Sharing a Socket Pool of Asynchronous Handlers among Multiple Threads

Introduction

SocketPro is next generation technology that streams both requests and responses. It is written from scratch with continuous in-line request/result batching, asynchronous data transferring and parallel computation in mind. It prefers streaming all requests by use of one single socket connection at client side from different threads for the best network efficiency through SocketPro inner batching algorithm. Therefore, you, a software developer or architect, must know how to share one single socket connection or its associated asynchronous handler among multiple threads within a multithreaded environment.

This short article is focused on sharing a pool of asynchronous handlers among multiple threads for client side development.

Source Codes and Samples

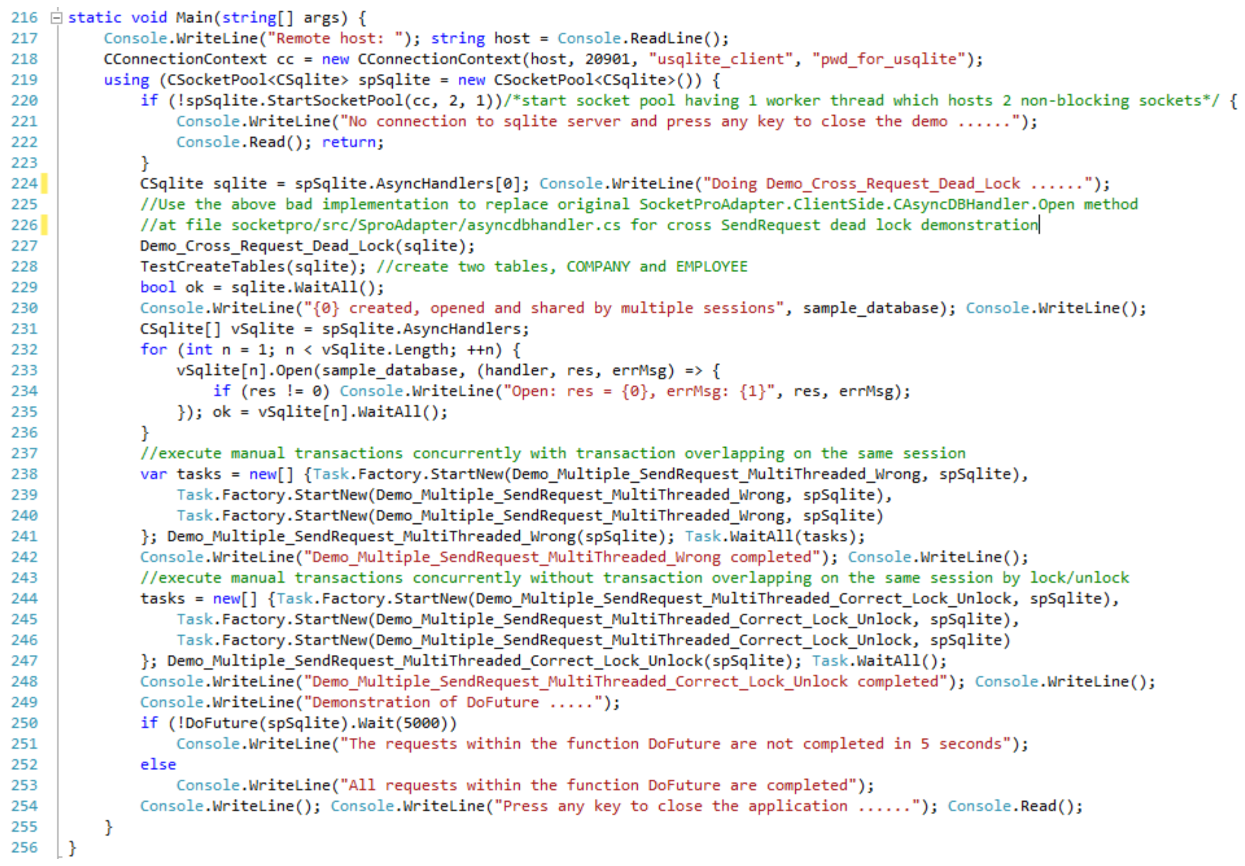
All source codes and samples are located at https://github.com/udaparts/socketpro. After cloning it into your computer by GIT and having a quick look at the subdirectory socketpro/samples/fatclient_thread, you will find that there are four subdirectories, cplusplus, dotnet, java_demo and python_demo for C++, C#, Java and Python development languages, respectively. It is noted that all of them are focused on client side development. However, this article only uses C# code example for explanations as usual.

To running one of these client side applications, we need a sample SQLite server application test_ssqlite inside a subdirectory win or Linux of socketpro/bin. The sample server, which is described at the article socketpro/doc/sqlstream_sqlite.pdf, is very simple to run without any required configuration for you to understand ahead. SocketPro many demonstration samples are using the server for demonstration purpose.

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

First, let’s see the below Figure 1 for main function.



```
216 static void Main(string[] args) {
217     Console.WriteLine("Remote host: "); string host = Console.ReadLine();
218     CConnectionContext cc = new CConnectionContext(host, 20901, "usqlite_client", "pwd_for_usqlite");
219     using (CSocketPool<CSqlite> spSqlite = new CSocketPool<CSqlite>()) {
220         if (!spSqlite.StartSocketPool(cc, 2, 1))/*start socket pool having 1 worker thread which hosts 2 non-blocking sockets*/ {
221             Console.WriteLine("No connection to sqlite server and press any key to close the demo ......");
222             Console.Read(); return;
223         }
224         CSqlite sqlite = spSqlite.AsyncHandlers[0]; Console.WriteLine("Doing Demo_Cross_Request_Dead_Lock ......");
225         //Use the above bad implementation to replace original SocketProAdapter.ClientSide.CAsyncDBHandler.Open method
226         //at file socketpro/src/SproAdapter/asyncdbhandler.cs for cross SendRequest dead lock demonstration
227         Demo_Cross_Request_Dead_Lock(sqlite);
228         TestCreateTables(sqlite); //create two tables, COMPANY and EMPLOYEE
229         bool ok = sqlite.WaitAll();
230         Console.WriteLine("{0} created, opened and shared by multiple sessions", sample_database); Console.WriteLine();
231         CSqlite[] vSqlite = spSqlite.AsyncHandlers;
232         for (int n = 1; n < vSqlite.Length; ++n) {
233             vSqlite[n].Open(sample_database, (handler, res, errMsg) => {
234                 if (res != 0) Console.WriteLine("Open: res = {0}, errMsg: {1}", res, errMsg);
235             }); ok = vSqlite[n].WaitAll();
236         }
237         //execute manual transactions concurrently with transaction overlapping on the same session
238         var tasks = new[] {Task.Factory.StartNew(Demo_Multiple_SendRequest_MultiThreaded_Wrong, spSqlite),
239             Task.Factory.StartNew(Demo_Multiple_SendRequest_MultiThreaded_Wrong, spSqlite),
240             Task.Factory.StartNew(Demo_Multiple_SendRequest_MultiThreaded_Wrong, spSqlite)
241         }; Demo_Multiple_SendRequest_MultiThreaded_Wrong(spSqlite); Task.WaitAll(tasks);
242         Console.WriteLine("Demo_Multiple_SendRequest_MultiThreaded_Wrong completed"); Console.WriteLine();
243         //execute manual transactions concurrently without transaction overlapping on the same session by lock/unlock
244         tasks = new[] {Task.Factory.StartNew(Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock, spSqlite),
245             Task.Factory.StartNew(Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock, spSqlite),
246             Task.Factory.StartNew(Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock, spSqlite)
247         }; Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock(spSqlite); Task.WaitAll();
248         Console.WriteLine("Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock completed"); Console.WriteLine();
249         Console.WriteLine("Demonstration of DoFuture .....");
250         if (!DoFuture(spSqlite).Wait(5000))
251             Console.WriteLine("The requests within the function DoFuture are not completed in 5 seconds");
252         else
253             Console.WriteLine("All requests within the function DoFuture are completed");
254         Console.WriteLine(); Console.WriteLine("Press any key to close the application ......"); Console.Read();
255     }
256 }
```

Figure 1: Main function for parallel processing within multi-threaded environment

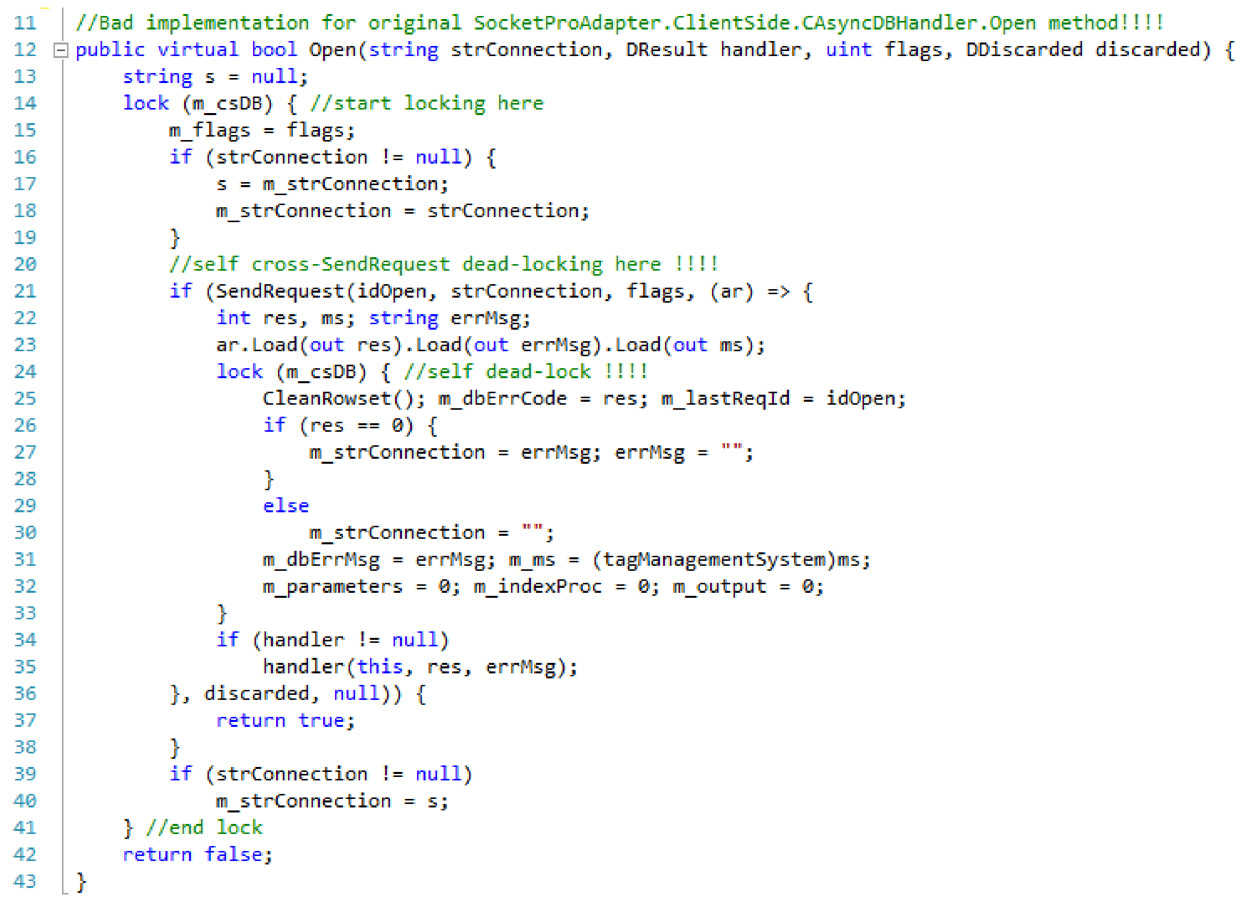
We use the main function in the above Figure 1 to demonstrate the following topics:

- Cross request dead lock as shown at line 227
- Sharing one single connection by multiple threads concurrently at lines 238 through 241
- Locking one single connection from a pool of sockets by one of threads at lines 244 through 247
- Waiting one result for one specific request at line 250

This short article is designed to focus on the above four topics.

Cross request dead lock

The first issue is dead lock which you must keep away from, when you create an own asynchronous handler. This is specific issue to SocketPro request streaming technology, which you will not see with traditional non-streaming client/server communication frameworks. To help you recognize the dead lock, see the below Figure 2 which is a bad implementation for original method *Open* of the class *CAsyncDBHandler*.



```
11  //Bad implementation for original SocketProAdapter.ClientSide.CAsyncDBHandler.Open method!!!!
12  public virtual bool Open(string strConnection, DResult handler, uint flags, DDiscarded discarded) {
13      string s = null;
14      lock (m_csDB) { //start locking here
15          m_flags = flags;
16          if (strConnection != null) {
17              s = m_strConnection;
18              m_strConnection = strConnection;
19          }
20          //self cross-SendRequest dead-locking here !!!!
21          if (SendRequest(idOpen, strConnection, flags, (ar) => {
22              int res, ms; string errMsg;
23              ar.Load(out res).Load(out errMsg).Load(out ms);
24              lock (m_csDB) { //self dead-lock !!!!
25                  CleanRowset(); m_dbErrCode = res; m_lastReqId = idOpen;
26                  if (res == 0) {
27                      m_strConnection = errMsg; errMsg = "";
28                  }
29                  else
30                      m_strConnection = "";
31                  m_dbErrMsg = errMsg; m_ms = (tagManagementSystem)ms;
32                  m_parameters = 0; m_indexProc = 0; m_output = 0;
33              }
34              if (handler != null)
35                  handler(this, res, errMsg);
36          }, discarded, null)) {
37              return true;
38          }
39          if (strConnection != null)
40              m_strConnection = s;
41      } //end lock
42      return false;
43  }
```

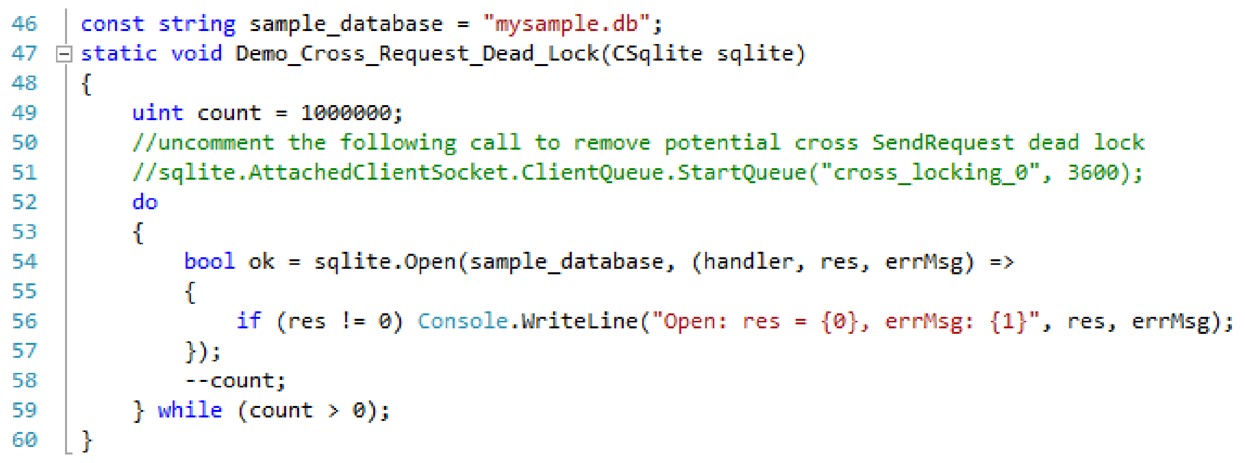
Figure 2: Cross request dead lock demonstration by method Open of class CAsyncDBHandler

The class *CAsyncDBHandler* has a member *m_csDB*, which is used to synchronize all its members, *m_flags*, *m_strConnection*, *m_dbErrMsg*, and others. It starts locking from line 14 through line 41. The lock is crossing the request at line 21, whose request data will be sent to a remote SocketPro server for processing. Further, the lock is used to lock all statements between lines 25 and 32 inside a callback or Lambda expression that will be called by one of socket pool threads.

When many requests are streamed from your calling thread and too much request data may be queued in memory at client side, SocketPro client core library will silently wait to prevent too many requests queued or too much memory consumed which leads to a dead lock. The dead lock is named as cross request dead lock within SocketPro technology, which may happen **only if all** the following three situations meet at the same time.

- Many requests are streamed at client side
- There is a lock cross request, and the lock is also used to synchronize other data within anyone of functions that are called by one of socket pool threads
- Request sending speed is faster than network and server processing speeds

To solve the issue completely, you may use a new locking algorithm so that there is no lock across the method *SendRequest* at line 21 in the above Figure 2. This is the first way. In case you don’t see the issue at glance, you can write a unit test code to detect the dead lock as shown in the below Figure 3.



```
const string sample_database = "mysample.db";
static void Demo_Cross_Request_Dead_Lock(CSqlite sqlite)
{
    uint count = 1000000;
    //uncomment the following call to remove potential cross SendRequest dead lock
    //sqlite.AttachedClientSocket.ClientQueue.StartQueue("cross_locking_0", 3600);
    do
    {
        bool ok = sqlite.Open(sample_database, (handler, res, errMsg) =>
        {
            if (res != 0) Console.WriteLine("Open: res = {0}, errMsg: {1}", res, errMsg);
        });
        --count;
    } while (count > 0);
}
```

Figure 3: A sample unit test code snippet to detect cross request dead lock within method Open

As hinted in the above Figure 3, you can open a local message queue to avoid this dead lock because requests can be queued into the message queue file instead of memory. This is the second way to get rid of the dead lock. However, this way may slow sending requests because of cost of saving requests into disk. As a software developer for writing asynchronous handler, you should do your best to use the first way instead. Just for your information, all client handlers except the class *CStreamingFile* inside an adapter use the first way to avoid the cross request dead lock.

As hinted at lines 225 and 226 in Figure 1, you can replicate the dead lock by replacing the original implementation with this bad one.

Sharing one single connection by multiple threads concurrently

SocketPro favors that multiple threads share one single connection so that requests are streamed for the best network efficiency and processing sequence in general. The class CSocketPool at client side provides two sets of methods to find a proper handler from one of calling threads. Multiple threads may send different requests possibly by the same handler through calling one of the first set of methods Seek and SeekByQueue as shown at left side of the below Figure 4. This is situation leading to no request streaming synchronization.

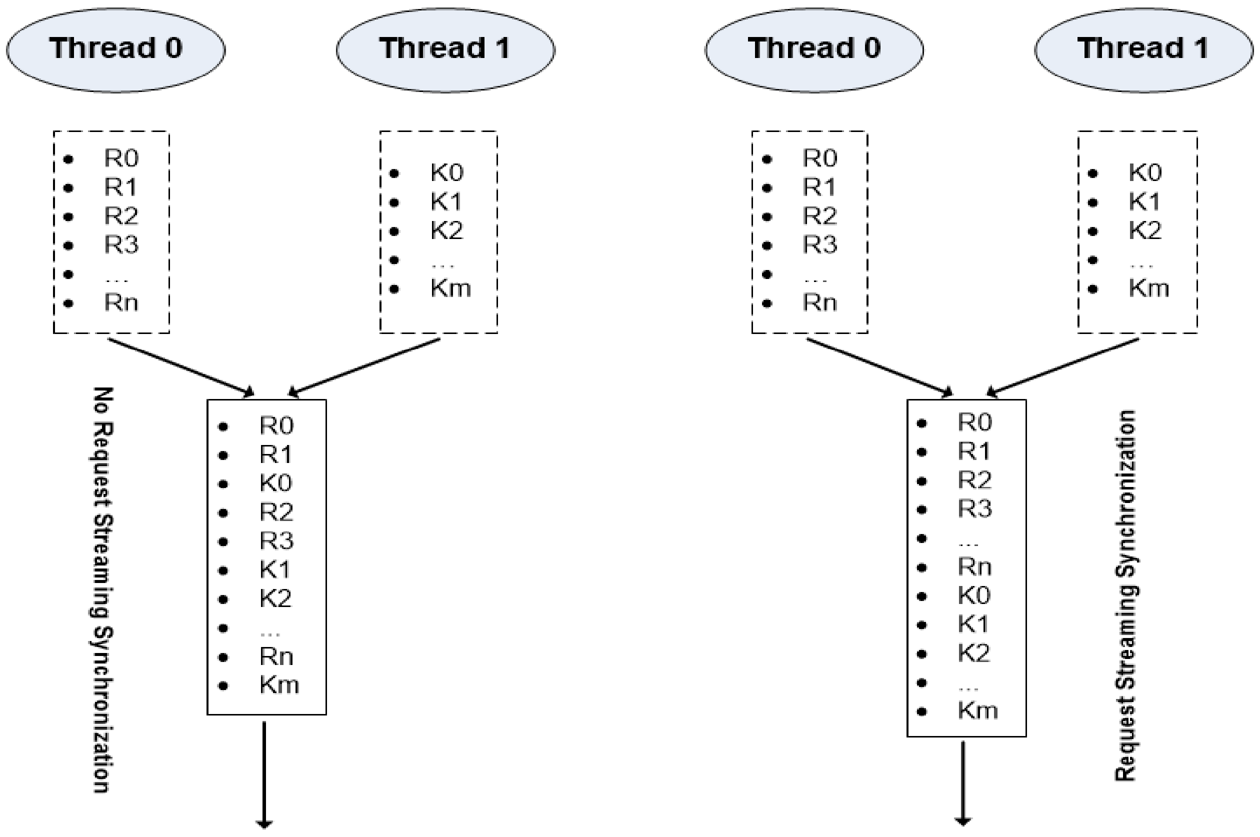
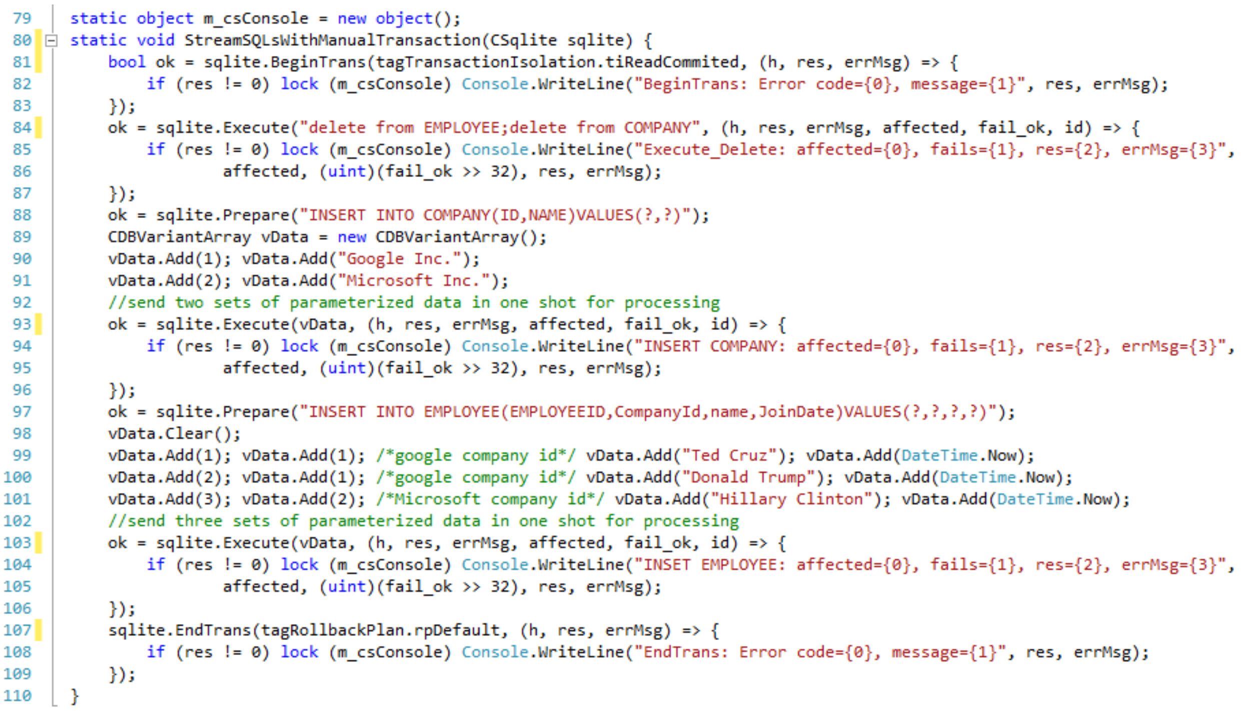

Thread 0
Thread 1
R0 R1 R2 R3 ... Rn
K0 K1 K2 ... Km
R0 R1 K0 R2 R3 K1 K2 ... Rn Km
No Request Streaming Synchronization
Thread 0
Thread 1
R0 R1 R2 R3 ... Rn
K0 K1 K2 ... Km
R0 R1 R2 R3 ... Rn K0 K1 K2 ... Km
Request Streaming Synchronization

Figure 4: Share one single socket connection by two calling threads

Under many cases, no request streaming synchronization is just fine if there is no request sequence requirement for a stream of requests. For example, a client sends a stream of SQL query statements from different working threads by use of socket connection. However, request streaming synchronization is often a must, which is shown at right side of the above Figure 4. For example, you like to update (insert, update and delete) database tables in manual transaction style. Another example is to execute SQLs by parameterized statements in such an order like preparing at first, sending multiple sets of parameter data at next, and executing at last. To make sure request streaming synchronization, we can use the method *Lock* of the class *CSocketPool*.

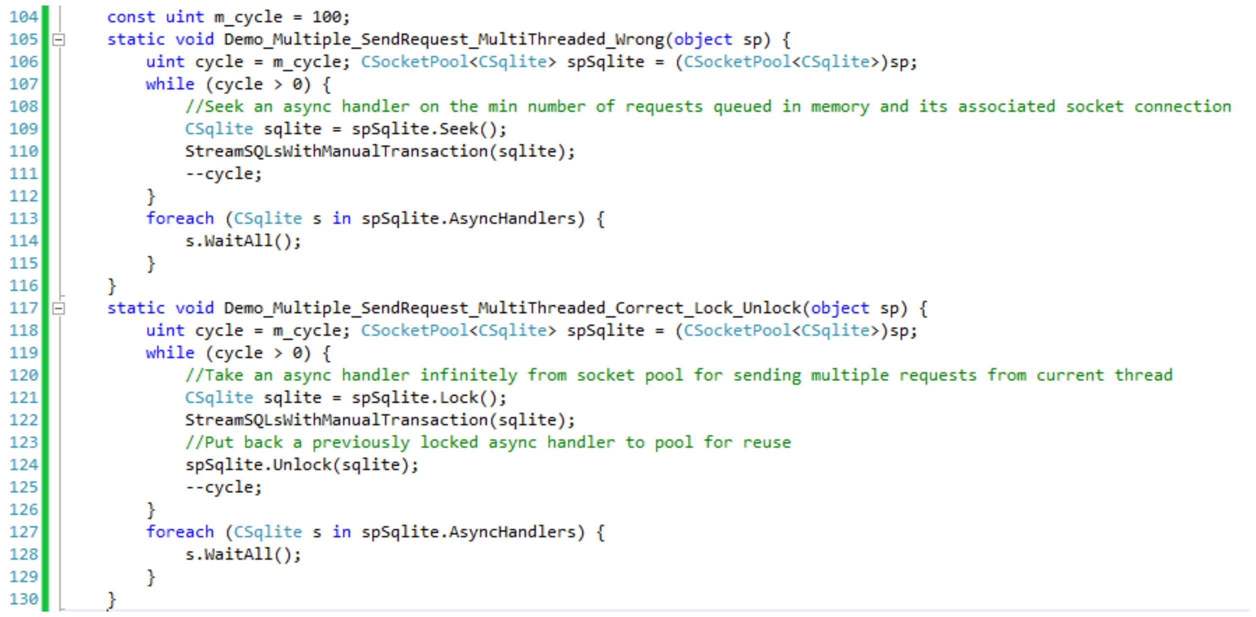
To demonstrate how to use the above two sets of methods, we need a sample of streaming a series of SQL statements in a manual transaction as shown in the below Figure 5.



```
79  static object m_csConsole = new object();
80  static void StreamSQLsWithManualTransaction(CSqlite sqlite) {
81      bool ok = sqlite.BeginTrans(tagTransactionIsolation.tiReadCommited, (h, res, errMsg) => {
82          if (res != 0) lock (m_csConsole) Console.WriteLine("BeginTrans: Error code={0}, message={1}", res, errMsg);
83      });
84      ok = sqlite.Execute("delete from EMPLOYEE;delete from COMPANY", (h, res, errMsg, affected, fail_ok, id) => {
85          if (res != 0) lock (m_csConsole) Console.WriteLine("Execute_Delete: affected={0}, fails={1}, res={2}, errMsg={3}",
86                  affected, (uint)(fail_ok >> 32), res, errMsg);
87      });
88      ok = sqlite.Prepare("INSERT INTO COMPANY(ID,NAME)VALUES(?,?)");
89      CDBVariantArray vData = new CDBVariantArray();
90      vData.Add(1); vData.Add("Google Inc.");
91      vData.Add(2); vData.Add("Microsoft Inc.");
92      //send two sets of parameterized data in one shot for processing
93      ok = sqlite.Execute(vData, (h, res, errMsg, affected, fail_ok, id) => {
94          if (res != 0) lock (m_csConsole) Console.WriteLine("INSERT COMPANY: affected={0}, fails={1}, res={2}, errMsg={3}",
95                  affected, (uint)(fail_ok >> 32), res, errMsg);
96      });
97      ok = sqlite.Prepare("INSERT INTO EMPLOYEE(EMPLOYEEID,CompanyId,name,JoinDate)VALUES(?,?,?,?)");
98      vData.Clear();
99      vData.Add(1); vData.Add(1); /*google company id*/ vData.Add("Ted Cruz"); vData.Add(DateTime.Now);
100     vData.Add(2); vData.Add(1); /*google company id*/ vData.Add("Donald Trump"); vData.Add(DateTime.Now);
101     vData.Add(3); vData.Add(2); /*Microsoft company id*/ vData.Add("Hillary Clinton"); vData.Add(DateTime.Now);
102     //send three sets of parameterized data in one shot for processing
103     ok = sqlite.Execute(vData, (h, res, errMsg, affected, fail_ok, id) => {
104         if (res != 0) lock (m_csConsole) Console.WriteLine("INSET EMPLOYEE: affected={0}, fails={1}, res={2}, errMsg={3}",
105                 affected, (uint)(fail_ok >> 32), res, errMsg);
106     });
107     sqlite.EndTrans(tagRollbackPlan.rpDefault, (h, res, errMsg) => {
108         if (res != 0) lock (m_csConsole) Console.WriteLine("EndTrans: Error code={0}, message={1}", res, errMsg);
109     });
110 }
```

Figure 5: Stream a number of SQL statements within a manual transaction

The above Figure 5 starts a manual transaction at line 81. Afterwards, it does deleting all records of tables EMPLOYEE and COMPANY at line 84. Next, it sends a request for preparing at line 88. Next, it executes two inserts with two sets of data in one shot. Next, it sends a new prepare statement at line 97. At end, it executes three inserts with three sets of data in batch. Finally, it ends the manual transaction at line 107. Totally, there are seven requests streamed, which must be synchronized with no overlapping among threads at client side for properly processing at remote server side as shown in the bottom method of the below Figure 6.



```
const uint m_cycle = 100;
static void Demo_Multiple_SendRequest_MultiThreaded_Wrong(object sp) {
    uint cycle = m_cycle; CSocketPool<CSqlite> spSqlite = (CSocketPool<CSqlite>)sp;
    while (cycle > 0) {
        //Seek an async handler on the min number of requests queued in memory and its associated socket connection
        CSqlite sqlite = spSqlite.Seek();
        StreamSQLsWithManualTransaction(sqlite);
        --cycle;
    }
    foreach (CSqlite s in spSqlite.AsyncHandlers) {
        s.WaitAll();
    }
}
static void Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock(object sp) {
    uint cycle = m_cycle; CSocketPool<CSqlite> spSqlite = (CSocketPool<CSqlite>)sp;
    while (cycle > 0) {
        //Take an async handler infinitely from socket pool for sending multiple requests from current thread
        CSqlite sqlite = spSqlite.Lock();
        StreamSQLsWithManualTransaction(sqlite);
        //Put back a previously locked async handler to pool for reuse
        spSqlite.Unlock(sqlite);
        --cycle;
    }
    foreach (CSqlite s in spSqlite.AsyncHandlers) {
        s.WaitAll();
    }
}
```

Figure 6: Sharing a connection among multiple calling threads by methods, Seek and Lock

It is noted that the top method *Demo_Multiple_SendRequest_MultiThreaded_Wrong* is good for streaming a series of SQL queries but not good for streaming a series of update or parameterized statements, which is demonstrated at lines 238 through 241 of Figure 1. Contrarily, the bottom one *Demo_Multiple_SendRequest_MultiThreaded_Correct_Lock_Unlock* is good for all cases, which is verified at lines 244 through 247 of Figure 1.

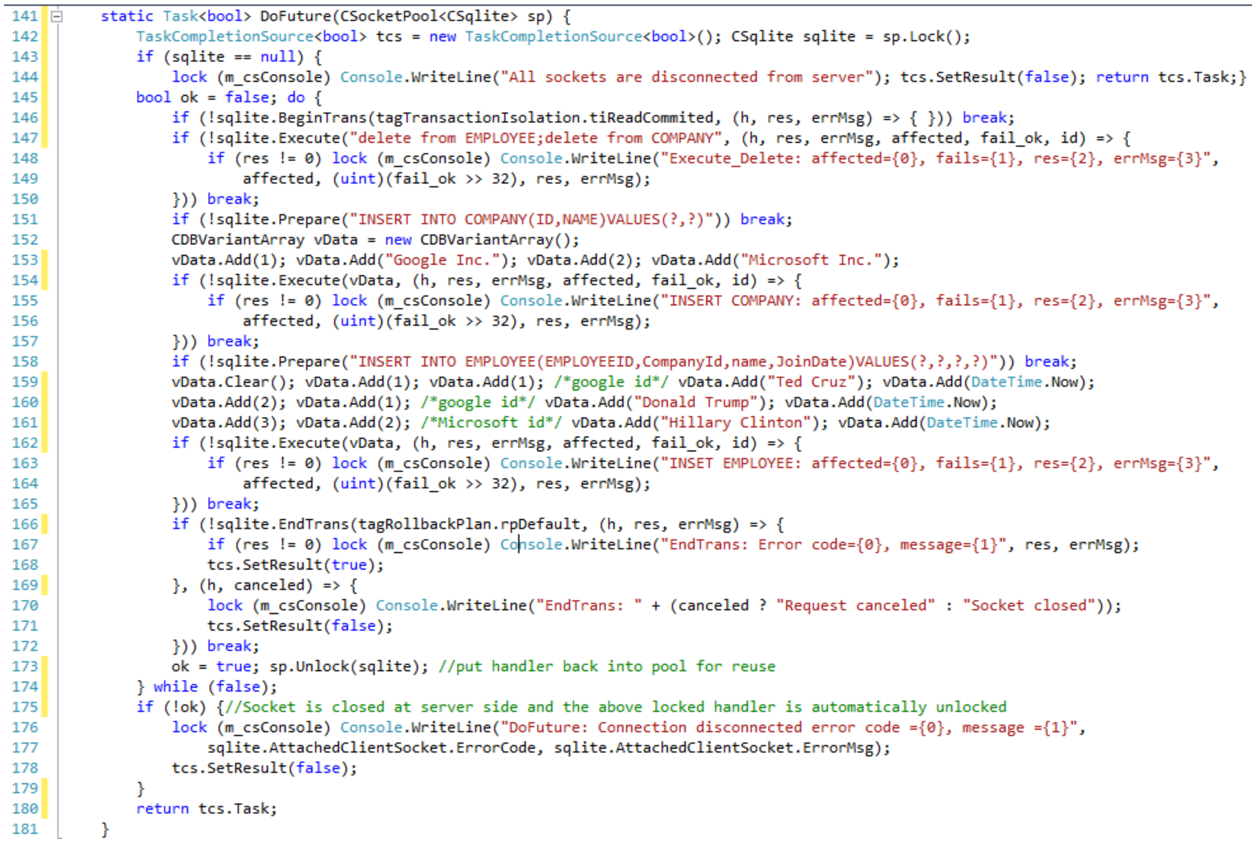
Locking one single connection from a pool of sockets

As shown at bottom of Figure 6, a calling thread could lock a handler from a socket pool by the method *Lock* first so that no other calling threads are able to get the same handler at a moment. By this way, it ensures request streaming synchronization as shown at right side of Figure 4, which ensures a remote server to process all streamed requests properly in order. At last, don't forget calling the method *Unlock* and putting a previously locked handler back into pool for reuse by other threads as soon as possible.

Waiting one result for one specific request

We use the method *WaitAll* at client side to wait until all requests queued on one socket connection are processed and returned. This is usually fine within one thread as shown in most of samples for code snippet clarity. However, the method may not work as you expected in case a handler or its associated socket connection is being shared by multiple calling threads. The method may be not returned to a calling thread if another calling thread is keeping sending requests continuously through the same handler or sever is processing requests at slow speed.

Under such a situation, we must have a way to wait for a specific request returned. We could use future for this purpose. SocketPro adapters for Python and Java have a simple class *UFuture* for help. However, there is no such class in .NET and C++ adapters, as it is very simple for us to write code for waiting one specific request *EndTrans* at line 166 as shown in the below Figure 7.



```
141     static Task<bool> DoFuture(CSocketPool<CSqlite> sp) {
142         TaskCompletionSource<bool> tcs = new TaskCompletionSource<bool>(); CSqlite sqlite = sp.Lock();
143         if (sqlite == null) {
144             lock (m_csConsole) Console.WriteLine("All sockets are disconnected from server"); tcs.SetResult(false); return tcs.Task;}
145         bool ok = false; do {
146             if (!sqlite.BeginTrans(tagTransactionIsolation.tiReadCommited, (h, res, errMsg) => { })) break;
147             if (!sqlite.Execute("delete from EMPLOYEE;delete from COMPANY", (h, res, errMsg, affected, fail_ok, id) => {
148                 if (res != 0) lock (m_csConsole) Console.WriteLine("Execute_Delete: affected={0}, fails={1}, res={2}, errMsg={3}",
149                     affected, (uint)(fail_ok >> 32), res, errMsg);
150             })) break;
151             if (!sqlite.Prepare("INSERT INTO COMPANY(ID,NAME)VALUES(?,?)")) break;
152             CDBVariantArray vData = new CDBVariantArray();
153             vData.Add(1); vData.Add("Google Inc."); vData.Add(2); vData.Add("Microsoft Inc.");
154             if (!sqlite.Execute(vData, (h, res, errMsg, affected, fail_ok, id) => {
155                 if (res != 0) lock (m_csConsole) Console.WriteLine("INSERT COMPANY: affected={0}, fails={1}, res={2}, errMsg={3}",
156                     affected, (uint)(fail_ok >> 32), res, errMsg);
157             })) break;
158             if (!sqlite.Prepare("INSERT INTO EMPLOYEE(EMPLOYEEID,CompanyId,name,JoinDate)VALUES(?,?,?,?)")) break;
159             vData.Clear(); vData.Add(1); vData.Add(1); /*google id*/ vData.Add("Ted Cruz"); vData.Add(DateTime.Now);
160             vData.Add(2); vData.Add(1); /*google id*/ vData.Add("Donald Trump"); vData.Add(DateTime.Now);
161             vData.Add(3); vData.Add(2); /*Microsoft id*/ vData.Add("Hillary Clinton"); vData.Add(DateTime.Now);
162             if (!sqlite.Execute(vData, (h, res, errMsg, affected, fail_ok, id) => {
163                 if (res != 0) lock (m_csConsole) Console.WriteLine("INSET EMPLOYEE: affected={0}, fails={1}, res={2}, errMsg={3}",
164                     affected, (uint)(fail_ok >> 32), res, errMsg);
165             })) break;
166             if (!sqlite.EndTrans(tagRollbackPlan.rpDefault, (h, res, errMsg) => {
167                 if (res != 0) lock (m_csConsole) Console.WriteLine("EndTrans: Error code={0}, message={1}", res, errMsg);
168                 tcs.SetResult(true);
169             }, (h, canceled) => {
170                 lock (m_csConsole) Console.WriteLine("EndTrans: " + (canceled ? "Request canceled" : "Socket closed"));
171                 tcs.SetResult(false);
172             })) break;
173             ok = true; sp.Unlock(sqlite); //put handler back into pool for reuse
174         } while (false);
175         if (!ok) {//Socket is closed at server side and the above locked handler is automatically unlocked
176             lock (m_csConsole) Console.WriteLine("DoFuture: Connection disconnected error code ={0}, message ={1}",
177                 sqlite.AttachedClientSocket.ErrorCode, sqlite.AttachedClientSocket.ErrorMsg);
178             tcs.SetResult(false);
179         }
180         return tcs.Task;
181     }
```

Figure 7: Wait for a specific request processed by use of future

It is noted that this implementation is nearly the same as one of Figure 5. After creating a task at line 142, we must call the method *SetResult* properly, which is shown at lines 144, 168, 171 and 178, after either an error happens or a result is returned from server.

You can see the lines 250 through 253 in Figure 1 to find how to call the sample method *DoFuture*.

Finally, it is pointed that you'd better use the future instead of *WaitAll* within multi-thread environments (for example, middle tier and web server developments) in most cases. The method WaitAll is given for simplicity, clarity and single thread environment.