UDAParts

Development guide for SocketPro asynchronous communication framework

support@udaparts.com
12/30/2016

Table of Contents

1. Version history

The table below records major changes to this document

Date	Comment
2016-12-14	Initial

2. Audiences

2.1 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
- Request load balancing and routing
- HTTP and web socket protocols
- Socket, latency, throughput and network bandwidth
- Synchronous (blocking) and asynchronous (non-blocking) communication
- TCP/IP protocol for plain text communication
- SSL/TLSv1 encryption, decryption and server authentication
- Message replication
- C API functions and interfaces
- Adapters

2.2 Audiences

- Software architect
- Software developer
- Software development manager

3. Introduction and goals

Your applications have to deal with data movements among networked computers under most cases. High development simplicity, rich feature, high availability, secure, great performance and unlimited scalability are major qualities to be considered and wished. Originally inspired by Microsoft old communication technology DCOM, long time ago UDAParts began exploring raw TCP/IP socket APIs and tried to use both blocking and non-blocking sockets for better performance as shown at the site http://www.codeproject.com/Articles/1785/Offset-weaknesses-of-DCOM-with-strong-points-of-so. After many trials and failures for a long time, UDParts has finally created a powerful communication framework on non-block TCP/IP socket, which is named as SocketPro.

SocketPro is created with the following goals in mind:

- Taking advantages of full-duplex network concurrency transferring by use of non-blocking TCP/IP socket
- Industrial standard SSL3/TLSv1.x for secure communication
- Inline request/result data batching for better network efficiency
- One complete and integrated solution supporting multiple communication patterns such as client/server, publish/subscribe, asynchronous durable message queue, and load balancing/routing
- Cross operation systems supported
- Great compatibility among different development languages and platforms
- Super performance and scalability
- New async development features such as Lambda expression, anonymous callback, async/await and future fully supported for various development languages
- Parallel data transferring and computation
- Guaranteed thread safety for core components and adapters
- High reusability
- Simple development
- Low coupling and high maintainability

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 for the best network efficiency and concurrency by continuous inline data batching at both client and server sides simultaneously. To help you understand SocketPro data transferring pattern more clearly, please pay close attention to the below Figure 1 with your careful analyses which certainly help you understand the power of SocketPro.

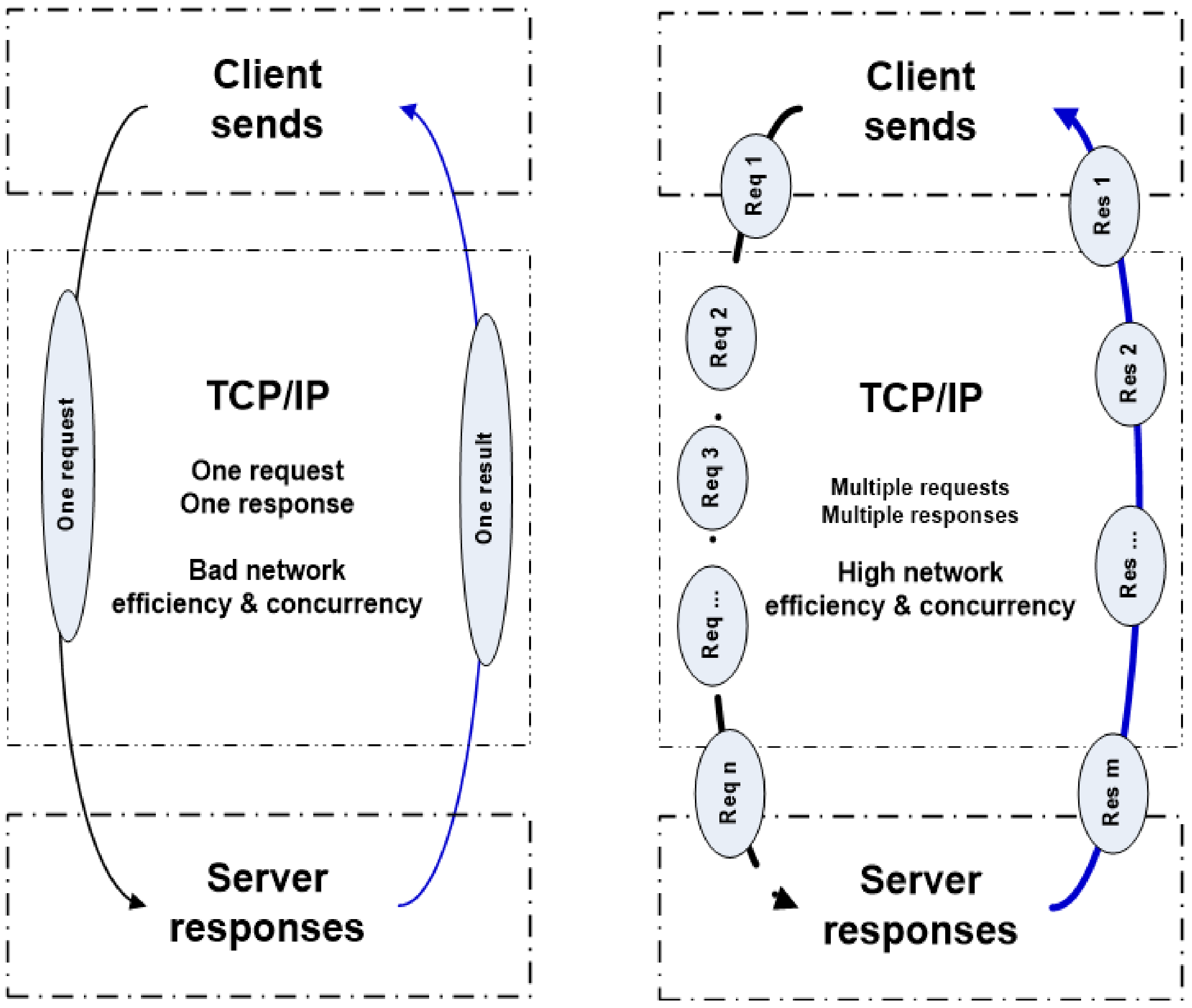


Figure 1: Two data transferring patterns on one socket connection

4.1 One-request for one-response or result

The above picture shows two types of data transferring patterns in pair for better comparison. First of all, let's have a close look at the left one with one socket connection. It really represents a typical synchronous communication pattern by use of blocking TCP/IP socket, which is 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 its corresponding result from a server to fake an asynchronous communication. However, it is not considered by UDAParts as a true asynchronous one 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 small portion of distributed application systems does 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 one 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 three fundamental issues, high latency, 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 milliseconds. 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 with how small requests. Further, most of requests and their corresponding results are smaller than 1460 in bytes, which lead 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 with a huge amount of software engineering effort and cost. The second factor influencing efficiency or throughput is request data size. Obviously, the efficiency or throughput would be very bad if your application has to support high volume of small requests having a few bytes. Therefore, very few modern communication frameworks such as Apache Kafka, Apache Spark and 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. It is not practical to improve network efficiency by use of better hardware having lower latency between two end points. If your application has to support WAN (wide area network) or wireless LAN, the situation could become much worse as latency could be easily between 2 and 250 milliseconds or more.

If we consider a data transferring path, it is not difficult for us to find another flaw that devices on the whole path cannot run concurrently at any time on one TCP/IP socket. This is actually a concurrency issue as it also degrades the total performance of a distributed application system significantly.

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 above figure 1.

4.2 Multiple-requests for multiple-responses or results

SocketPro framework is created solely on non-blocking TCP/IP socket communication to reduce the bad influence of high network latency on network efficiency or throughput by an in-line data batching algorithm on the fly so that all devices on a socket communication path run concurrently as long as multiple requests are available. Both client and server ends are able to push bigger data packed across multiple requests/results with inline data batching algorithm onto the other side concurrently and simultaneously.

Assuming we send n requests one-by-one with different sizes of bytes from a client to a server (see the right side of the above figure 1), the server will process one request after another sequentially once a request arrive. Since sending requests in memory is usually faster than network data transferring, an in-line data batching algorithm works silently to pack different sizes of request data into one bigger chunk before being put on network wire. This improves network efficiency or throughput due to bigger chunk data. Besides, it also reduces degradation of high networking latency on throughput. SocketPro server side does real-time stream processing since we can think SocketPro takes requests from a client as a request stream to requests just like a binary stream to bytes.

Similarly, a SocketPro server returns different sizes of results to a client by pushing style with inline data batching algorithm. Certainly, a SocketPro server is also able to emit best throughput from server to client. Under many cases, m results which are larger than the number of requests n, will be pushed onto a client on one single socket concurrently while a client is sending requests. By this time, the number m of responses or results is either equal to or larger than the number n of requests (see the right side of the above figure 1) with SocketPro. Under some situations, the value m could be far larger than the value n. Assuming we need to download a big file from server to client, the server could push one result of a file attribute containing error code, file availability and size, a number (j) of results of file chunks, and one result of ending notification at the end. The number of request may be just one, but the number of results or responses would be 1 + j + 1 totally. Apparently, this case would also happen with pushing any types of large collections such as data table and structure.

At this writing time, SocketPro data transferring pattern is unique. As far as we know, there is no other distributed application employing this or similar pattern. UDAParts has spent a considerable amount of effort to study other popular frameworks, libraries or distributed applications. UDAParts has not found any one has better throughput than SocketPro. Under most cases, SocketPro provides much better throughput under a set of the same hardware and operation system. Now, let’s talk about latency. As mentioned at the above section, Apache Kafka batches multiple requests at client side and sends them in one shot at a predefine time interval before putting a bigger chunk into a TCP/IP socket. Kafka does so for better throughput, but it sacrifices the latency as a predefine time interval, which is usually a few milliseconds up to seconds so that a user has to balance between throughput and latency, which is not fun at all. Contrarily, SocketPro’s latency would be in the range of 0.x up to a few milliseconds which is dependent on hardware and request size in bytes. Note that SocketPro also provides persistent message queue functionality as Kafka does, but a SocketPro user doesn’t have to balance between throughput and latency at all. UDAParts internal studies have shown that SocketPro persistent message queue is much better than Kafka in both throughput and latency.

Due to the SocketPro unique communication pattern having inline data batching algorithm, you can easily create highly reusable thick/fat client applications on SocketPro without much degradation of application performance as a thick client has to provide more flexibility with much chatter and richer functions.

Summarily, SocketPro is a world-leading package of secured communication software components written with continuous inline request/result batching, real-time stream processing and asynchronous data transferring in mind. UDAParts performance studies show that SocketPro is very easily able to saturate one 1-GB or less network bandwidth for typical power computers. SocketPro really runs nicely on 10-GB network bandwidth with powerful computer systems.

4.3 Multiple-requests for less-responses or results

This pattern could be found on user own defined protocols with some applications using non-blocking TCP/IP socket. It is not difficult to create such pattern communication components, which have more number of requests than the number of returned results. This type of systems typically focuses on better request throughput but ignores on result throughput. Contrarily, SocketPro takes care of both request and result throughputs equally. Usually, this type of systems provides much simpler and less functionalities than SocketPro. Therefore, we stop here and don't spend more effort to describe or compare it with SocketPro any more as they are not really reusable to us.

5. SocketPro communication architecture

Similar to all other communication frameworks, SocketPro is designed with its own design goals as shown in the below Figure 2. SocketPro framework has one client core library (usocket) and one server core library (uservercore). Both of them, which export a number of operation system C functions, are written by use of C/C++ for the best performance. You can find these C functions at the files uclient.h and userver.h, respectively. Currently, both core libraries are available for window ce, window and various Linux variant platforms.

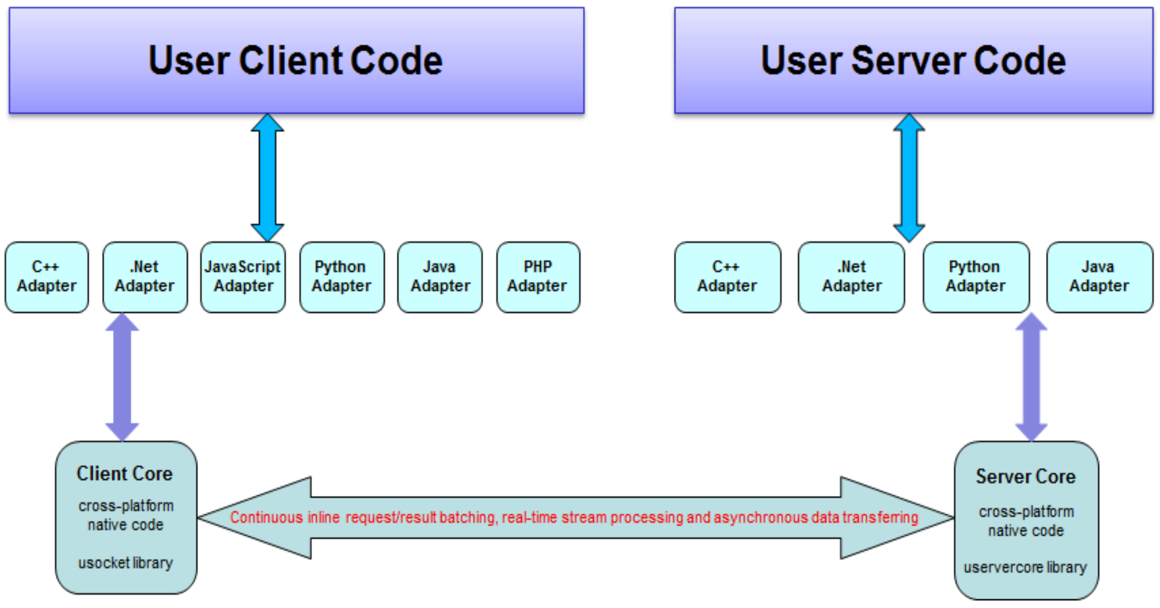


Figure 2: SocketPro communication architecture on one socket connection

Since these system C functions are not so friendly to be used by you, UDAParts has already created an adapter for each of development languages to make you development easier. Therefore, typically your client and server codes will directly communicate only with one of adapters in middle at both client and server side. Note that your client and server could use different adapters, which are all compatible across both development languages and operation systems.

5.1 Client core library (usocket)

As described at the above section, all of basic features of one socket connection are implemented within SocketPro client core library (usocket.dll for windows and libusocket.so for Linux platforms) as shown in the below Figure 3. One single socket connection supports online message bus for publish-subscribe communication pattern and two sets (base and user-defined) of requests within client side.

Additionally, SocketPro client library has implemented inline data compression (zipping) and decompression (unzipping) and persistent message queue (request log or backup). The first feature can be a convenient tool for you to improve data transferring performance on WAN having low bandwidth, but it is not recommended for you to use it on LAN as it requires too much CPU especially for data compression on LAN. The second feature is implemented to improve client side fault tolerance focused on network instability and client/server application shutdown for all types of reasons such as uncaught exceptions, software upgrade, server power-off, and so on. For example, in case a network switch is turned off, SocketPro client is able to resend requests saved in a persistent message queue automatically when the switch is turned on. The two features are optional.

Finally, SocketPro framework uses industrial standard SSL/TLS to secure communication by encryption and decryption between client and server sides. SocketPro employs SSPI (security support provider interface) channel on window platforms, and openssl (https://www.openssl.org) on Linux platforms. Today, this is a standard feature for anyone of communication frameworks.

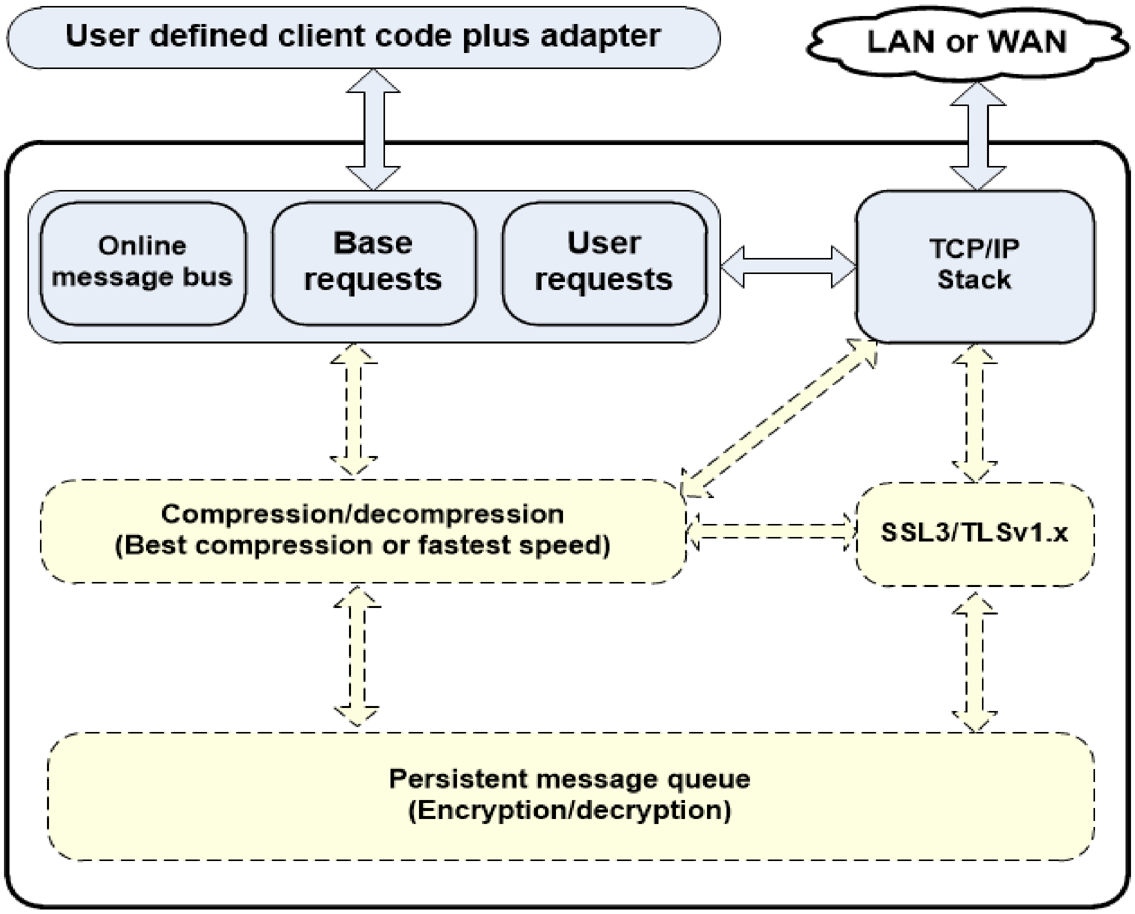

User defined client code plus adapter
LAN or WAN
Online message bus
Base requests
User requests
TCP/IP Stack
Compression/decompression (Best compression or fastest speed)
SSL3/TLSv1.x
Persistent message queue (Encryption/decryption)

Figure 3: SocketPro client core built-in features on one socket connection

5.1.1 Base and user-defined request identification numbers

Each of requests is labeled by a unique identification number (id). When a client sends a request to a remote server, the server side is able to properly parse the request according to an obtained request id as each of requests has its own input signature of parameters. SocketPro has already defined a set of request ids (identification numbers) for a set of base and built-in requests, which are less than idReservedTwo (0x2001) defined at the file ../socketpro/include/ucomm.h.

All of user defined request ids are larger than or equal to idReservedTwo.

5.1.2 Online message bus

SocketPro client core has a built-in feature for you to quickly and conveniently use publish-subscribe pattern for exchanging various messages among online (or connected) clients. People may call the pattern as internet chatting, message notification, and online message bus as well as others. It is noted that UDAParts may use these terms interchangeably. A client can use the feature to send (publish) any messages onto one or different chat groups of connected clients through SocketPro server in middle. In addition, a client is able to notify another client identified by a targeted client login user id.

5.1.3 Compression and decompression

Compression is very CPU extensive. Therefore, it is not recommended for LAN in general. However, you may use the SocketPro convenient feature to reduce data transferring size for better performance if your system has to support WAN or wireless LAN. SocketPro has two compression levels, best speed and best compression. The first one is focused on compression speed with less CPU cost and compression rate, but the second one is focused on better compression rate with much lower compression speed and higher CPU cost.

5.1.4 Persistent message queue at client side

As described previously, this SocketPro feature is to improve client side fault tolerance. A reliable distributed application system must be designed to tolerate various faults as many as possible. However, it is not easy to solve these issues under many situations. Besides, solving these issues may make distributed application code very messy. Common faults are net wire unplug, switch power-off, software component upgrade, uncaught exception and computer power-off. In case anyone of faults happens, SocketPro is automatically able to resend the request from persistent message queue saved previously once the issue is removed. As you can see, SocketPro client persistent message queue can thought as request log for request backup.

5.1.5 SocketPro pool for parallel computation

So far we have just elaborated client data transferring fundamentals, which is focused on one non-blocking socket only. In fact, a client is always required to obtain a socket from a socket pool before sending any request. A socket pool is made of one or more worker threads, and each of them hosts one or more non-blocking sockets as shown in the following Figure 4.

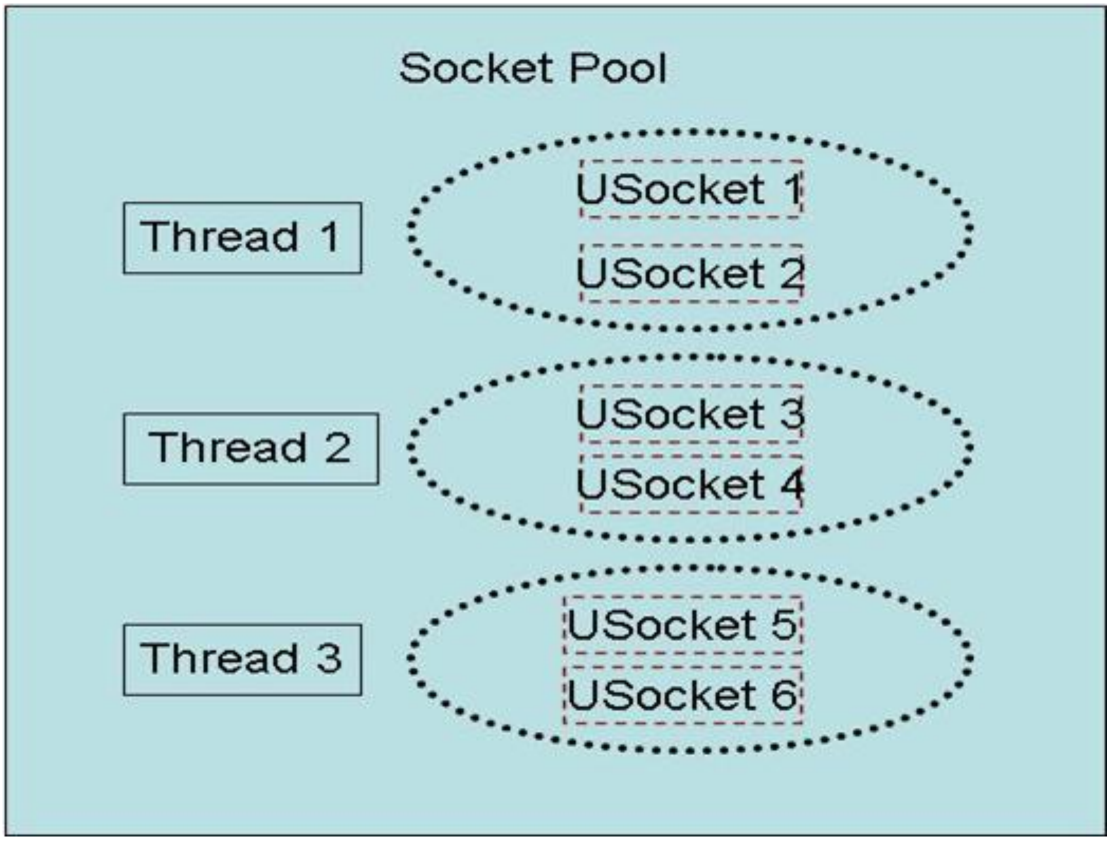


Figure 4: A socket pool having three worker threads for parallel computation

SocketPool is designed for transferring data from a client to one or more SocketPro servers for parallel processing. In case a client may have to process returning results with expensive CPU costs, the client should start a pool with multiple threads. For example, a high performance web server application has to deliver a considerable large record set of data onto browsers from backend database table, which may require high CPU costs concurrently on multiple cores at web server application side to process encrypted binary results as fast as possible from a SocketPro server. Under many cases, one worker thread may be enough if processing response results does not require much CPU cost.

5.1.6 WaitAll for converting asynchronous requests into synchronous ones

All of sockets are always running with non-blocking communication style so that all of requests are always executed asynchronously. However, most of software developers are not used to write asynchronous codes although some development languages and libraries are gradually starting to support it. Additionally, asynchronous codes are still more complex to read usually, even though we have closure, lambda expression and anonymous delegate for help. To reduce the complexity of asynchronous codes, SocketPro client core library provides one particular function WaitAll to convert asynchronous requests into synchronous one from your code so that your code complexity may be reduced for better readability.

5.2 Server core library (uservercore)

It is time to focus discussing key features supported at server side, which are implemented within server core library uservercore. Similar to client side, server core library also uses in-line continuous data batching algorithm on non-blocking sockets to silently pack real-time stream processing results for best network efficiency. Server core built-in features are shown in the below Figure 5.

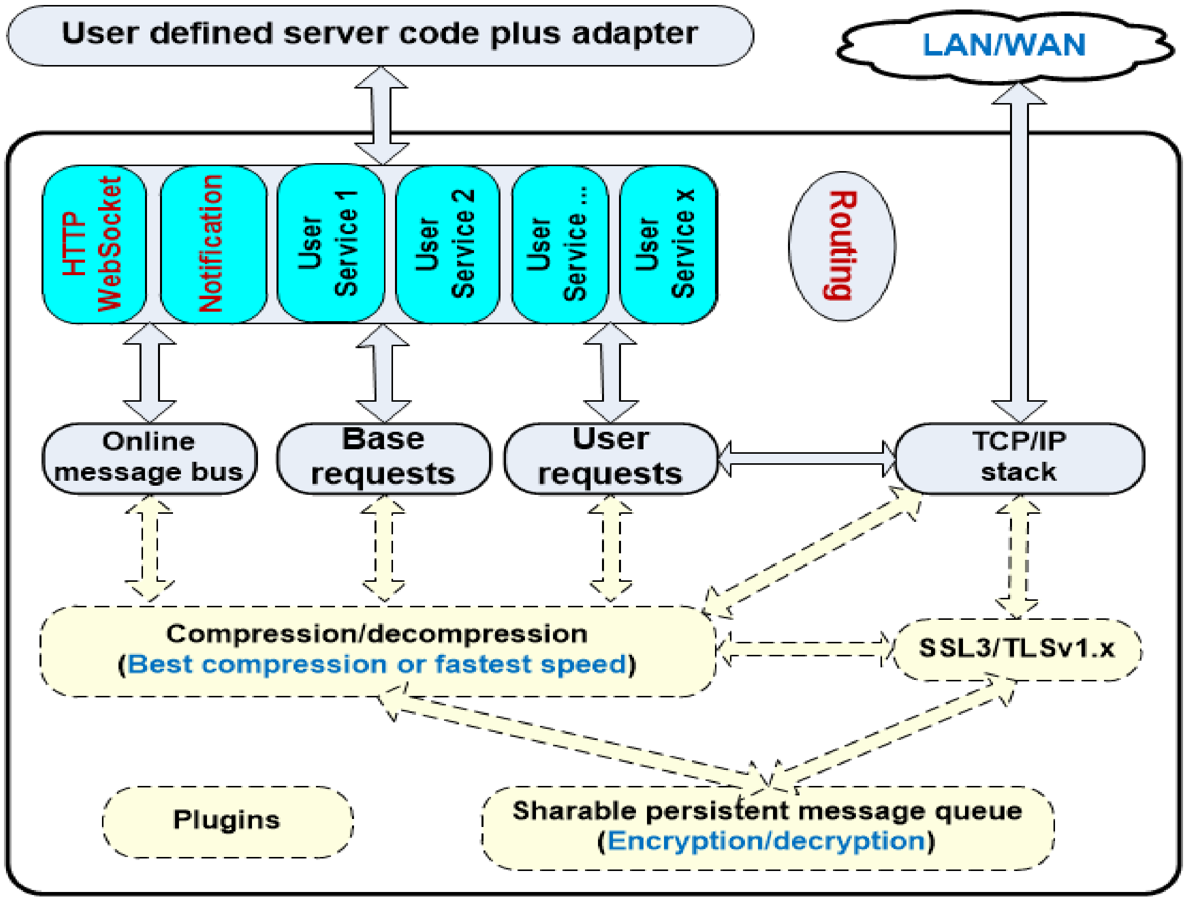


Figure 5: SocketPro server core built-in features

SocketPro server core uses one listening socket or one port to support unlimited number of services such as HTTP/websocket, file, database access, server side persistent message queue and other user defined services. Each of them represents a group of requests. There are two built-in services, HTTP/websocket and notification (or online message bus). In addition, SocketPro server core has a built-in feature Routing implemented for load balancing. Like client core, server core supports industrial standard SSL3/TLSv1.x for secure communication, zipping/unzipping for reducing data transferring size on low bandwidth network, and server side persistent message queue. Note that the three features are optional.

5.2.1 Base and user requests processing

As described at the above Section 5.1.1, SocketPro server core must be able to process both base and user defined requests from clients. SocketPro processes all base requests internally without your code at all. It is reminded that all base requests have identification numbers that are less than idReservedTwo (0x2001) defined at the file ../socketpro/include/ucomm.h. On the other hand, all user requests, which always have identification numbers no less than the value idReservedTwo, are processed from your code. Besides, your code must return one or more results for each of user requests.

5.2.2 Online message bus and built-in service Notification

Like client core, server core has a built-in feature Notification implemented for you to quickly and conveniently use publish-subscribe pattern for exchanging various messages among online (or connected) clients from serve r side. You can use the feature to send (publish) any messages onto one or different chat or topic groups of connected clients from SocketPro server side. In addition, you are able to notify messages onto a client identified by its client login user id.

5.2.3 Compression and decompression

Similar to client core, SocketPro server core has this feature implemented for your convenience in case you like to reduce sending response result data size on low bandwidth network environment. Since compression is usually very CPU extensive, it is not recommended for LAN in general. At server side, SocketPro also supports two compression levels, best speed and best compression. The first one is focused on compression speed with less CPU cost, but the second one is focused on better compression rate with much higher CPU cost.

5.2.4 Sharable persistent message queue

Currently there are a lot of persistent message queue frameworks available, which save messages at server side only. However, SocketPro offers you persistent message queues at both client and server sides. Client persistent message queue is implemented for better fault tolerance as described at the Section 5.1.4, and server persistent queue for asynchronously offline consuming and processing with decoupled fashion. You can start any number of persistent message queues within one server application. Further, you can not only save messages from multiple providers or clients into each of these queues, but also make each of these queues de-queued by any number of consumers or clients simultaneously. Therefore, each of these queues is sharable at SocketPro server among providers and consumers. Finally, it is note that SocketPro persistent queue follows the rule of first-in-first-out.

5.2.5 HTTP/websocket built-in service

SocketPro server code supports both HTTP and websocket protocols. Therefore, you can access a SocketPro server from web browsers by use of SocketPro javascript adapter if this service is enabled.

5.2.6 Routing/load balancing and alpha requests

SocketPro server core has a special feature named as routing, which functions like load balancer or network device switch. As shown in the Figure 6, a large number of various requests from a set of clients can be routed onto a new set of clients or computers for processing in load balancing or parallel style.

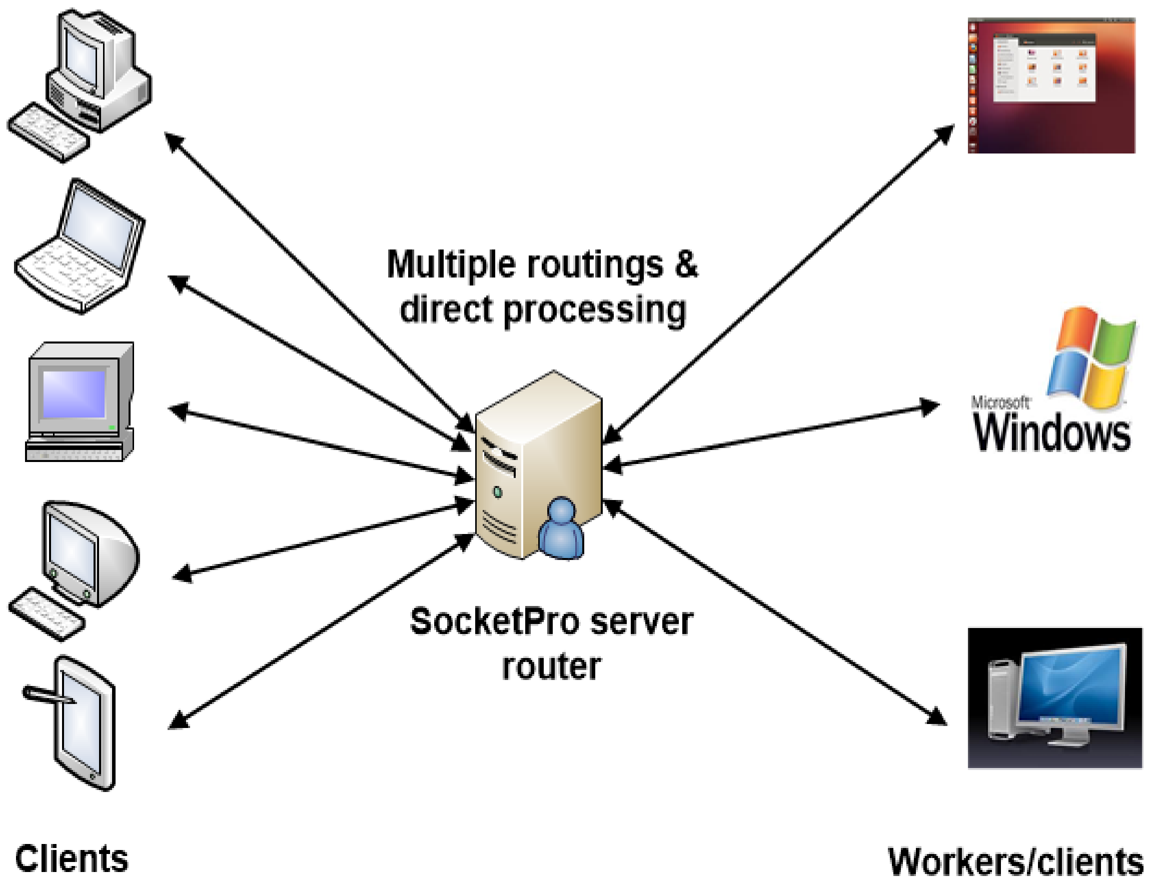


Figure 6: SocketPro server core routing/load balancing feature

Under many cases, you may like some of client requests should be directly processed at server side instead of routed clients. These requests are called as **alpha** requests.

SocketPro routing/load balancing feature supports **sticky sessions partially** by either transaction queue or manual request batching at client side. Therefore, you could use the two approaches to force a set of requests processed at one client together.

SocketPro routing/load balancing feature also supports **auto fault recovery** by client persistent message queue for any errors which are related with workers as described previously at the Section 5.1.4. In case a failure happens with anyone of workers, both client and SocketPro servers are able to detect it. Automatically, failed requests could be resent and re-routed onto new workers for processing from client request persistent queue.

Finally, SocketPro routing/load balancing supports three algorithms**, default, random and average** for scheduling which one of workers to send a request to. If you use algorithm average, SocketPro will simply routes requests onto different workers in average on the number of client requests. If you use algorithm random, SocketPro scheduling is internally based on session data transferring speed between SocketPro server and worker. The last algorithm, default, is actually the combination of the two algorithms, random and average.

One particular issue you must pay attention is that all response results are not returned sequentially from different workers. However, you can embed an index for each of requests and force all of workers to always return results containing such an index. Later, you can sort results according to the index.

As you can see, SocketPro routing/load balancing feature can be a simple but powerful server tool for you to easily improve your application system horizontal scalability.

5.2.7 User defined server services written from various development languages

In addition to SocketPro built-in services as described at the Sections 5.2.2 and 5.2.5, you are able to create your own services from your development languages with help of their adapters. At the moment, you could create your own server services with C/C++, .NET C#/VB.NET, Java and Python languages. Note that each of services must be registered with a unique identification number right before listening on port. A client must ask for a service based on its unique identification number before sending any other requests.

A SocketPro server can be registered with any number of services. Each of services will always support all of base requests and online message bus or notification requests by inheritance.

Service requests are classified into two categories, fast and slow requests. Fast requests are always processed within one or more pre-started threads. However, slow requests are dispatched onto worker threads for processing at run time. These worker threads are created at run time, and they are automatically killed when they are idle for a pre-defined time period. This design is created for reducing expensive thread context switches if there are lots of fast requests from client.

5.2.8 Plug-ins written from C/C++

SocketPro supports server side plug-in development. A SocketPro server plug-in is actually a system standard dynamic library, which must have one or more user defined services implemented with exposing the following required seven functions at least. When a SocketPro server loads a plug-in, it will use the below required functions to dispatch client requests onto your plug-in for processing.

- *bool WINAPI InitServerLibrary (int param); //The method will be called from SocketPro server core right after the library is loaded*
- *void WINAPI UninitServerLibrary(); //The method will be called from SocketPro server core right before the library is going to be unloaded*
- *unsigned short WINAPI GetNumOfServices(); //SocketPro will use the method to query how many services the library has defined*
- *unsigned int WINAPI GetAServiceID(unsigned short index); //The method will be called from SocketPro server core to query each service id on zero-based index*
- *CSvsContext WINAPI GetOneSvsContext(unsigned int serviceId); //The method will be called from SocketPro server core to get service context for a given service id*
- *unsigned short WINAPI GetNumOfSlowRequests(unsigned int serviceId); //The method will be called from SocketPro server core to query the number of slow requests for one service id*
- *unsigned short WINAPI GetOneSlowRequestID(unsigned int serviceId, unsigned short index); //The method will be called from SocketPro server core to get a slow request id from given service id and zero-based index*

Currently, you can create SocketPro server plug-ins only from C/C++.

5.3 Replication of one queue of messages to others

As described at the above two sections 5.1 and 5.2, both client and server sides support persistent message queue. SocketPro client queue is used for request backup so that a request can be resent to server in case there is any error such as network failure, server application shutdown and power-off. SocketPro server queue, like other typical queues, provides an asynchronous communication between publishers and subscribers which do not need to interact with message queue at the same time. Further, SocketPro is also able to replicate one queue of messages onto others as shown in the below Figure 7 at both client and server sides.

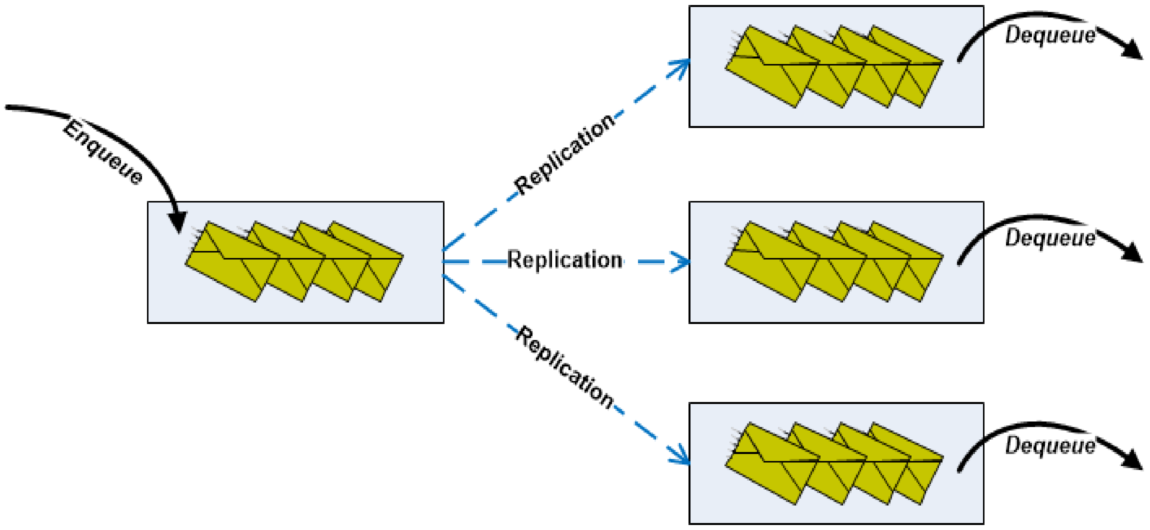


Figure 7: One source queue of messages replicated onto three target queues with transaction style

It is noted that SocketPro queue replication is fully in agreement of all ACID (atomicity, consistency, isolation and durability) properties, and guarantees complete consistency or the exactly same data among all replicated copies.

5.4 Protected password

A client must always send its user id and password before sending any other requests. Internally SocketPro always clean password string as soon as possible at both client and server side. Further, SocketPro does special treatment to clean memory traces of password as soon as possible for better security.

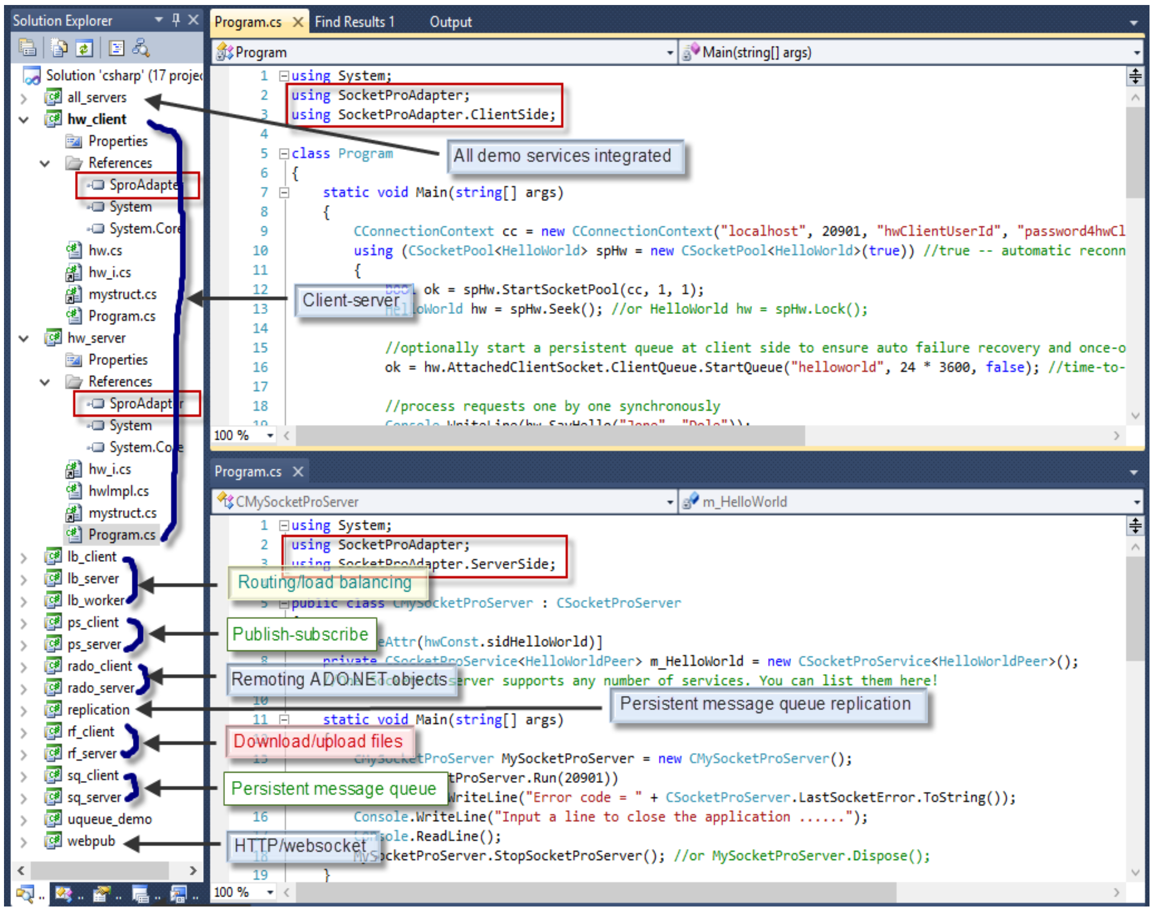
6. SocketPro tutorials

Previous sections are focused on discussion about SocketPro design and its features. It is time to experience SocketPro by playing a set of tutorial examples.

6.1 IDE (integrated development environment) tools

First of all, let’s talk about IDE tools used by UDAParts, but you can use any other tools you like.

SocketPro supports Microsoft visual studio 2008 or later for development on window platforms. However, visual studio 2010 is used under most cases as shown in the Figure 8.



```
using System;
using SocketProAdapter;
using SocketProAdapter.ClientSide;

class Program
{
    static void Main(string[] args)
    {
        CConnectionContext cc = new CConnectionContext("localhost", 20901, "hwClientUserId", "password4hwCl
        using (CSocketPool<HelloWorld> spHw = new CSocketPool<HelloWorld>(true)) //true -- automatic reconn
        {
            bool ok = spHw.StartSocketPool(cc, 1, 1);
            HelloWorld hw = spHw.Seek(); //or HelloWorld hw = spHw.Lock();

            //optionally start a persistent queue at client side to ensure auto failure recovery and once-o
            ok = hw.AttachedClientSocket.ClientQueue.StartQueue("helloworld", 24 * 3600, false); //time-to-

            //process requests one by one synchronously
```

```
using System;
using SocketProAdapter;
using SocketProAdapter.ServerSide;

public class CMySocketProServer : CSocketProServer

                    eAttr(hwConst.sidHelloWorld)]
    private CSocketProService<HelloWorldPeer> m_HelloWorld = new CSocketProService<HelloWorldPeer>();
                              erver supports any number of services. You can list them here!

    static void Main(string[] args)

        CMySocketProServer MySocketProServer = new CMySocketProServer();
                           tProServer.Run(20901))
                           WriteLine("Error code = " + CSocketProServer.LastSocketError.ToString());
        Console.WriteLine("Input a line to close the application ......");
        Console.ReadLine();
        MySocketProServer.StopSocketProServer(); //or MySocketProServer.Dispose();
    }
```

Figure 8: A visual studio 2010 solution for all SocketPro .NET C#/VB.NET tutorial projects

Figure 8 shows a visual studio 2010 solution containing all SocketPro .NET C# tutorial projects on window platforms. Additionally, the picture is labeled with demonstrated features. We also use the IDE tool for C++ development on window platforms as shown at the Figure 9.

In fact, these .NET tutorial projects can also be compiled within IDE tool monodevelop on mono and .NET framework. Compiled managed applications are just able to run on Linux platforms.

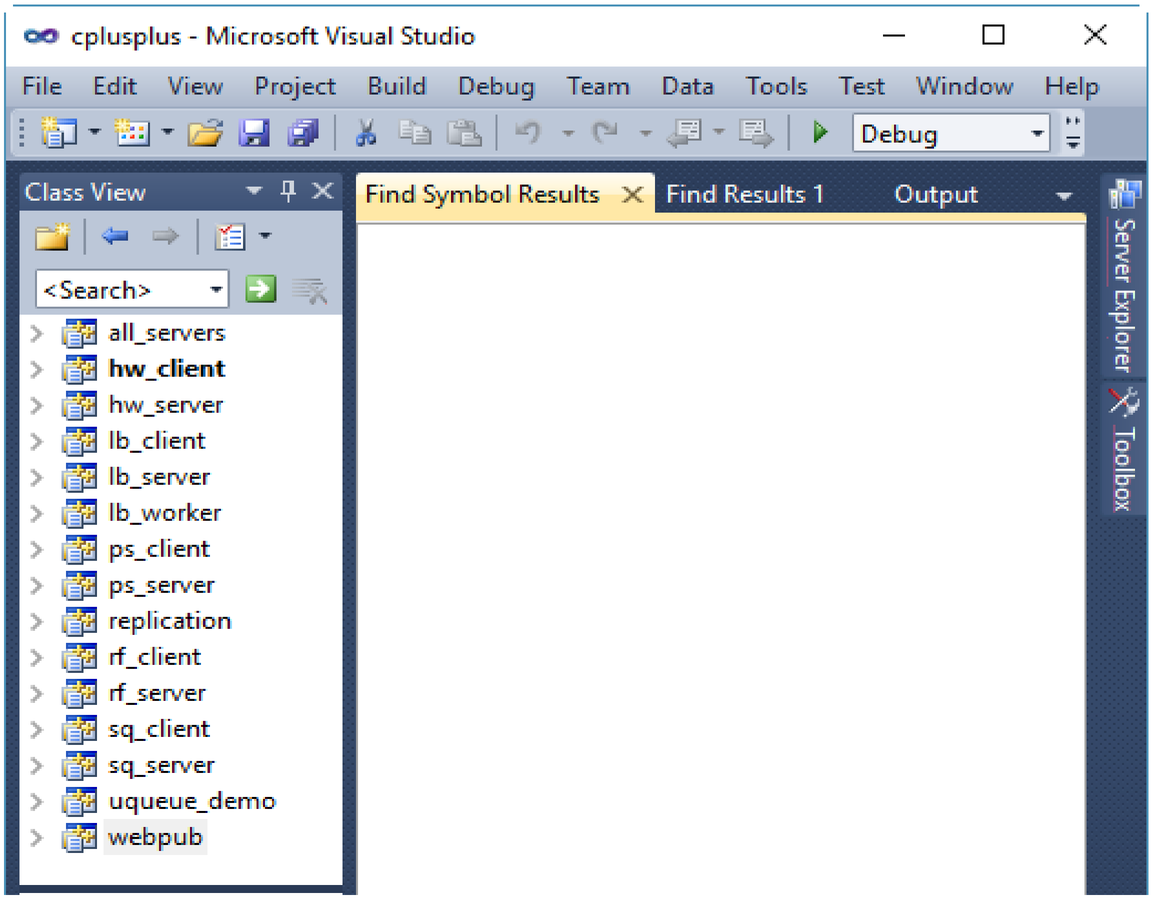
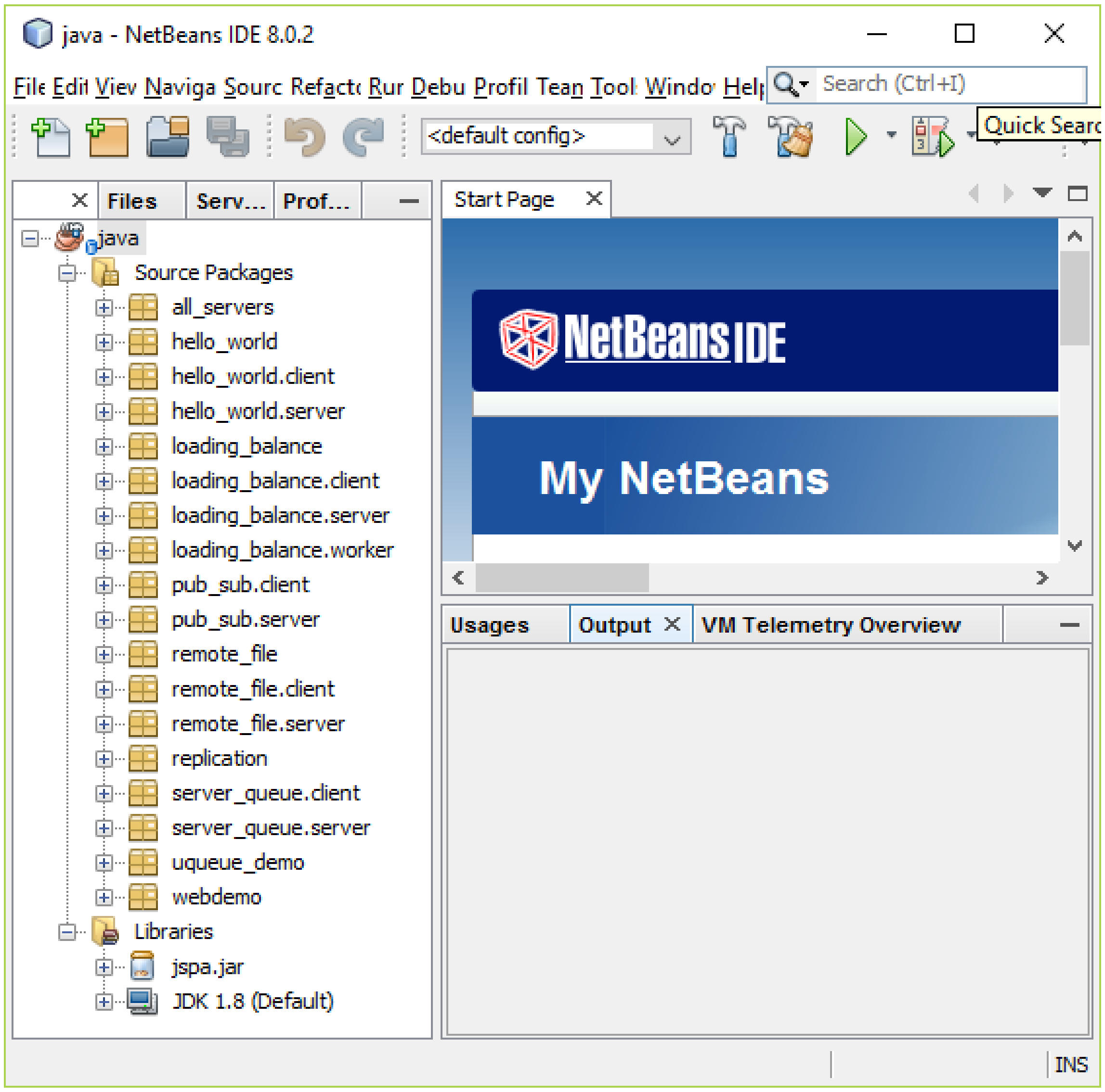


Figure 9: A visual studio 2010 solution for all SocketPro C++ tutorial projects on window platforms

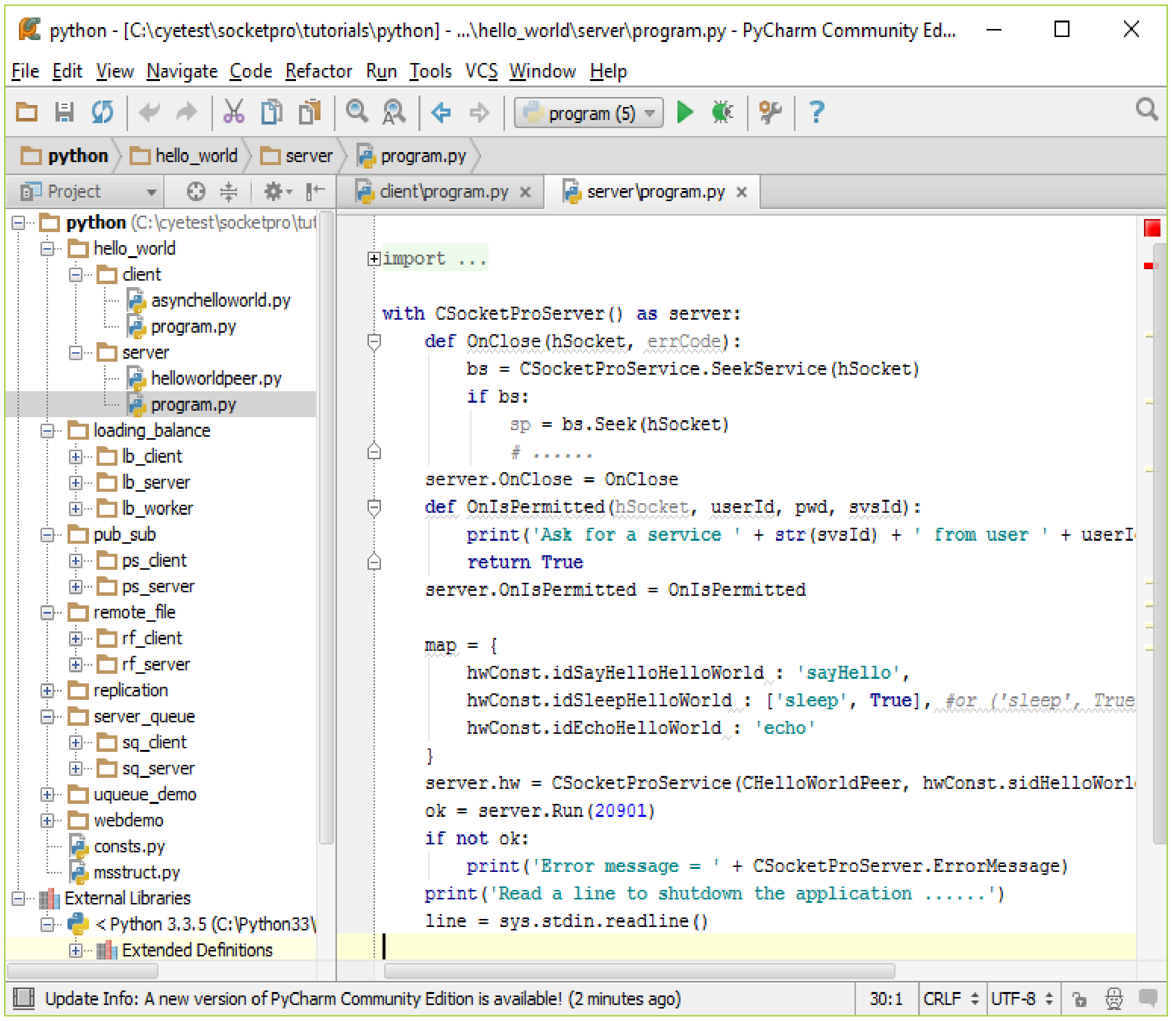
In regards to Java environment, Netbeans (https://netbeans.org) is employed on both window and Linux platforms as shown in the Figure 10. Further, we also use the IDE tool Netbeans for C++ demo projects on Linux platforms.



```
java
  Source Packages
    all_servers
    hello_world
    hello_world.client
    hello_world.server
    loading_balance
    loading_balance.client
    loading_balance.server
    loading_balance.worker
    pub_sub.client
    pub_sub.server
    remote_file
    remote_file.client
    remote_file.server
    replication
    server_queue.client
    server_queue.server
    uqueue_demo
    webdemo
  Libraries
    jspa.jar
    JDK 1.8 (Default)
```

Figure 10: A Netbeans solution for demonstration of various SocketPro features on Java platform

We use PyCharm (https://www.jetbrains.com/pycharm) for Python development on all platforms. Its free Community Edition works just fine as shown in Figure 11.



```
import ...

with CSocketProServer() as server:
    def OnClose(hSocket, errCode):
        bs = CSocketProService.SeekService(hSocket)
        if bs:
            sp = bs.Seek(hSocket)
            # ......
    server.OnClose = OnClose
    def OnIsPermitted(hSocket, userId, pwd, svsId):
        print('Ask for a service ' + str(svsId) + ' from user ' + userI
        return True
    server.OnIsPermitted = OnIsPermitted

    map = {
        hwConst.idSayHelloHelloWorld : 'sayHello',
        hwConst.idSleepHelloWorld : ['sleep', True], #or ('sleep', True
        hwConst.idEchoHelloWorld : 'echo'
    }
    server.hw = CSocketProService(CHelloWorldPeer, hwConst.sidHelloWorl
    ok = server.Run(20901)
    if not ok:
        print('Error message = ' + CSocketProServer.ErrorMessage)
    print('Read a line to shutdown the application ......')
    line = sys.stdin.readline()
```

Update Info: A new version of PyCharm Community Edition is available! (2 minutes ago)

Figure 11: A Python solution for demonstration of various SocketPro features on Python environment

It is noted that all of these tutorial applications are compatible to each other on all development environments and platforms.

6.2 SocketPro cross-platform and cross-language compatibility

SocketPro has great compatibility across both platforms and development languages by use of a help class CUQueue at adapter level. Refer to the brief article at the file memoryqueue.pdf.

Once reading through the article, it is expected that you are able to answer the follow questions.

- How to pack or unpack primitive data types like **int, long, short, byte, float, double, bool, UUID, decimal, and so on** into or from an instance of CUQueue
- How to pack or unpack **ASCII string** into or from CUQueue
- How to pack or unpack **Unicode string** into or from CUQueue across platforms and languages. It is noted that each Unicode char size is 2 and 4 on windows and Linux platforms, respectively
- How to pack or unpack date time by use of CUQueue
- How to pack or unpack **Java Object, .NET object, C/C++ variant and Python object**

- How to pack or unpack a **complex data structure**
- How to use **CScopeUQueue** for better performance by reducing memory allocating and deallocating repeatedly

6.3 SocketPro user-defined services

SocketPro supports unlimited number of user-defined services hosted within one SocketPro server application. Certainly, A SocketPro is able to access them simultaneously and concurrently by use of multiple socket pools. From the view of communication pattern or model, SocketPro supports four patterns such as client-server, publish-subscribe, persistent message queue, and routing/load balancing. Refer to the labels in the previous Figure 8.

SocketPro have four tutorial samples for each of these patterns. It is pointed out that there is no other communication framework in the world is able to support the above four communication patterns integrated at the same time!

6.3.1 Client-server pattern – hello_world or hw

Refer to the file **helloworld.pdf**. Once reading through and exercising the sample, you'll be able to figure out the following questions.

- How to use a UID file to quickly create a skeleton code of client and server for C++, C# and VB.NET development environment
- How to tell server core library if a request is a slow one and need long time to process within a worker thread at server side
- How to start a simple socket pool and make a connection from client and server
- How to transfer a complex structure between client and server as described previously at the Section 6.2
- How to convert an asynchronous request into a synchronous one
- Why asynchronous requests are recommended by default within SocketPro communication framework
- How to reduce round-trips by asynchronous requests
- How to make fault tolerance for any issues like network down, server application down, power-off, and so on by use of persistent message queue at client side

6.3.2 Publish-subscribe pattern – pub_sub or ps

This pattern is very useful for communicating among connected clients by a SocketPro server application in middle. The tutorial focuses the two categories of subjects, **online message bus** and use of **SSL3/TLSv1.x** for secure communication. Refer to the file **messagepush.pdf**.

Online messaging: After playing with this tutorial sample, you will know:

- How to create a set of chat groups or topics at server side
- How to subscribe or unsubscribe one or more groups or topics of messages
- How to send any message to one or more groups of subscribers
- How to send a message to one specific receiver by its user id (case-insensitive)
- How to track coming various online messages

It is noted that you can make the above calls at either client or server side.

SSL3/TLSv1.x: You can use the attached certificate and key files to test the SocketPro security feature. After playing with the sample, you must figure out the following questions for building an industrial standard secure distributed system.

- How to enable SSL3/TLSv1.x with a SocketPro server application on both windows and Linux platforms by use of certificate, key and password
- How to start a SSL3/TLSv1.x-enabled connection from client side
- How to look up and authenticate a certificate from server against a root stored certificate at client side
- How to verify certificate chain and prevent possible man-in-middle attack

6.3.3 Persistent message queue – server_queue or sq

UDAParts has a free server plug-in for high performance persistent message queue service. Refer to the tutorial documentation file **serverqueue.pdf** for details. The sample answers the following questions.

- Brief introduction about SocketPro asynchronous message queue plug-in
- How to load SocketPro asynchronous message queue plug-in at server side
- How to enqueue messages into a remote SocketPro server queue by queue key (or queue file name) and message identification numbers
- Why each of messages should be given whenever a message is sent from a client
- How to dequeue messages and parse messages according to message ids
- How to improve the throughput of message dequeuing

After playing with the sample projects and looking at client queue class methods, you will see you can open multiple queues with one server application.

Since you can start one or more pools of sockets connecting with different servers from one client or provider, you can send messages to different servers for en-queuing them onto different queues in parallel. Similarly, one consumer or client is also able to create one or more pools of sockets connecting with different servers for de-queuing messages from different queues in parallel.

At last, SocketPro server queues are sharable among providers and consumers. Multiple providers and consumers are able to access one SocketPro server queue at the same time.

6.3.4 Routing/load balancing – loading_balance or lb

SocketPro server supports routing requests from different clients onto different workers (or clients) for processing them in load balancing way. The sample is created based on calculating π value by use of trapezoidal integration. For details, refer to the tutorial documentation file **urouter.pdf**. After exercising the sample, you will know the following how-tos.

- How to make two services routed against each other at server side
- How to send requests from a client
- How to process requests from client at worker side, process them and return responses back to source client

- How to collect results from workers
- How to ensure auto failures recovered at client side

At last, it is worth pointing out that all results from workers are randomly returned. The response order may be different from request sending order.

6.4 Replication

As described at the Sections 5.1.4 and 5.2.4, SocketPro both core client and server libraries have persistent message queue implemented. In addition, SocketPro queue supports both en-queuing and de-queuing with transaction style. Further, SocketPro supports replicating messages within one queue onto different queues in agreement with all ACID properties, and guarantees complete consistency among all copies of replicated data. For details, refer to the short article **replication.pdf**.

After playing the simple tutorial applications, you will know the following how-tos.

- How to set a set of target machines which a root or source queue of messages will be replicated to
- How to create an instance of replication object for replicating queue
- How to replicate a set of messages onto different target queues

It is noted that the tutorial sample is created from SocketPro client persistent message queue. Actually, it is also very simple to do queue replication at server side. Refer the two methods AppendTo and EnsureAppending of the server side class CServerQueue.

6.5 HTTP/websocket – webdemo

As described at the Section 5.2.5, SocketPro has a built-in HTTP/websocket service at server side. You can access a SocketPro server by use of HTTP and/or websocket protocols anywhere. Refer to the short article **socketproweb.pdf**. After playing with the tutorial, you are expected to answer the following questions

- How to refer to SocketPro Javascript adapter at server side
- How to handle HTTP RFC-2616 methods, OPTIONS, GET, POST, HEAD, PUT, DELETE, TRACE and CONNECT.
- How to deal with HTTP/websocket user defined requests
- How to track HTTP/websocket online message pushes at server side
- How to refer to uloader.js for loading SocketPro javascript adapter from server to browser
- How to tracking SocketPro javascript adapter loading event inside browser
- How to build a connection from browser to remote SocketPro server with tracking communication events
- How to subscribe for messages and publish messages with browser
- How to send a message from web browser onto others by a SocketPro server in middle
- How to enable SocketPro javascript adapter debug information

6.6 Remoting file – remote_file or rf

Exchanging files between client and server is a common task for a software engineer, but it is usually difficult to write code for downloading and uploading files efficiently. SocketPro adapters already have utility classes implemented for you to reuse them for the common task at the fastest speed. For detail, refer to the file **remotingfile.pdf**.

The tutorial sample answers you the following questions:

- How to use the class CStreamHelper for both downloading and uploading files at server side
- How to use the class CStreamHelper for both downloading and uploading files at client side
- How to download files from server to client
- How to set callback for tracking file downloading progress
- How to upload files from client to server
- How to set callback for tracking file uploading progress
- How to make sure downloading and uploading completed before executing other requests

In addition to the above answers, the tutorial also answers the following questions:

- What SocketPro server threading model is
- What fast and slow requests are at server side
- What main and worker threads are at server side
- How SocketPro server core library dispatches request onto main and worker threads for processing
- What socket pool threading model is at client side
- When we should use a socket pool having multiple threads at client side

6.7 Remoting ADO.NET objects, DataReader, DataTable and DataSet

ADO.NET objects DataReader, DataTable and DataSet are widely used almost by all of applications as long as these applications are involved with database accessing. If these applications require exchanging these objects between client and server, both performance and scalability are highly dependent on data communication speed if there are a considerable number of data records returned. SocketPro .NET adapter has helper classes ready for you to download and upload them from and onto a remote SocketPro server at the fastest speed no matter how many records are involved. We have performance studies completed, and found that SocketPro is never slower than WCF (window communication framework), and usually two to eight times faster than WCF which depends on the number of involved records. The more records the larger SocketPro improvement will be. Refer to the file **scalableweb.pdf**.

The tutorial sample answers you the following questions:

- How to use the class CAdoClientPeer for exchanging ADO.NET objects DataReader, DataTable and DataSet at server side
- How to use the method Send for pushing these ADO.NET objects from server to client
- How to use the class CAsyncAdohandler for exchanging ADO.NET objects DataReader, DataTable and DataSet at client side

- How to start a client socket pool having multiple threads and non-blocking sockets connected to different remote servers
- How to bring one or more sets of records from server to client synchronously one by one
- How to bring one or more sets of records from server to client asynchronously in parallel

For simplicity, the tutorial sample does not show how to send these ADO.NET objects from client to server. However, it is very simple and expected that you can easily figure out to use the method Send of the class CAsyncAdohandler for the purpose.

6.8 SocketPro server having multiple services – all_servers

Most of the previous tutorial server samples show one service only, as we like to make these tutorial samples as simple and understandable as possible. In fact, a SocketPro server application is able to handle any number of different services as long as it has enough resources. To show you the power and rich-feature of SocketPro, we put all of different tutorial services into one server application.

In addition to the above tutorial services, you can also load SocketPro server plug-ins into one server application. In regards to server plug-in, refer to the next Section 7 for details.

By this time, it is expected you are able to start understanding the power of SocketPro. Most of communication frameworks just have one specific communication pattern implemented. However, SocketPro supports all of the patterns such as client/server, publish/subscribe, persistent message queue, and routing/load balancing at the same time within one central server. You can easily integrate all of them into one SocketPro server application with much less dependency and simpler configuration settings.

An enterprise application system may be involved with different communication patterns, which may force you to spend lots of time and human resources just for integrating different frameworks together. The resulting system may be superior complex and venerable to various hidden pitfalls because all software users and engineers in a team must fully understand both strengths and weaknesses of all involved individual frameworks. Further, integrating different frameworks may significantly degrade the performance and scalability of the resulted application system. If you use SocketPro as a complete solution, your enterprise application must be much simpler, faster and more scalable.

7. SocketPro server plug-in development

Previous tutorial samples show you how to create user-defined services from your development languages. This way is usually fine. Under many cases, we like to create one or more services at server side and wish them are reusable and accessible to both servers and clients from different development language environments. Currently, you can create SocketPro server plug-ins, which are dynamic system libraries written with C/C++.

One plug-in may contain multiple SocketPro services implemented, although most of them just have one service only. Once a SocketPro server plug-in is available, you can load it from a SocketPro adapter into your server application by calling the method CSocketProServer.DllManager.AddALibrary.

7.1 Seven required functions

As shown at the previous Section 5.2.8, a SocketPro server plug-in must expose the seven methods, InitServerLibrary, UninitServerLibrary, GetNumOfServices, GetAServiceID, GetOneSvsContext, GetNumOfSlowRequests and GetOneSlowRequestID. When you call the method CSocketProServer.DllManager.AddALibrary, SocketPro server core library will load your plug-in library and seek the above required seven functions. If anyone of them is not available, the call will fail. At the beginning, the server core library will call the method InitServerLibrary to initialize your plug-in.

Afterwards, SocketPro server core library will keep on using the method GetNumOfServices to find how many services a plug-in has. Next, the core library obtains a service context structure by use of the methods GetAServiceID and GetOneSvsContext. It is noted that a service context structure mainly contains a number of callbacks although you don't access it from your code.

Next, the server core library finds how many slow requests within each of services by use of method GetNumOfSlowRequests. At the end, the core library uses the method GetOneSlowRequestID to register slow requests for all services on behalf of plug-in automatically.

The server core library may call the method UninitServerLibrary so that you can release resources allocated within plug-in library if you call the method CSocketProServer.DllManager.RemoveALibrary. However, this call can be always ignored.

UDAParts has prepared four sample plug-in samples for you to study and help at directory ../socketpro/samples/module_sample.

7.2 HelloWorld – the simplest SocketPro server plug-in

The first SocketPro server plug-in is created by reuse of the tutorial 1 server side code. It is recommended that you need to study this plug-in sample first as it is the simplest one among all four plug-in samples.

All required seven functions are implemented at the file ../ socketpro/samples/module_sample/HelloWorld/helloworld.cpp.

7.3 usqlite – a client server database for SQLite

This server plug-in is professional, and can be directly reused within your enterprise application systems for free. The source code is actually located at the directory ../socketpro/include/sqlite although its project file is inside the directory ../socketpro/samples/module_sample/usqlite/ssqlite.

The plug-in solution is at the directory ../socketpro/samples/module_sample/usqlite, which contains all test source codes. The free precompiled plug-in libraries for both windows (ssqlite.dll) and Linux (libssqlite.so) platforms are located inside the directory ../socketpro/bin.

As you know, SQLite is very popular database engine by use of local database files but it is serverless. Therefore, it is not accessible from a remote computer. UDAParts has made SQLite as client-server database so that you can use a single but unified API to access remote SQLite database files through the server plug-in relying on SocketPro asynchronous communication framework.

SocketPro adapter has a new class CAsyncDBHandler for you to reuse at client side. After playing with client test source codes, you will find the class is designed with continuous inline request/result batching, real-time stream processing, asynchronous data transferring and parallel computation in mind.

The simple API offers the following advantages:

- Simple for reuse
- Support of both windows and Linux platforms
- Power and rich features with many unique ones because of its unique design
- Superior performance and scalability
- Asynchronous SQLite service is totally free

Finally, you can base the plug-in source code to create your own one for your specific requirements.

7.4 umysql – a database broker for MySQL and Mariadb

In addition to SQLite, UDAParts has created a new plug-in for popular open-source databases MySql and Mariadb. The plug-in is professional too, and can be directly reused within your enterprise application systems for free. The source code is actually located at the directory ../socketpro/include/mysql, although its project file is inside the directory ../socketpro/samples/module_sample/umysql/smysql.

The plug-in solution is at the directory ../socketpro/samples/module_sample/umysql, which contains all test source codes. The free precompiled plug-in libraries for both windows (smysql.dll) and Linux (libsmysql.so) platforms are located inside the directory ../socketpro/bin.

It is noted that the plug-in is initially created for accessing MySql/Mariadb embedded version by use of SocketPro asynchronous communication framework instead of MySql/Mariadb native communication protocols. A client is able to use SocketPro continuous inline request/result batching algorithm, real-time stream processing and asynchronous data transferring for accessing databases at the fastest speed. UDAParts has done the plug-in performance studies against standard MySql/Mariadb client/server databases on window platforms. You can find a performance comparison projects at the directory ../socketpro/samples/module_sample/umysql/DBPerf. Our study results show that SocketPro plus MySql/Mariadb embedded could be up to three times faster than standard MySql/Mariadb database in querying records.

SocketPro MySql/Mariadb plug-in also supports standard MySql/Mariadb database. If you use this plug-in, it will be a database accessing broker on behalf of clients.

At client side, you use SocketPro adapter class CAsyncDBHandler for accessing MySql/Mariadb databases like SQLite through SocketPro SQLite plug-in. After comparing test samples of SocketPro MySql/Mariadb against SQLite plug-in, you will find they are almost the same. You can rely on two plug-ins to execute various SQL statements in batch, request stream, asynchrony and parallel computation for the best performance and scalability.

Finally, you can also base the plug-in source code to create your own one for your specific requirements.

7.5 uasyncqueue – SocketPro server persistent queue

Message queues are extremely useful under a number of situations. For example, when we want to execute a series of tasks asynchronously, a provider sends them to a centric remote machine for saving (or en-queuing) them onto a disk. Sometime later, a consumer read them from the centric remote machine for processing. Persistent message queue enables asynchronous message processing and eliminates many application system tough problems such as network instability, message loss, consumer/provider offline, application abnormal shutdown, and so on. Both provider and consumer are not required to be connected at the same time. Since a queue can be sharable among multiple providers and consumers, persistent queue could be a great way to improve system scalability by parallel enqueuing and dequeuing. Both SocketPro client and server core libraries have persistent message queue implemented as described at Sections 5.1.4 and 5.2.4. UDAParts has created a server plug-in to further simplify application of SocketPro server persistent message queue to your application systems.

Note that the plug-in is free and can be directly reused within your enterprise application systems. The source code is actually located at the directory ../socketpro/include/uasyncqueue, but its project file is inside the directory ../socketpro/samples/module_sample/uasyncqueue/uasyncqueue.

Like the previous two plug-ins, this plug-in solution is at the directory ../socketpro/samples/module_sample/uasyncqueue, which contains all test source codes. The free precompiled plug-in libraries for both windows (uasyncqueue.dll) and Linux (libuasyncqueue.so) platforms are located inside the directory ../socketpro/bin. In addition to these test examples, you can also refer to the Section 6.3.3 and file **serverqueue.pdf** for details and its usages.

SocketPro adapter has a new class CAsyncQueue for you to enqueue messages at provider side and dequeue messages from consumer sides. In addition, these messages can be enqueued in transaction style. One SocketPro server application is able to support multiple queues simultaneously. In regard to one queue, multiple providers are able to enqueue various messages simultaneously onto the queue, while multiple consumers are able to concurrently dequeue these messages from the queue at the same time.

Finally, you can also base the plug-in source code to create your own one for your specific requirements.

8. Real-time cache update -- online message push for database event

As described at the Section 5.1.2, SocketPro client core has message bus feature implemented. You can push various messages from a client onto a SocketPro server application and other clients. Under this situation, the server application is used as a message dispatcher and the dispatcher routes these messages onto other targeted clients. It is noted that all clients must be online and connected to the central SocketPro server application. Therefore, this feature is called online message bus push.

Caching is technique to improve system both performance and scalability as it may eliminate a large portion of expensive network roundtrips and database queries. It also allows flexibility in the processing of data. Further, caching can improve availability of data by providing continued service for applications that depend only on cached tables even if the backend server is unavailable.

Many relational databases support triggers at backend server sides. We could use these triggers to track various data table change and other database events. It would be very useful if we have a generic way to track these events and send them onto a remote server application so that the server application is able to update its cache corresponding to obtained event types in real-time fashion. Many databases like MySql and Mariadb don't have this feature ready to use.

8.1 Plug-ins for databases MySql, Oracle, MS SQL and DB2

UDAParts has created four databases plug-in on SocketPro online message push for this purpose. You could find them at the directory ../socketpro/dbupdate. In addition to the four plug-in libraries, there is a simple base library (udbubase) for online message push. You could use it to notify any events from any applications other than database systems. To use the libraries, you need to complete the following three steps:

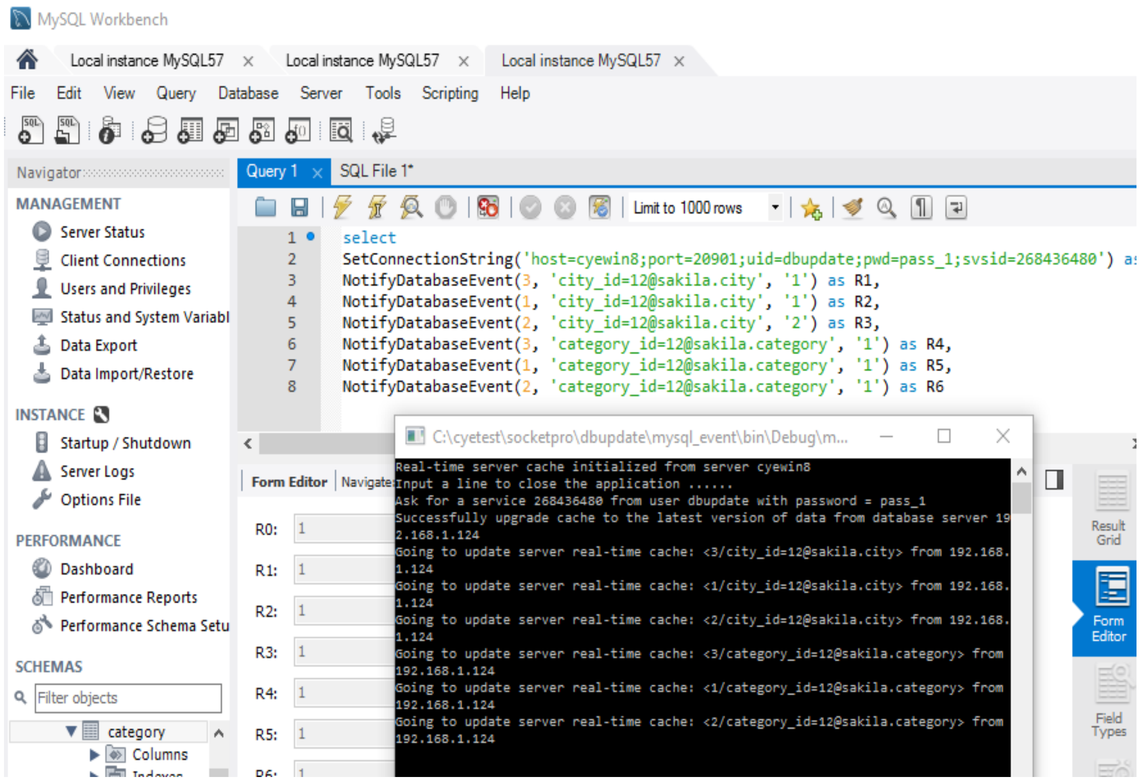
- Register a database plug-in library for one database system after consulting its documentation for how to create user defined functions specifically
- Create three user defined functions, SetConnectionString, NotifyDatabaseEvent, and GetConnections. For detail, refer to registration.sql for each of database systems
- Create data table change triggers, and refer to these functions inside triggers

The directory ../socketpro/dbupdate currently contains the following projects:

1. spdb2push: a project for creating DB2 server plug-in
2. spmssqlpush: a project for creating Microsoft SQL server plug-in
3. spmysqlpush: a project for creating MySql/Mariadb server plug-in
4. sporaclepush: a project for creating Oracle server plug-in
5. test_udbubase: a project for demonstration of usages of the base library udbubase
6. udbubase: a project for creating basic real-time notification library reusable by any applications other than the above four databases
7. mysql_event: a sample SocketPro server real-time cache implemented on MySql/Mariadb using database sakila

8.2 SocketPro server real-time cache implementation on MySql

This is a sample at the directory ../socketpro/dbupdate/mysql_event for how to update server cache in real-time on MySql sample database sakila by use of SocketPro online message push. See the below Figure 12.



```
select
SetConnectionString('host=cyewin8;port=20901;uid=dbupdate;pwd=pass_1;svsid=268436480') a
NotifyDatabaseEvent(3, 'city_id=12@sakila.city', '1') as R1,
NotifyDatabaseEvent(1, 'city_id=12@sakila.city', '1') as R2,
NotifyDatabaseEvent(2, 'city_id=12@sakila.city', '2') as R3,
NotifyDatabaseEvent(3, 'category_id=12@sakila.category', '1') as R4,
NotifyDatabaseEvent(1, 'category_id=12@sakila.category', '1') as R5,
NotifyDatabaseEvent(2, 'category_id=12@sakila.category', '1') as R6
```

```
Real-time server cache initialized from server cyewin8
Input a line to close the application ......
Ask for a service 268436480 from user dbupdate with password = pass_1
Successfully upgrade cache to the latest version of data from database server 19
2.168.1.124
Going to update server real-time cache: <3/city_id=12@sakila.city> from 192.168.
1.124
Going to update server real-time cache: <1/city_id=12@sakila.city> from 192.168.
1.124
Going to update server real-time cache: <2/city_id=12@sakila.city> from 192.168.
1.124
Going to update server real-time cache: <3/category_id=12@sakila.category> from
192.168.1.124
Going to update server real-time cache: <1/category_id=12@sakila.category> from
192.168.1.124
Going to update server real-time cache: <2/category_id=12@sakila.category> from
192.168.1.124
```

Figure 12: SocketPro server real-time cache on MySql sample database sakila

The above sample cache implementation supports real-time data cache from different databases across MySql servers. It is noted that the sample code is very simple and you can easily write your own server real-time caches for other database and non-database systems.

9. Performance and scalability studies

UDAParts designed and created SocketPro with performance and scalability in mind after studying common but fundamental communication issues. During development, UDAParts has been writing a number of performance test samples against the world top performant frameworks to improve the qualities of SocketPro framework.

9.1 Performance

The first performance study against WCF and RabbitMQ is documented at the file perf_comparison.pdf. Our study results are:

- **SocketPro vs WCF**: First of all, SocketPro could be easily 100 times faster than WCF for simple client/server requests if SocketPro is used with asynchronous request batching. SocketPro could be easily 50% or more faster than WCF if it is used in synchronous communication without request batching. UDAParts has never seen a case that WCF is faster than SocketPro!
- **SocketPro persistent queue vs RabbitMQ**: First of all, RabbitMQ will NEVER catch the performance of SocketPro under any situation. The improvements of SocketPro over RabbitMQ are dependent on message size. For high volume of small messages (200 bytes), SocketPro could

be easily over 50 times faster. For large messages (10 Kilobytes), SocketPro will be still 10 times faster.

In addition to the above study, UDAParts doesn't ignore Apache kafka persistent queue. UDAParts has been internally comparing SocketPro persistent queue with Apache kafka a lot during development as documented here. In short, SocketPro could be 200% faster than kafka. In regards to large size of messages, SocketPro could be still easily 20% faster than kafka. SocketPro persistent message queue performance studies for different languages can be found at the directory ../socketpro/samples/qperf.

Finally, UDAParts has done performance study for SocketPro asynchronous MySql/Mariadb service. You can find the performance study code at the directory ../socketpro/samples/module_sample/umysql/DBPerf. The performance is completed on the asynchronous service plus embedded MySql against standard MySql server by use of popular sample database sakila. Our studies show that MySql SocketPro asynchronous database service could be 300% faster than standard MySql server for high volume of faster queries. UDAParts has never found a case that standard MySql server is faster than SocketPro asynchronous database service even if SocketPro is used in synchronous communication without any batching.

9.2 Scalability

Unlike performance study, creating a complete system for scalability is tough as the system requires lots of machines and initial startup involved. Therefore, UDAParts has not completed a document yet. As you read through the development guide, you are expected to know that SocketPro has many features implemented for scalability as shown in the below Figure 13.

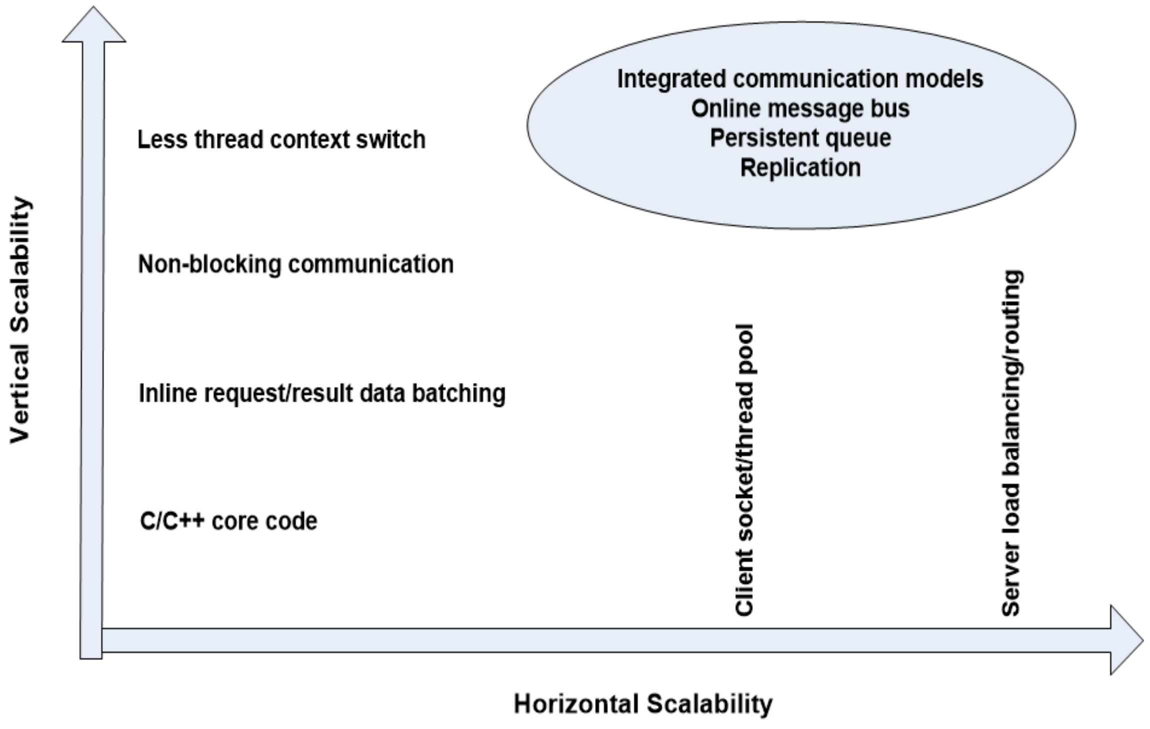

Vertical Scalability
Less thread context switch
Non-blocking communication
Inline request/result data batching
C/C++ core code
Integrated communication models
Online message bus
Persistent queue
Replication
Client socket/thread pool
Server load balancing/routing
Horizontal Scalability

Figure 13: SocketPro features for scalability

In addition to the features on the above Figure 13, you can also access SocketPro server from any devices or machines through SocketPro built-in HTTP/Websocket service. This is also a kind of scalability though.

At last, you could use real-time cache for better scalability as described at the Section 8.

10. Other examples

In addition to tutorial samples and samples mentioned previously in the guide, there are two more samples for you to study.

- mssqlpush: The sample is created for tracking various MS SQL triggers and events and pushing interesting data onto different SocketPro servers consistently by replicating a persistent queue of messages in transaction style. The sample is created on C#.
- ws_study: The sample is originally created for studying websocket protocol. The sample is similar to the tutorial webdemo as described at the Section 6.5.

11. SocketPro key selling points

After reading through the above sections, SocketPro has a number of attractive features.

11.1 Asynchronous processing by default

- Non-blocking sockets are used exclusively at both client and server sides within SocketPro framework.
- All requests are always sent from client side and processed at server side asynchronously.
- Client request sending, server request processing and result returning could happen concurrently at the same time.
- Lambda expression, anonymous delegate, closure, async/await, and so on are extensively used at client side programming with help of SocketPro adapters.
- All asynchronous requests can be converted into synchronous ones.

11.2 Inline data batching for the best throughput and lowest latency

- Inline continuous data batching algorithm is extensively and silently used at both client and server sides.
- Both network efficiency and latency are well improved with SocketPro.
- Manual batching is supported although inline continuous data batching is defaulted at both client and server sides.

11.3 Multiple communication patterns integrated for complex enterprise systems

- Client server communication pattern.
- Publish subscribe pattern.
- Asynchronous persistent message queue pattern.
- Server side routing and load balancing pattern.

11.4 SSL3/TLSv1.x used for secure communication

- Secure channel is used for security on windows platforms.
- Openssl library is employed for securing communication on non-windows platforms.

11.5 Low coupling, cross-language and cross-platform developments

- All requests are solely de-coupled between client and server sides.
- Cross-platform communicating is fully supported.
- Data transferring among different development languages and operation systems are fully compatible.

11.6 Great performance and scalability

- SocketPro inline continuous data batching algorithm at both client and server sides significantly improves request processing speed across all types of networks.
- Native C/C++ codes within core libraries help SocketPro performance.
- SocketPro both client and server sides show less thread context switch in comparison with other similar frameworks.
- UDAParts performance studies have shown that SocketPro delivers the best performance over other high quality communication frameworks.
- Client socket pool and server routing/load balancing are two tools for better scalability.
- SocketPro server HTTP/websocket service extends SocketPro accessibility to all devices/machines and development languages.

11.7 Thread safety for all core libraries and adapters

- Both SocketPro client and server core libraries are virtually thread-safe.
- SocketPro client and server core libraries manage threads automatically for you.
- SocketPro adapters are thread-safe.
- There are very few chances for you to meet thread dead locks if you use SocketPro for development.

11.8 SocketPro persistent message queue

- SocketPro message queues support message enqueuing and dequeuing in transaction style.
- SocketPro client queue improve application stability as it is designed to have better system fault tolerance over a set of common issues.
- One SocketPro server persistent queue is able to support multiple providers and consumers simultaneously at the same time.
- SocketPro server persistent queues support message enqueuing notification so that consumers are able to dequeue messages in real time fashion.

11.9 Three free but professional and highly reusable server plug-ins

- SocketPro asynchronous persistent message queue plug-in.
- SocketPro asynchronous SQLite database plug-in.
- SocketPro asynchronous MySql/Mariadb embedded database/broker plug-in.

11.10 Other features

- SocketPro core libraries have two compression tools, fast compression and best compression rate, to reduce request and result data sizes for your conveniences.
- SocketPro comes with real-time update caches implemented on SocketPro online message bus.
- A SocketPro message queue can be replicated onto different queues consistently in transaction style.
- Fast exchanging files having any sizes between client and server side.

11.11 Diversity of samples and development simplicity

- Compatible, cross-platform and cross-language tutorial samples.
- SocketPro server plug-in development samples.
- Performance study examples.
- Database real-time cache examples.
- Development is simple.

12. Limitations

1. Continuously sending lots of requests may have dead-lock at client side. This dead-lock may happen only if all the following situations are met at the same time:
    - Continuously sending **lots of requests** asynchronously.
    - A socket is **not** associated with persistent message queue at client side.
    - A mutex or critical section is used **across calling SendRequest** within your request.
    - The **same** mutex or critical section is also used within **result callback**.

    To avoid the potential dead lock, you may consult source code for solution at one of the files, CAsyncDBHandler.java, asyncdbhandler.cs, asyncdbhandler.py and udb_client.cpp.