**Brief introduction of SocketPro high performance and scalable persistent message queue**

**Introduction**

Persistent message queue allows applications running on separate machines/processes to communicate in a failsafe manner. A message queue is a temporary storage location or file from which messages can be saved and read reliably, as and when conditions permit. Unlike sockets and other common channels that require direct connections always exist, persistent message queue enables communication among applications which may not always be connected. There are many persistent message queues implemented in own ways. SocketPro comes with an extremely high performance persistent message queue for you to freely reuse.

Both SocketPro client and server core libraries are internally implemented with persistent message queue. Its client queue is used to back up requests so that all requests can be resent to a server for processing in case the server is not accessible for whatever reasons such as server power-off, server application down, network off and so on. Essentially, client queue is used as a tool for fault auto recovery to increase application stability and reduction of development complexity as shown at the directory of socketPro/samples/auto\_recovery/(test\_cplusplus|test\_java|test\_python|test\_sharp).

This article is focused on SocketPro server side persistent message queue. It is noted that precompiled SocketPro server side library of persistent message queue is **completely free** to you with open source codes which are extremely simple and understandable. You can also rely on the open source codes to extend them for your complex needs.

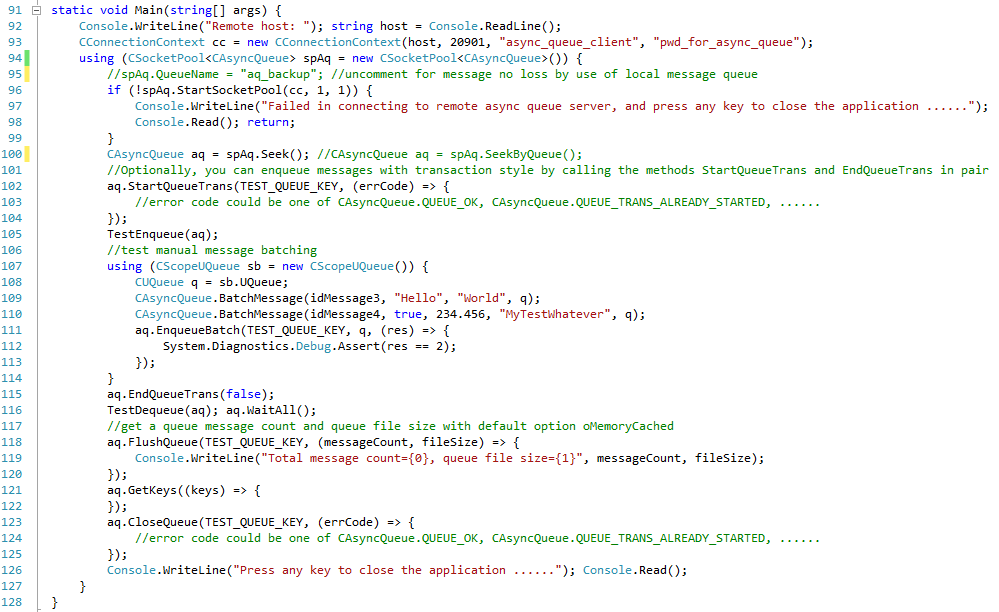
**Source codes and samples**

All related source codes and samples are located at https://github.com/udaparts/socketpro. After cloning it into your computer by GIT, pay attention to the subdirectory uasyncqueue inside the directory socketpro/samples/module\_sample. You can see these samples are created from .NET, C/C++, Java and Python development environments. They can be compiled and run on either Linux or window platforms. SocketPro comes with a pre-compiled system library uasyncqueue, which is located at directories socketpro/bin/win and socketpro/bin/linux for both windows and linux variants, respectively. In addition, you can figure out how to load the SocketPro queue service into a server application with your familiar development environment by looking at tutorial sample all\_servers at the directory socketpro/tutorials/(cplusplus|csharp|vbnet|java/src)/all\_servers. However, we only use C# client code (socketpro/samples/module\_sample/uasyncqueue /test\_csahrp) at this article for explanations.

You should distribute these system libraries inside the directory socketpro/bin into your system directory before running these sample applications. In regards to SocketPro communication framework, you may also refer to its development guide documentation at socketpro/doc/SocketPro development guide.pdf.

**Main function**

SocketPro is written from bottom to support parallel computation by use of one or more pools of non-blocking sockets. Each of pools may be made of one or more threads, and each of threads hosts one or more non-blocking sockets at client side. To increase scalability, you can create one or more pools having multiple non-block sockets that are connected to different queue servers so that you can send messages for queuing in parallel style. However, we just use one pool for demonstration clarity here. Further, the pool is only made of one thread and one socket for this sample at client side as shown in the below Figure 1.



*Figure 1: Main function for demonstration of use of SocketPro persistent message queue at client side*

**Starting one socket pool:** The above Figure 1 starts one socket pool which only has one worker thread that only hosts one non-blocking socket at line 96 for demonstration clarity by use of one instance of connection context. It is noted that you can create multiple pools within one client application if necessary. Afterwards, we get one asynchronous CAsyncQueue handler at line 100.

**Streaming message:** We can send individual messages onto a server for saving in stream style without batching at client side at line 105. We are going to talk with details in a new section ***TestEnqueue***.

**Manual message batching:** When there are many small messages to be sent for saving, these small messages will require very much CPU costs at both client and server sides because of thread synchronization, function processing, SocketPro internal inline batching as well as others. To reduce these costs, we can batch these small messages into one bigger chunk, and send them as one larger unit to server for saving as shown in lines 107 through 114. This is a way to improve message en-queue performance, but it also increases latency because it requires a time interval, which is usually more than 1 millisecond, for collecting an enough number of small messages before manual batching. Also, it requires more codes. It is **NOT** recommended with SocketPro as long as either performance of streaming message queues meets your needs or message sizes are not very small.

**Saving message in transaction style:** SocketPro persistent message queue supports saving messages in transaction style. To use this feature, you have to call the methods *StartQueueTrans* and *EndQueueTrans* in pair as shown at lines 102 and 115, respectively. It is noted that total size of batched messages shouldn’t be over four gig bytes.

**Reading messages in a queue file from multiple consumers:** Certainly, you can read messages from a queue as shown at line 116. We’ll elaborate it more at the section ***TestDequeue*** in detail. It is noted that one SocketPro queue supports message writing from multiple providers and message reading from multiple consumers simultaneously at the same time. Just for your information, it is common that other queue implementations don’t support multiple consumers on one queue file.

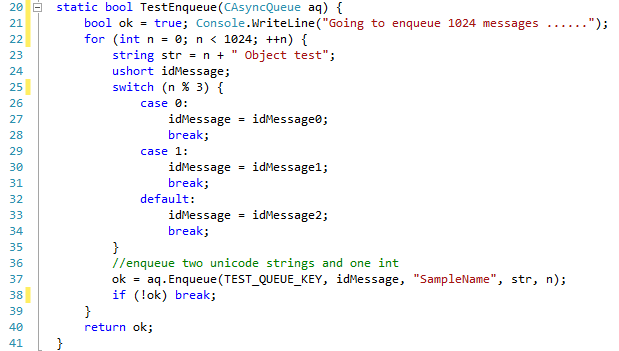
**Scalability:** A client is able to create a pool that has multiple sockets connected to different server queue machines. A client is able to use the pool method *Seek* or *SeekByQueue*, and distribute messages onto different servers for saving. Don’t be fooled by this sample code because the demonstration is designed for clarity and beginner.

**No message loss:** Message saving requires transferring messages from client or provider to a message queue server. The server and network may be down for many possible reasons. Therefore, messages could be lost without your care by extra coding. You can prevent it with SocketPro easily by use of client or local message queue for backing up these messages before putting them on wire at line 95. In case a server or network is down, SocketPro can resend messages that are backed up in a local or client message queue file when a queue server application is re-accessible.

**Other functionalities:** SocketPro persistent message queue provides other methods to check the count of messages, the size of a queue file and keys to different message queues as well as closing a queue at lines 118, 121 and 123, respectively.

**TestEnqueue**

The function, an example for en-queuing messages, is simple as shown at the below Figure 2.

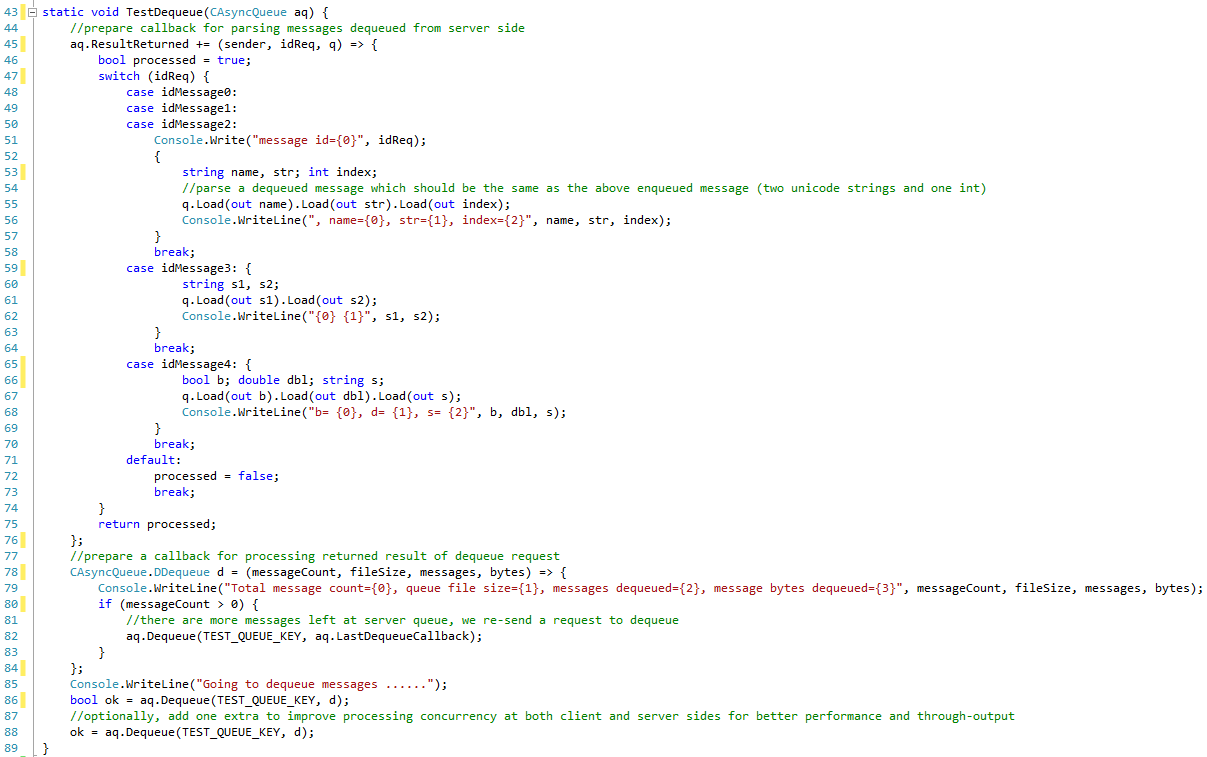


*Figure 2: Sample code for sending 1024 message queues to a server for saving*

As shown at line 37 in Figure 2, we can continuously send individual messages in streaming style. You can see that it is really easy to en-queue messages with SocketPro.

**TestDequeue**

The below Figure 3 is a demonstration for de-queuing messages in batch.



*Figure 3: Sample code snippet for de-queuing messages in batch*

The callback of lines 45 through 76 in Figure 3 is used to parse messages that come from remote message queue file. The codes of line 48 through 57 are used to parse messages originated from the previous Figure 2. The codes of lines 59 through 70 are used to parse manual batched messages that originated from lines 109 and 110 in Figure 1. As hinted by comment at line 77 in Figure 3, the callback of lines 78 through 64 is used to monitor key message queue data like message count (messages to be de-queued), server queue file size, messages transferred by the below call *Dequeue*, and message size in bytes. Inside the callback, it is necessary to call the method *Dequeue* recursively if there is a message remaining in a server queue file as shown at lines 80 through 83.

After preparing the previous two callbacks, we finally call the method *Dequeue* for sending a request to server for reading messages in batch. Optionally, we can call the method *Dequeue* one or two times more so that it can increase de-queuing throughput or performance because client side message parsing and server message reading can have better concurrency in processing.

**Performance study**

SocketPro is written from beginning to support streaming requests by use of non-block sockets and inner algorithms for the best network and code efficiency. You can refer to the short article of the file [socketpro/doc/sq\_kafka\_perf.pdf](sq_kafka_perf.pdf) for performance study result. The performance study samples are located at the directory socketpro/samples/qperf.

Our results show that SocketPro queue is significantly faster than Kafka, especially when writing high volume of small messages.

**Highlights of SocketPro persistent message queue**

Recently, Kafka queue is the most popular for its performance and scalability. Therefore, it is worth to compare SocketPro persistent message queue with Kafka, which may highlight SocketPro persistent message queue advantages.

1. SocketPro persistent message queue **has no complex configuration settings** for you to understand and configure. Contrarily, Kafka requires you to understand many configuration settings ahead.
2. SocketPro persistent message queue supports **manual transaction** for better stability, but Kafka doesn’t.
3. A queue file can be **sharable among multiple consumers** at the same time with SocketPro queue, but Kafka is not capable to do so.
4. SocketPro queue can guarantee **no message loss** as long as you turn on local or client message queue, but you cannot do so with Kafka.
5. SocketPro queue supports message availability to notify all connected consumers in real-time fashion for the shortest latency. Its latency is **always** equal to 1.5 times of network latency and could be as low as 0.3 ms on local area network. Kafka’s lowest latency is 1 ms at best after you must configure a setting specifically for it.
6. SocketPro queue is significantly **faster** than Kafka especially in high volume of small message writing.
7. You can **selectively** en-queue a portion of messages at your will with SocketPro, but you are forced to en-queue all messages with Kafka. Further, you can integrate message queue with SocketPro other features such as online message bus, local message queue, client server communication, and so on.
8. Both client and server codes of SocketPro persistent message queue are extremely **simple**, you can easily extend and modify them for your complex needs. It is not so easy for you to do so with Kafka.
9. You can embed SocketPro queue within your application system with **much simpler distribution and low dependency**. It is not so easy for you to do so with Kafka.