

## Tutorial 1 – Hello world for a simple client/server application

### Contents:

#### *Preface*

#### *Objectives*

#### *Create client and server skeleton codes with the tool uidparser.exe*

#### *Server side development*

- *Reference SocketProAdapter for easy development*
- *Derive HelloWorldPeer from CClientPeer*
- *Register services and deal with slow requests*
- *Permission to client connections*
- *Start one listening socket per server instance*
- *Only OnSlowRequestArrive called within worker threads*

#### *Responsibilities of main thread within SocketPro server*

#### *Client side development*

- *Async handler*
- *Serialization of complex structures and interface IUSeilizer for .NET, Java and Python*
- *Synchronous and asynchronous requests as well as requests batching for the best network efficiency*
- *Client side persistent message queue*
- *Fat client*

#### *Async/Await for asynchronous tasks*

#### *Cross-platform and cross-development language tests*

### 1. Preface

SocketPro, a communication framework, is written with continuous inline request/result batching, asynchrony, and parallel computation in mind by use of raw non-blocking TCP/IP socket. The framework is considerably different from common communication frameworks like Java RMI, WCF and web service. SocketPro offers many specific features which cannot be found in other frameworks. Therefore, we have created a series of tutorial samples to assist your SocketPro development. As you study these samples, it is strongly recommended that you fully understand every one of the features and code comments while experimenting with them. All of the tutorial samples are written with C++, C#, VB.Net, Java, Python and Node.js languages. Each of the samples contains

one console server application and at least one client application for each of these development languages. We'll add supports to other development language in the future.

Many development languages have been enhanced to better support asynchronous computation by use of Lambda expressions, anonymous delegate/function and *async/await*. Therefore, as you will see, new SocketPro client side adapters are rewritten and improved to fully take advantages of these new features for supported languages.

To get used to SocketPro fast, you must keep in mind that SocketPro uses asynchronous and request batching computation by default. Once you are used to SocketPro computation models, you will like the power of SocketPro!

To maximize tutorial code reusability, all of tutorial projects may refer to files inside other project directory, dependent on each of tutorial project types like C++, C#, Java, VB.NET, Python, Node.js and window CE .NET. At this writing time, there are seven solution files for the six development environments, and each of development language solutions contains all of available tutorial projects. These solutions are:

Dev. language	Solution file path	IDE tools
C++	../socketpro/tutorials/cplusplus/cplusplus.sln	Visual studio 2010 only. No solution file for Linux
C#	../socketpro/tutorials/csharp/csharp.sln	Visual studio 2010 and MonoDevelop 3.0 later
VB.Net	../socketpro/tutorials/vbnet/vbnet.sln	Visual studio 2010 only
Java	../socketpro/tutorials/java/nbproject	Netbean 8.0 or later for both windows and Linux
C# for wince	../socketpro/tutorials/ce/ce.sln	Visual studio 2008 only
Python	../socketpro/tutorials/python/.idea	PyCharm
Node.js	../socketpro/src/njadapter	Nodepad++ or any other editor tools

## 2. Objectives

Understand a UID (universal interface definition) file, and use it with SocketPro tool uidparser.exe to quickly create skeleton codes for both client and server.

This very first sample is designed to quickly create a service that processes the following three requests with traditional client/server communication model.

- Create a hello message at server side and return the message back to a client with two inputs first and last names.
- Send a sleep request from a client to a server so that a server will sleep for a given time. As you can see, the request may take a longer time to complete. Therefore, the request will be processed within a worker thread so that it will not block processing requests from other clients.
- Echo a complex structure between a client and a server. This request demonstrates how to exchange a complex structure across applications which may be created from different development languages or platforms within SocketPro framework.
- Convert asynchronous requests into synchronous ones with SocketPro client method *WaitAll*.
- Process multiple requests in batch to improve the underlying network efficiency for better application performance and scalability.
- Use Lambda expressions at client side for tracking various common events at client side as well as processing request return results.
- Use SocketPro persistent message queue at client side to improve communication stability over instable communication environments and software maintenance.
- Override server side virtual functions to track server communication events.

Finally, tutorial demo projects are located at the directory of  
../socketpro/tutorials/(csharp|vbnet|java|cplusplus|ce|python)/hello\_world.

### 3. Create client and server skeleton codes with uidparser.exe

SocketPro comes with a code generator to write skeleton client and server codes from a given universal interface definition file with file extension uid. It is very similar to other interface definition files like COM and CORBA. Take this sample, its UID file contains the below code.

```
49 [
50     ServiceID = 1 //service id can't be less than 1
51 ]
52 HelloWorld $ = Slow request
53 {
54     string SayHello(string firstName, string lastName);
55     $void Sleep(int ms);
56     CMyStruct Echo(CMyStruct ms);
57 }
```

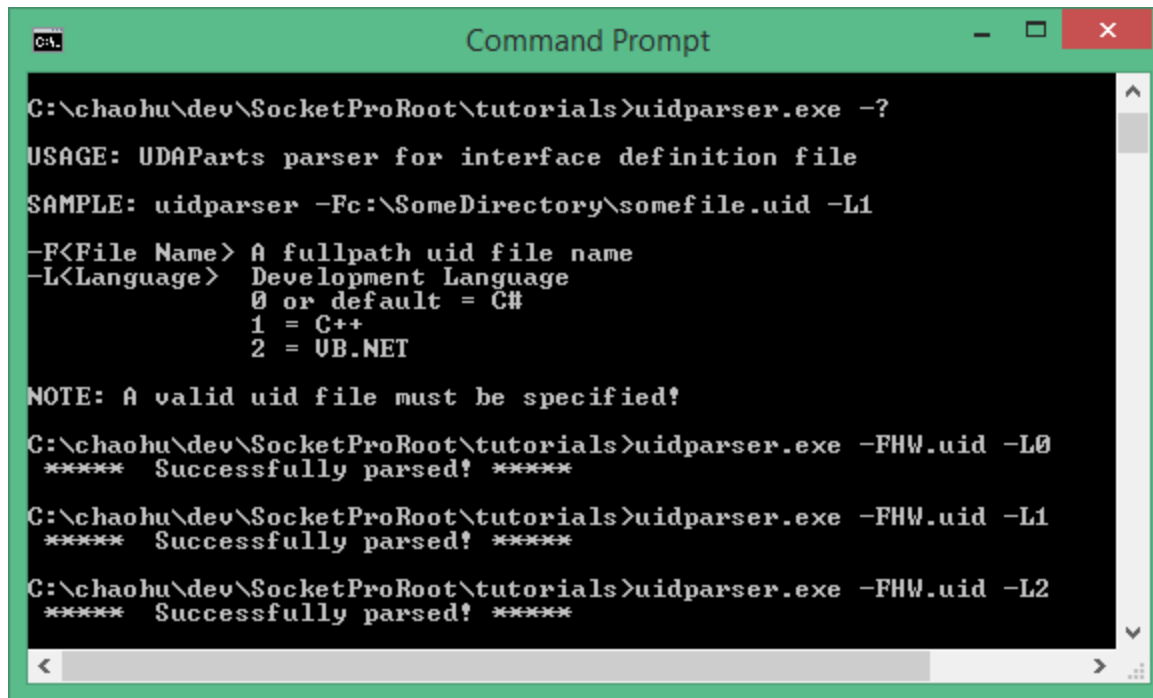
Figure 1: A sample universal interface definition.

In regards to the correct use of uidparser.exe, please see detailed comments inside file HW.uid. There is no need to re-describe the various simple rules in this tutorial again. However, you must be clear if a request takes a long time to process. Take this sample as an example, the request Sleep may require a long time to process, and all of other two requests will require very little time to be completed. Put the char '\$' before the return value to indicate a request to be slow as labeled in the picture above.

In addition to common data types, the tool also supports complex data structures. Take this sample as an example. The complex structure is *CMyStruct* as underlined in the above image. Note that this file is actually located at the directory ../SocketProRoot/tutorials/(csharp|cplusplus|vbnet|java|src|ce|python)/uqueue\_demo

Note that the data types are based on C#. The tool uidparser.exe will map them to data types of supported languages. If not proper, you could just manually change data types by mapping later.

You can use the tool uidparser.exe to quickly create skeleton codes for both client and side. See the below Figure 2 for creating skeleton codes for C#, C++ and VB.NET.



```
C:\chaohu\dev\SocketProRoot\tutorials>uidparser.exe -?
USAGE: UDAParts parser for interface definition file
SAMPLE: uidparser -Fc:\SomeDirectory\somefile.uid -L1
-F<File Name> A fullpath uid file name
-L<Language> Development Language
               0 or default = C#
               1 = C++
               2 = VB.NET
NOTE: A valid uid file must be specified!
C:\chaohu\dev\SocketProRoot\tutorials>uidparser.exe -FHW.uid -L0
***** Successfully parsed! *****
C:\chaohu\dev\SocketProRoot\tutorials>uidparser.exe -FHW.uid -L1
***** Successfully parsed! *****
C:\chaohu\dev\SocketProRoot\tutorials>uidparser.exe -FHW.uid -L2
***** Successfully parsed! *****
```

Figure 2: Create SocketPro client and server skeleton codes from a given UID file with the tool uidparser.

For this sample, the tool uidparser creates a set of files with proper file extensions for each of three development languages. Take C# as a sample. The first file named HW\_i.cs contains all of the constant definitions as shown in the following figure 3.

```
1 public static class HWConst
2 {
3     //defines for service HelloWorld
4     public const uint sidHelloWorld = (SocketProAdapter.BaseServiceID.sidReserved + 1);
5
6     public const ushort idSayHelloHelloWorld = ((ushort)SocketProAdapter.tagBaseRequestID.idReservedTwo + 1);
7     public const ushort idSleepHelloWorld = (idSayHelloHelloWorld + 1);
8     public const ushort idEchoHelloWorld = (idSleepHelloWorld + 1);
9 }
```

Figure 3: A sample C# file for constant definitions.

This file will be referenced by both client and server projects. The constant *sidHelloWorld* is a service id for this sample service. All of the others are identifier numbers for three requests. SocketPro supports multiple services within one listening socket. Each of the services must be identified by one unique identifier number named as service id. Similarly, each of the requests has one unique identification number per service, which is named as request id.

#### 4. Server side development

**Reference SocketPro adapter for easy development:** At the very beginning, you need to reference SproAdapter.dll if your developmental environment is one of the dotNet languages (C# and VB.NET), or add the files membuffer.cpp and aserverw.cpp into your C++ server project. In addition, you need to refer to the C++ server adapter header file aserverw.h if a sever project is based on C++.

Java adapter is located at the directory ../socketpro/bin/java

Python adapter is located at the directory ../socketpro/bin/spa

**Derive HelloWorldPeer from CClientPeer for .NET and C++:** Next, the tool automatically derives a class (*HelloWorldPeer*) from the class *CClientPeer* inside the namespace *SocketProAdapter.ServerSide* as shown in the file hwImpl.cs and the below figure 4. SocketPro automatically manages threads for you. On the server side, SocketPro will dispatch all fast requests on one main thread for processing and all slow requests onto worker threads for processing. Note that the number of main threads are fixed and set during starting listening socket, but SocketPro server may create multiple worker threads automatically as needed on the fly. It is noted that the number of main threads is default to 1. If a multi-core server requires high CPU for encryption/decryption and compression/decompression, you may need to increase main threads for better performance.

```
1
2  /* **** including all of defines, service id(s) and request id(s) **** */
3  using System;
4  using SocketProAdapter;
5  using SocketProAdapter.ServerSide;
6
7  //server implementation for service HelloWorld
8  public class HelloWorldPeer : CClientPeer
9  {
10     [RequestAttr(hwConst.idSayHelloHelloWorld)]
11     private string SayHello(string firstName, string lastName)
12     {
13         //processed within main thread
14         System.Diagnostics.Debug.Assert(CSocketProServer.IsMainThread);
15
16         return "Hello " + firstName + " " + lastName;
17     }
18
19     [RequestAttr(hwConst.idSleepHelloWorld, true)] //true -- slow request
20     private void Sleep(int ms)
21     {
22         //processed within
23         System.Diagnostics.Debug.Assert(!CSocketProServer.IsMainThread);
24
25         System.Threading.Thread.Sleep(ms);
26     }
27
28     [RequestAttr(hwConst.idEchoHelloWorld)]
29     private CMyStruct Echo(CMyStruct ms)
30     {
31         return ms;
32     }
33 }
34
```

Slow - worker thread; fast - main thread

Figure 4: Server implementation for C#

As you can see from the above figure 4, SocketPro .NET server adapter uses attribute to specify if a request is slow one.

For C++, SocketPro uses two pure virtual functions *OnFastRequestArrive* and *OnSlowRequestArrive* to help you implement all supported requests as shown in the below figure 5. As each name individually indicates, all fast requests are called from the first virtual function by one main thread, and all slow requests from the second virtual function by one of worker threads. Pay close attention to the labels on the figure for how to combine inputs and outputs at server side.

```

15
16 /* **** including all of defines, service id(s) and request id(s) **** */
17 #include "../HW_i.h"
18
19 //server implementation for service HelloWorld
20
21 class HelloWorldPeer : public CClientPeer {
22 private:
23     void SayHello(const std::wstring &firstName, const std::wstring &lastName, /*out*/std::wstring &SayHelloRtn) {
24         assert(CSocketProServer::IsMainThread());
25         SayHelloRtn = L"Hello " + firstName + L" " + lastName;
26     }
27
28     void Sleep(int ms) {
29         assert(!CSocketProServer::IsMainThread());
30 #ifdef WIN32_64
31         ::Sleep(ms);
32 #else
33         boost::this_thread::sleep(boost::posix_time::milliseconds(ms));
34 #endif
35     }
36
37     void Echo(const CMyStruct &ms, /*out*/CMyStruct &msOut) {
38         msOut = ms;
39     }
40
41 protected:
42     virtual void OnFastRequestArrive(unsigned short reqId, unsigned int len) {
43         BEGIN_SWITCH(reqId)
44             M_I2_R1(idSayHelloHelloWorld, SayHello, std::wstring, std::wstring, std::wstring)
45             M_I1_R1(idEchoHelloWorld, Echo, CMyStruct, CMyStruct)
46         END_SWITCH
47     }
48
49     virtual int OnSlowRequestArrive(unsigned short reqId, unsigned int len) {
50         BEGIN_SWITCH(reqId)
51             M_I1_R0(idSleepHelloWorld, Sleep, int)
52         END_SWITCH
53         return 0;
54     }
55 };

```

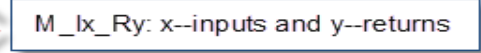


Figure 5: Server implementation for C++.



**Register services and deal with slow requests for .NET and C++:** SocketPro requires one to register all of the services supported at server side. For .NET development environments, you can simply specify an attribute for each of the services as shown in the below figure 6.

```
22 [ServiceAttr(hwConst.sidHelloWorld)]  
23 private CSocketProService<HelloWorldPeer> m_HelloWorld = new CSocketProService<HelloWorldPeer>();  
24 //One SocketPro server supports any number of services. You can list them here!  
25
```

Figure 6: Register services at server side for C#.

In regards to C++ development, you can register all services within the virtual function *CSocketProServer::OnSettingServer* as shown in the following figure 7. Inside the function, you can call the method *CBaseService::AddMe* with a service id and a COM thread apartment model (ignored on non-window platforms). For this sample, they are set to *sidHelloWorld* and *taNone*, respectively. The *taNone* indicates a worker thread that will not be initialized into any COM thread apartment because this sample doesn't create any COM object at all. Note that we register a slow request in C++ by calling the method *CBaseService::AddSlowRequest*, but we register a slow request in .NET by specifying its second parameter to true as shown in the figure 4.



```

61 private:
62     CSocketProService<HelloWorldPeer> m_HelloWorld;
63     //One SocketPro server supports any number of services. You can list them here!
64
65 private:
66     void AddService() {
67         //No COM -- taNone; STA COM -- taApartment; and Free COM -- taFree
68         bool ok = m_HelloWorld.AddMe(sidHelloWorld, taNone);
69         //If ok is false, very possibly you have two services with the same service id!
70
71         ok = m_HelloWorld.AddSlowRequest(idSleepHelloWorld);
72         //Add all of other services into SocketPro server here!
73     }
74
75 protected:
76     virtual bool OnSettingServer(unsigned int listeningPort, unsigned int maxBacklog, bool v6) {
77         //amIntegrated and amMixed not supported yet
78         CSocketProServer::Config::SetAuthenticationMethod(amOwn);
79
80         //add service(s) into SocketPro server
81         AddService();
82         return true; //true -- ok; false -- no listening server
83     }

```

Diagram annotations:

- Register a service**: Points to line 68 (`bool ok = m_HelloWorld.AddMe(sidHelloWorld, taNone);`).
- Register a slow request**: Points to line 71 (`ok = m_HelloWorld.AddSlowRequest(idSleepHelloWorld);`).

Figure 7: Register services at server side for C++.

Once you specify a request is slow one, SocketPro server will use the information to route a request onto different threads and also create a thread automatically on the fly if necessary.

SocketPro simplifies your code development at server side significantly. It manages all threads for you automatically. Usually, you will not create your own threads even though you can be assured to create your own threads. When a thread is idle for a long time, perhaps for one minute by default, SocketPro server will also silently kill it for you.

**Permission to client connections:** We need to control socket connections from different clients for the sake of security. You can deny or give permission to a client by its credentials (user id and password). To achieve such a goal, you can implement it by overriding the below virtual function *OnIsPermitted* as shown in the figure 8. If the function returns true, you give permission to a client. Otherwise, you deny the socket connection, and SocketPro will automatically close it for you.

```

7   protected override bool OnIsPermitted(ulong hSocket, string userId, string password, uint nSvsID)
8   {
9       Console.WriteLine("Ask for a service " + nSvsID + " from user " + userId + " with password = " + password);
10      return true;
11  }
12

```

Figure 8: Authenticate a client by its user id, password and service id.

**Start one listening socket per application instance:** At this point, we have almost completed the server application. At last, we need a piece of code to start one listening socket per application instance as shown in the figure 9.

```

27  static void Main(string[] args)
28  {
29      CMySocketProServer MySocketProServer = new CMySocketProServer();
30
31      //test certificate and private key file, server.pem, is located at ../SocketProRoot/bin
32      //MySocketProServer.UseSSL("server.pem", "server.pem", "test");
33
34      if (!MySocketProServer.Run(20901))
35          Console.WriteLine("Error code = " + CSocketProServer.LastSocketError.ToString());
36      Console.WriteLine("Input a line to close the application .....");
37      string str = Console.ReadLine();
38      MySocketProServer.StopSocketProServer(); //or MySocketProServer.Dispose();
39  }

```

Figure 9: Start one listening socket per application instance by calling the method Run on a given port (for example, 20901).

By this time, you can compile and run the sample SocketPro server which is able to support tens of thousands of clients very easily.

**Only *OnSlowRequestArrive* called within a worker thread:** As you may have already known, SocketPro offers a simple thread model that all the virtual functions *OnXXX*, except the virtual function *OnSlowRequestArrive*, are called with one main thread.

## 5. Responsibilities of main thread within SocketPro server

The main thread at SocketPro server is the thread running with the listening socket. The main thread has a lot of responsibilities. Major ones are listed in the following:

- a. Monitoring various network events and communication events between worker threads and the main thread.
- b. Manage various pools such as thread pools, global and private memory pools, and pools of *CClientPeer*-derived objects.
- c. Route various slow requests onto different worker threads.
- d. Process all of fast requests from all of clients.
- e. Authenticate a client.
- f. Decompress incoming data if it is originally compressed (zipped).
- g. Decrypt incoming data if it is originally encrypted.
- h. Create or kill worker threads on the fly if necessary.
- i. Encrypt or compress returned data of all of fast requests.
- j. Manage plug-in libraries written from C/C++.
- k. Find dead clients for unknown reasons.

You are not required to understand how the main thread manages these in detail. However, you should keep in mind that no slow requests should be processed in the main thread in general. Worker threads are used to process all slow requests, and to encrypt or compress their returned results.

## 6. Client side development

SocketPro has one client system library, *usocket.dll* on window platforms or *usocket.so* on Linux platforms, which must be used for all of client applications development. In addition, SocketPro uses *openssl* (Linux) or *SSPI* (windows) libraries for SSL/TLS secure communication. In order to reduce client coding complexities, SocketPro provides wrappers. To use these helpful wrappers, simply use the name spaces *SocketProAdapter* and *SocketProAdapter.ClientSide*.

**Async handler** – is designed for sending requests and processing returning results using asynchrony style. However, you can easily convert all asynchronous requests and results into synchronous ones. SocketPro client adapter also provides templates in C++ or generics in .NET to convert them as shown in the below Figure 10 if you like to use synchronous computation style with help of the functions *ProcessRx*, where x indicates the number of returning data which could range from 0 to 5.

```
2 using System;
3 using SocketProAdapter;
4 using SocketProAdapter.ClientSide;
5
6 public class HelloWorld : CAsyncServiceHandler
7 {
8     public HelloWorld()
9         : base(hwConst.sidHelloWorld)
10     {
11     }
12
13     public string SayHello(string firstName, string lastName)
14     {
15         string SayHelloRtn;
16         bool bProcessRy = ProcessR1(hwConst.idSayHelloHelloWorld, firstName, lastName, out SayHelloRtn);
17         return SayHelloRtn;
18     }
19
20     public void Sleep(int ms)
21     {
22         bool bProcessRy = ProcessR0(hwConst.idSleepHelloWorld, ms);
23     }
24
25     public CMyStruct Echo(CMyStruct ms)
26     {
27         CMyStruct EchoRtn;
28         bool bProcessRy = ProcessR1(hwConst.idEchoHelloWorld, ms, out EchoRtn);
29         return EchoRtn;
30     }
31 }
```

Figure 10: Hello world asynchronous service handler for C#

**Serialization of complex structures and interface IUSerilizer for .NET** – SocketPro adapter can automatically serialize and de-serialize simple types of data easily and effortlessly. However, it may require a little effort for you to implement the interface *IUSerialize* for structures if you would like your application data movement to be compatible across multiple platforms and development languages C++, C#, and others. For details, see this short article [memoryqueue.pdf](#). Note SocketPro also supports .NET

serialization by reflection. We recommend *IUSerilize* for the sake of performance and compatibility across different operation systems and developmental languages.

**Synchronous and asynchronous requests as well as requests batching for the best network efficiency** – Let's see the Hello world client code as shown in the below Figure 11.

```

7  static void Main(string[] args)
8  {
9      CConnectionContext cc = new CConnectionContext("127.0.0.1", 20901, "hwClientUserId", "password4hwClient");
10     using (CSocketPool<HelloWorld> spHw = new CSocketPool<HelloWorld>(true)) { //true -- automatic reconnecting
11         bool ok = spHw.StartSocketPool(cc, 1, 1);
12         HelloWorld hw = spHw.Seek(); //or HelloWorld hw = spHw.Lock();
13         //optionally start a persistent queue at client side
14         ok = hw.AttachedClientSocket.ClientQueue.StartQueue("helloworld", 24 * 3600, false); //time-to-live 1 day
15
16         //process requests one by one synchronously
17         Console.WriteLine(hw.SayHello("Jone", "Dole"));
18         hw.Sleep(5000);
19         CMyStruct msOriginal = CMyStruct.MakeOne();
20         CMyStruct ms = hw.Echo(msOriginal);
21
22         //process multiple requests in batch asynchronously
23         ok = hw.StartBatching();
24         ok = hw.SendRequest(hwConst.idSayHelloHelloWorld, "Jack", "Smith", (ar) => {
25             string ret; ar.Load(out ret); Console.WriteLine(ret);
26         });
27         CAsyncServiceHandler.DAsyncResultHandler arh = null;
28         ok = hw.SendRequest(hwConst.idSleepHelloWorld, (int)5000, arh);
29         ok = hw.SendRequest(hwConst.idEchoHelloWorld, msOriginal, (ar) => {
30             ar.Load(out ms);
31         });
32         ok = hw.CommitBatching(true); //true -- ask server return multiple results in one shot
33         ok = hw.WaitAll();
34         Console.WriteLine("Press a key to shutdown the demo application .....");
35         Console.Read();
36     }
37 }

```

*Figure 11: Client main codes for sync and async requests as well as requests batching*

At first, SocketPro client requires a connection context for establishing socket connection from client to server as shown on the line 9. Next, SocketPro client starts a pool of sockets on the lines 10 and 11 with a number of threads for hosting non-blocking sockets. Here, we just create a single socket hosted within one thread only.

From line 17 through line 20, we send three requests synchronously, which requires three round trips to finish processing all three requests. Typically, developers just write such synchronous requests. However, we are able to send all three requests in one shot with only one round-trip after manually batching them. Each of the requests is set with one call back for processing returning results as shown in the lines 25, 28 and 30. This is requests batching with asynchrony computation style. The key feature is especially great for middle tier performance and scalability in case your application requires supporting high volume of small requests remotely. Note the calls at lines 23, 32 and 33 are optional. Without calls at lines 23 and 32, SocketPro is still able to do requests batching internally, but manual batch seems slightly better. The call `WaitAll` at line 32 is used to wait until all three requests are processed.

**Client side persistent message queue** – The call at line 14 starts a persistent message queue if you like to make sure all of the client requests are delivered to remote server, no matter what happens to the remote server, network or client machine. In case the network or server is down for any reasons such as network unplug, machine power-off, software update, server shutdown, and so on, the client requests are automatically backed up by the client side. The backup requests can be re-sent to server for processing after a remote server is restored to work.

After you run the Hello world server first, you can start running the simple client sample and step through codes now.

**Fat client** – Since SocketPro uses inline continuous data batching algorithm at both client and server side for best network efficiency, you can rely on it to create a fat but highly reusable client component as shown at SocketPro asynchronous mysql/mariadb and SQLite database plug-ins at the site [https://github.com/udaparts/socketpro/tree/master/samples/module\\_sample](https://github.com/udaparts/socketpro/tree/master/samples/module_sample).

## 7. Async/Await

If you do .NET development, you may like to use the new feature *async/await* for asynchronous tasks. SocketPro well-supports the new key statements *async* and *await* for all requests without the involvement of .NET thread pool or your worker thread. There is also visual studio 2012 .net sample project located at the directory `../socketpro/tutorials/(csharp|vbnet)/hello_world/win_async` for both C# and VB.NET. Here is the C# code snippet for demonstration.

```

18 private async void btnTest_Click(object sender, EventArgs e) {
19     if (m_spHw.ConnectedSockets == 0) {
20         txtRes.Text = "No connection";
21         return;
22     }
23     btnTest.Enabled = false;
24     try {
25         //execute one request asynchronously
26         txtRes.Text = await GetTask();
27
28         //execute multiple requests asynchronously in batch
29         txtRes.Text = await GetTasksInBatch();
30         btnTest.Enabled = true;
31     }
32     catch (Exception err) {
33         txtRes.Text = err.Message;
34     }
35 }
36 Task<string> GetTask() {
37     HelloWorld hw = m_spHw.AsyncHandlers[0];
38     return hw.Async<string, string, string>(hwConst.idSayHelloHelloWorld, "Jack", "Smith");
39 }
40 Task<string> GetTasksInBatch() {
41     HelloWorld hw = m_spHw.AsyncHandlers[0];
42     bool ok = hw.SendRequest(hwConst.idSleepHelloWorld, 5000, (ar) => { });
43     Task<string> task = hw.Async<string, string, string>(hwConst.idSayHelloHelloWorld, "Jone", "Don");
44     return task;
45 }

```

Figure 12: Use async and await for asynchronous tasks within SocketPro client adapter

In addition to VB.net and C#, Node.js well supports async and await now. All client server requests can be used with the two key words as shown at files ../socketpro/src/njadapter/hello\_world.js and server\_queue.js if necessary or required.

Beginning from MS Visual C++ 2015, you can use async and await with C++ development. However, the features are not available with other C/C++ compilers yet. You can see application of C++ async and await within the Visual C++ sample project at directory ../socketpro/tutorials/cplusplus/hello\_world/win\_async.



## 8. Cross-platform and cross-development language tests

SocketPro is created to fully support cross-platform and cross-language developments. For example, you can create a C++ Linux SocketPro server application which is directly accessible from a window client application written from C#. You can test all tutorial samples from different platforms and applications written from different languages. As you can see, SocketPro is written to support loose coupling communication architecture. Communications among clients and their server are decoupled, which is beneficial to software maintenance and evolution.