Tutoral 3 -- SocketPro server persistent queue

Contents:

1. Introduction

Requests can be persistently saved into a disk data file. If an application is crashed or its network is down, requests can be restored from the disk file after the application is restored or the network is online. As you can see, persistent message queue ensures the reliability of the whole application system.

SocketPro fully supports persistent message queues at both client and server sides. In regards to client persistent message queue, you can review it with tutorial one. This tutorial is focused on server persistent message queue only.

The tutorial projects are located at the directory ../tutorials/(csharp|vbnet\cplusplus|java\src|ce|python)/server_queue

2. Server side code

Brief introduction about SocketPro asynchronous persistent message queue plug-in

In regards to server persistent message queue, it can be used as asynchronous communication bridge among clients offline at different time through a server or broker in middle. After a client publishes a message to the server and saves (en-queues) it into a queue, another client can load (de-queue) the message at its own pace. There are a number of persistent queues available on the market, and some of them are widely used by many application systems. UDAParts has created a SocketPro server plug-in for its own persistent queue, which can be reused with its own development environment with inline en-queue/de-queue batching algorithm for the best performance and scalability as shown in the below Figure 1.

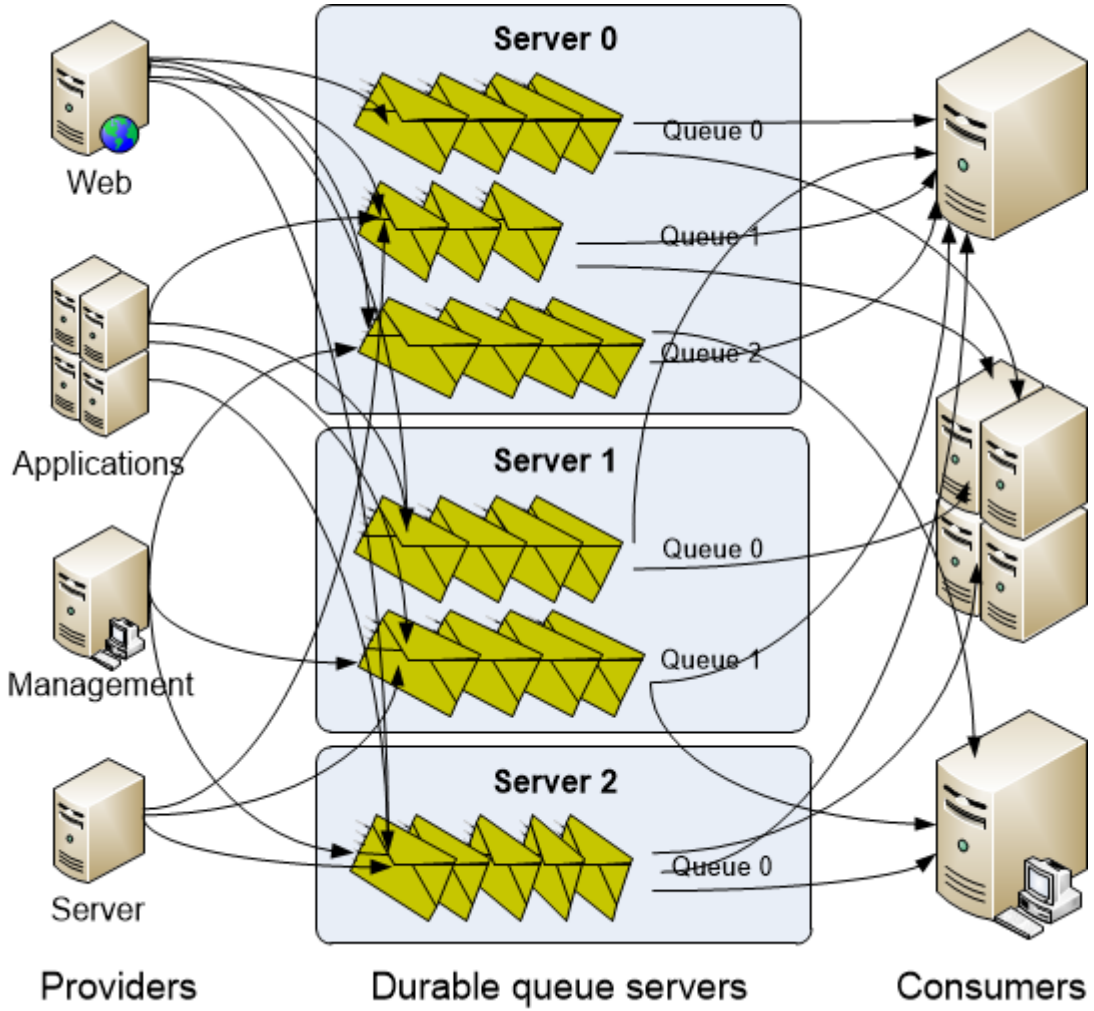

Figure 1: A server cluster with three SocketPro servers supporting multiple providers and consumers

Figure 1 illustrates a server cluster with three SocketPro servers for variable numbers of persistent message queues. It also shows that one specific queue is able to support both en-queuing and de-queuing messages from multiple providers and consumers simultaneously at the same time. You can also open multiple queues from one SocketPro server application.

Load persistent queue plug-in at server side: It is simple to load persistent message queue plug-in at server side within your development environment, as shown in the following Figure 2.

```
2  using System;
3  using SocketProAdapter.ServerSide;
4
5  public class CMySocketProServer : CSocketProServer {
6      static void Main(string[] args) {
7          if (System.Environment.OSVersion.Platform == PlatformID.Unix)
8              CSocketProServer.QueueManager.WorkDirectory = "/home/yye/sp_test/";
9          else
10             CSocketProServer.QueueManager.WorkDirectory = "c:\\sp_test";
11         using (CMySocketProServer MySocketProServer = new CMySocketProServer()) {
12             // load SocketPro async persistent plug-in library
13             // uasyncqueue.dll (windows) or libuasyncqueue.so (Linux)
14             IntPtr p = CSocketProServer.DllManager.AddALibrary("uasyncqueue", 16 * 1024);
15             // 16 * 1024 == dequeue batch size in bytes
16             if (p.ToInt64() == 0)
17                 Console.WriteLine("Cannot load SocketPro async queue library");
18             else if (!MySocketProServer.Run(20901))
19                 Console.WriteLine("Error code = " + CSocketProServer.LastSocketError.ToString());
20             Console.WriteLine("Input a line to close the application ......");
21             Console.ReadLine();
22         }
23     }
24 }
```

Figure 2: Load SocketPro async message queue and start a SocketPro server

The lines 7 through 10 set a working directory where we are going to save various message queue files. The line 14 loads SocketPro plug-in for asynchronous server side persistent message queue. You can find all the plug-in source code and different test projects at the directory ../socketpro/samples/module_sample/uasyncqueue.

3. Client side code

Send a number of requests: As shown in the following Figure 3, we send 1024 messages onto a remote SocketPro server for en-queuing at line 26 from a given queue key (or queue name).

```
 9      const ushort idMessage0 = (ushort)tagBaseRequestID.idReservedTwo + 100;
10      const ushort idMessage1 = (ushort)tagBaseRequestID.idReservedTwo + 101;
11      const ushort idMessage2 = (ushort)tagBaseRequestID.idReservedTwo + 102;
12      const string TEST_QUEUE_KEY = "queue_name_0";
13      static bool TestEnqueue(CAsyncQueue aq) {
14          bool ok = true;
15          Console.WriteLine("Going to enqueue 1024 messages ......");
16          for (int n = 0; n < 1024; ++n) {
17              string str = n + " Object test";
18              ushort idMessage;
19              switch (n % 3) {
20                  case 0: idMessage = idMessage0;
21                      break;
22                  case 1: idMessage = idMessage1; break;
23                  default: idMessage = idMessage2; break;
24              }
25              //enqueue two unicode strings and one int
26              ok = aq.Enqueue(TEST_QUEUE_KEY, idMessage, "SampleName", str, n);
27              if (!ok)
28                  break;
29          }
30          return ok;
31      }
```

Figure 3: Send 1024 messages onto remote server for en-queuing with one queue key or name

Each of messages contains two Unicode strings and one integer. It is noted that each message has one identification number (idMessage) at line 26. When a consumer gets a message from a server queue with an id, it will parse its source message according to the message id.

Dequeue messages in batch: It is also simple to de-queue messages from server queue in batch as shown in Figure 4 below.

```
33      static void TestDequeue(CAsyncQueue aq) {
34          //prepare callback for parsing messages dequeued from server side
35          aq.ResultReturned += (sender, idReq, q) => {
36              bool processed = true;
37              switch (idReq) {
38                  case idMessage0: case idMessage1: case idMessage2:
39                      Console.Write("message id={0}", idReq); {
40                          string name, str; int index;
41                          //parse a dequeued message (two unicode strings and one int)
42                          q.Load(out name).Load(out str).Load(out index);
43                          Console.WriteLine(", name={0}, str={1}, index={2}", name, str, index);
44                      }
45                      break;
46                  default: processed = false; break;
47              }
48              return processed;
49          };
50          //prepare a callback for processing returned result of dequeue request
51          CAsyncQueue.DDequeue d = (messageCount, fileSize, messages, bytes) => {
52              if (messageCount > 0)
53                  //there are more messages left at server queue, we re-send a request to dequeue
54                  aq.Dequeue(TEST_QUEUE_KEY, aq.LastDequeueCallback);
55          };
56          Console.WriteLine("Going to dequeue messages ......");
57          bool ok = aq.Dequeue(TEST_QUEUE_KEY, d);
58          // optionally, add one or two extra calls to improve processing concurrency
59          // at both client and server sides for better performance and through-output
60          ok = aq.Dequeue(TEST_QUEUE_KEY, d);
61      }
```

Figure 4: De-queue messages at client side from a server queue in batch

Lines 35 through 49 is actually a callback for parsing messages de-queued from a server queue. This sample message actually consists of two Unicode strings plus one integer as described at the previous section. The code of lines 51 to 55 is a callback for the method Dequeue. Inside the callback we may recall the method Dequeue as long there is a message remaining in a server queue.

At last, we may call the method Dequeue multiple times to improve concurrency at both client and server sides for better throughput.