PostMessage to update UI elements after a returning result arrives.