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). In regards to Node.js, the sample code is at the file ../socketpro/src/njadapter/server_queue.js.

This article is focused on SocketPro persistent message queue publish and consume developments. It is noted that precompiled SocketPro server 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/components. 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 platforms, 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, as usual we only use C# client code (socketpro/components/uasyncqueue /test_csahrp) at this article for explanations.

You should distribute 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.

```
91  static void Main(string[] args) {
92      Console.WriteLine("Remote host: "); string host = Console.ReadLine();
93      CConnectionContext cc = new CConnectionContext(host, 20901, "async_queue_client", "pwd_for_async_queue");
94      using (CSocketPool<CAsyncQueue> spAq = new CSocketPool<CAsyncQueue>()) {
95          //spAq.QueueName = "aq_backup"; //uncomment for message no loss by use of local message queue
96          if (!spAq.StartSocketPool(cc, 1, 1)) {
97              Console.WriteLine("Failed in connecting to remote async queue server, and press any key to close the application ......");
98              Console.Read(); return;
99          }
100         CAsyncQueue aq = spAq.Seek(); //CAsyncQueue aq = spAq.SeekByQueue();
101         //Optionally, you can enqueue messages with transaction style by calling the methods StartQueueTrans and EndQueueTrans in pair
102         aq.StartQueueTrans(TEST_QUEUE_KEY, (errCode) => {
103             //error code could be one of CAsyncQueue.QUEUE_OK, CAsyncQueue.QUEUE_TRANS_ALREADY_STARTED, ......
104         });
105         TestEnqueue(aq);
106         //test manual message batching
107         using (CScopeUQueue sb = new CScopeUQueue()) {
108             CUQueue q = sb.UQueue;
109             CAsyncQueue.BatchMessage(idMessage3, "Hello", "World", q);
110             CAsyncQueue.BatchMessage(idMessage4, true, 234.456, "MyTestWhatever", q);
111             aq.EnqueueBatch(TEST_QUEUE_KEY, q, (res) => {
112                 System.Diagnostics.Debug.Assert(res == 2);
113             });
114         }
115         aq.EndQueueTrans(false);
116         TestDequeue(aq); aq.WaitAll();
117         //get a queue message count and queue file size with default option oMemoryCached
118         aq.FlushQueue(TEST_QUEUE_KEY, (messageCount, fileSize) => {
119             Console.WriteLine("Total message count={0}, queue file size={1}", messageCount, fileSize);
120         });
121         aq.GetKeys((keys) => {
122         });
123         aq.CloseQueue(TEST_QUEUE_KEY, (errCode) => {
124             //error code could be one of CAsyncQueue.QUEUE_OK, CAsyncQueue.QUEUE_TRANS_ALREADY_STARTED, ......
125         });
126         Console.WriteLine("Press any key to close the application ......"); Console.Read();
127     }
128 }
```

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.

```
20  static bool TestEnqueue(CAsyncQueue aq) {
21      bool ok = true; Console.WriteLine("Going to enqueue 1024 messages ......");
22      for (int n = 0; n < 1024; ++n) {
23          string str = n + " Object test";
24          ushort idMessage;
25          switch (n % 3) {
26              case 0:
27                  idMessage = idMessage0;
28                  break;
29              case 1:
30                  idMessage = idMessage1;
31                  break;
32              default:
33                  idMessage = idMessage2;
34                  break;
35          }
36          //enqueue two unicode strings and one int
37          ok = aq.Enqueue(TEST_QUEUE_KEY, idMessage, "SampleName", str, n);
38          if (!ok) break;
39      }
40      return ok;
41  }
```

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.

```
43  static void TestDequeue(CAsyncQueue aq) {
44      //prepare callback for parsing messages dequeued from server side
45      aq.ResultReturned += (sender, idReq, q) => {
46          bool processed = true;
47          switch (idReq) {
48              case idMessage0:
49              case idMessage1:
50              case idMessage2:
51                  Console.Write("message id={0}", idReq);
52                  {
53                      string name, str; int index;
54                      //parse a dequeued message which should be the same as the above enqueued message (two unicode strings and one int)
55                      q.Load(out name).Load(out str).Load(out index);
56                      Console.WriteLine(", name={0}, str={1}, index={2}", name, str, index);
57                  }
58                  break;
59              case idMessage3: {
60                      string s1, s2;
61                      q.Load(out s1).Load(out s2);
62                      Console.WriteLine("{0} {1}", s1, s2);
63                  }
64                  break;
65              case idMessage4: {
66                      bool b; double dbl; string s;
67                      q.Load(out b).Load(out dbl).Load(out s);
68                      Console.WriteLine("b= {0}, d= {1}, s= {2}", b, dbl, s);
69                  }
70                  break;
71              default:
72                  processed = false;
73                  break;
74          }
75          return processed;
76      };
77      //prepare a callback for processing returned result of dequeue request
78      CAsyncQueue.DDequeue d = (messageCount, fileSize, messages, bytes) => {
79          Console.WriteLine("Total message count={0}, queue file size={1}, messages dequeued={2}, message bytes dequeued={3}", messageCount, fileSize, messages, bytes);
80          if (messageCount > 0) {
81              //there are more messages left at server queue, we re-send a request to dequeue
82              aq.Dequeue(TEST_QUEUE_KEY, aq.LastDequeueCallback);
83          }
84      };
85      Console.WriteLine("Going to dequeue messages ......");
86      bool ok = aq.Dequeue(TEST_QUEUE_KEY, d);
87      //optionally, add one extra to improve processing concurrency at both client and server sides for better performance and through-output
88      ok = aq.Dequeue(TEST_QUEUE_KEY, d);
89  }
```

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. The performance study samples, which are written from C++, Java and C#, are located at the directory socketpro/samples/qperf. In addition, we also compared SocketPro queue with the two popular queues Kafka and RabbitMQ, as shown at the two short articles, *perf_comparison.pdf* and *sq_kafka_perf.pdf*.

Our results show that SocketPro queue is significantly faster than not only RabbitMQ but also Kafka, especially when writing high volume of small messages. For clarity, this short article focuses on performance comparison of SocketPro queue with Kafka only.

It is the most important to compare SocketPro queue with Kafka on local area network **(LAN)** environment as this scenario is much closer to real queue applications under most of cases. Our test results are listed in the below Figure 4. It is noted that Kafka performance test script *kafka_perf_test.txt* is located at the directory *../socketpro/samples/qperf*. SocketPro queue supports message enqueuing in real-time streaming style by continuously sending messages by its nature. Further, SocketPro also supports sending all batched messages as one larger message though its manual batching feature, which is designed to sacrifice latency for better enqueuing speed or throughput. Finally, it is noted that Kafka enqueuing performance tests are always completed in batch style by setting configuration property batch.size.

	A	B	C	D	E
1	***Message count***	***Message size***		***Enqueuing speed***	***Dequeuing speed***
2		***(bytes)***		***(messages/second)***	***(messages/second)***
3	200,000,000	4	Kafka	1,083,00	1,406,000
4			***SocketPro + Batch***	***5,988,000***	1,195,000
5			SocketPro	839,300	
6	200,000,000	32	Kafka	819,300	1,043,000
7			***SocketPro + Batch***	***2,312,000***	974,100
8			SocketPro	752,300	
9	50,000,000	200	Kafka	238,900	377,200
10			***SocketPro + Batch***	***457,500***	392,400
11			SocketPro	352,300	
12	10,000,000	1024	Kafka	56,900	91,210
13			SocketPro	86,010	88,500
14	1,000,000	10240	Kafka	6,405	9,818
15			SocketPro	9,040	9,420
16	***Queue Server Machine: Ubuntu 16.04, Intel Core i7-4770 CPU 3.40 GHz, 16 GB RAM***				
17	***SocketPro version: 6.0.3.4; Kafka version: 2.11-0.10.1.1; Switch: 1 GBPS***				
18	***Provider/Consumer: Win10 Professional, Intel Core™ i7-5500U CPU @ 2.40 GHz, 8 GB RAM***				

Figure 4: Queue performance comparison between SocketPro and Kafka on LAN

Small messages: In regards to small messages (4 bytes & 32 bytes), Kafka is slightly faster (< 25%) than SocketPro for both enqueuing ([1,083,000 & 819,300]/[839,300 & 752,300]) and dequeuing ([1,406,000 & 1,043,000]/[1,195,000 & 974,100]) if SocketPro manual batching feature is **NOT** employed. SocketPro queue is slower in enqueuing because transferring small messages cross wire is very expensive in CPU without manual batching. However, SocketPro queue, if it is armed with manual batch like Kafka does, SocketPro queue could be significantly faster (5,988,000/1,083,00 = 5.53 or 450%) than Kafka.

It is found that Kafka dequeuing is about 15% faster than SocketPro queue for small messages as shown in the Figure 4. Its explanation is that SocketPro dequeuing always sends a dequeue acknowledgement from consumer to server for each of dequeued messages automatically. Further, dequeue acknowledgement also causes disk seeking for marking all dequeued messages at server side. This happens silently for reducing consumer side coding complexity, but it obviously degrades SocketPro dequeuing performance somewhat. Kafka is very simple in dequeuing message and has no support with the similar dequeue acknowledgement at all.

Middle messages: Considering middle size of messages (200 bytes), SocketPro is considerably faster than Kafka in enqueuing message even if SocketPro doesn't use manual batch. If SocketPro is armed with manual batching like Kafka does, SocketPro could be 90% (457,500/238,900 = 1.91 = 90%) faster than Kafka. However, both SocketPro and Kafka show similar performance in dequeuing middle size of messages.

Large messages: SocketPro is about 40% faster than Kafka in enqueuing large size of messages (1024 bytes & 10240 bytes). SocketPro and Kafka don't show performance difference in dequeuing middle size of messages.

At last, it is pointed that SocketPro manual batching is **not** recommended as long as message enqueuing speed meets your needs. The reason is that manual batch could significantly increase message enqueuing latency because you must have a time interval for collecting enough messages before putting messages onto wire in real applications. SocketPro manual batching should be used only if your application requires better enqueuing speed for high volume of small messages.

It is time to talk about wide area network (**WAN**). WAN challenges distributed application developments because it has not only low bandwidth but also significantly large latency. We like to compare SocketPro with Kafka on WAN for remoting message enqueuing and dequeuing. It is found that our SocketPro runs very well for both performance and stability as shown in the below Figure 5. Unfortunately, we cannot finish a Kafka performance testing on WAN, and eventually give it up after trying a few days and searching many web sites for its possible configuration settings. It seems to us that Kafka doesn't support remote message enqueuing or dequeuing at all.

	A	B	C	D	E
1	***Message count***	***Message size***	***Testing***	***Enqueuing speed***	***Dequeuing speed***
2		***(bytes)***	***method***	***(messages/second)***	***(messages/second)***
3	10,000,000	4	No batching	136,000	97,600
4			**Batching**	**336,000**	
5	10,000,000	32	No batching	62,900	51,400
6			**Batching**	**88,700**	
7	2,500,000	200	No batching	15,100	13,600
8			**Batching**	**16,300**	
9	500,000	1024	No batching	3,210	2,760
10	50,000	10240	No batching	327	288
11	***us-central1-c; Queue server: Ubuntu 16.04, 2 vCPUS, 7.5 GB RAM, 2.3 GHz Intel Xeon E5 v3***				
12	***us-west1-a; Provider/Consumer: Ubuntu 16.04, 2vCPUs, 7.5 GB RAM, 2.2 GHz Intel Xeon E5 v4***				
13	***SocketPro version 6.1.0.3, bandwidth=40 Mbps, latency=35 ms***				

Figure 5: SocketPro persistent queue performance results on two cheap virtual machines across Google cloud data centers

The above Figure 5 shows SocketPro queue results measured from two cheap virtual machines across Google cloud data centers. The network has a bandwidth around 40 Mbps with a high latency about 35 milliseconds. Both client and server application codes (sq_client and sq_server) could be found at the directory ../socketpro/samples/qperf/cperf. After cloning SocketPro at the site https://github.com/udaparts/socketpro by GIT and checking out the branch *linux_tests*, you will find the two pre-compiled applications at the directory ../socketpro/test_apps. You can use the two applications for your own testing.

The above test results show that the network bandwidth is fully consumed no matter enqueuing or dequeuing messages. WAN bandwidth determines speeds of both message enqueuing and dequeuing. The above Figure 5 also shows that both enqueuing and dequeuing speeds could be easily over 10,000 messages per second if message size is not more than 300 bytes. It is noted that the performance data could be doubled if you can turn on SocketPro in-line compression and decompression features.

If message sizes are smaller than 64 bytes, you can also use SocketPro manual batching feature to increase performance further.

At last, it is pointed that no other message queue could be found publicly to get similar WAN performance results as far as we know. If you find one, please let us know so that we can change this claim quickly.

Highlights of SocketPro persistent message queue

At end, it is worth highlighting SocketPro persistent message queue advantages over Kafka.

1. SocketPro persistent message queue **has no complex configuration settings** for you to understand and configure. Contrarily, Kafka requires you must understand many configuration settings ahead.
2. SocketPro persistent message queue runs on **WAN** very well with decent performance and stability, but Kafka doesn't.
3. SocketPro persistent message queue supports **manual transaction** for better stability, but Kafka doesn't.
4. A queue file can be **sharable among multiple consumers** at the same time with SocketPro queue, but Kafka is not capable to do so.
5. 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.
6. 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.
7. SocketPro queue is significantly **faster** than Kafka especially in high volume of small message writing when turning on manual batching.
8. You can **selectively** en-queue a portion of messages 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.
9. 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.

10. 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.
11. Like Kafka, SocketPro queue is extremely **scalable** too.