Brief introduction of SocketPro continuous SQL-stream sending and processing system (Part 1: SQLite)

Introduction

Most of client server database systems only support synchronous communication between client and backend database by use of blocking socket and some chatty protocol that requires a client or server to wait for an acknowledgement before sending a new chunk of data. The wait time, which is also called as latency, could be starting from a few tenths for a local area network (LAN) to hundreds of milliseconds for a wide area network (WAN). Large wait times can significantly increase response time and degrade the quality of an application.

Fortunately, UDAParts has developed a powerful and secure communication framework named as SocketPro, which is written with continuous inline request/result batching and real-time stream processing capabilities by use of asynchronous data transferring and parallel computation for the best network efficiency, development simplicity, performance, scalability, and many great and even unique features at the site (https://github.com/udaparts/socketpro).

Further, UDAParts has applied the powerful SocketPro framework onto a number of popular databases such as SQLite, MySQL and MS SQL as well as others through ODBC drivers to support continuous SQL-stream sending and processing. Additionally, most of these components for databases are **totally free forever** to the public with opened source codes for you to study and extend them to meet your complex needs.

For reduction of learning complexity, we use SQLite database as the first sample for the first article, and MySQL as the second sample for the coming second article.

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 usqlite inside the directory socketpro/samples/module_sample.

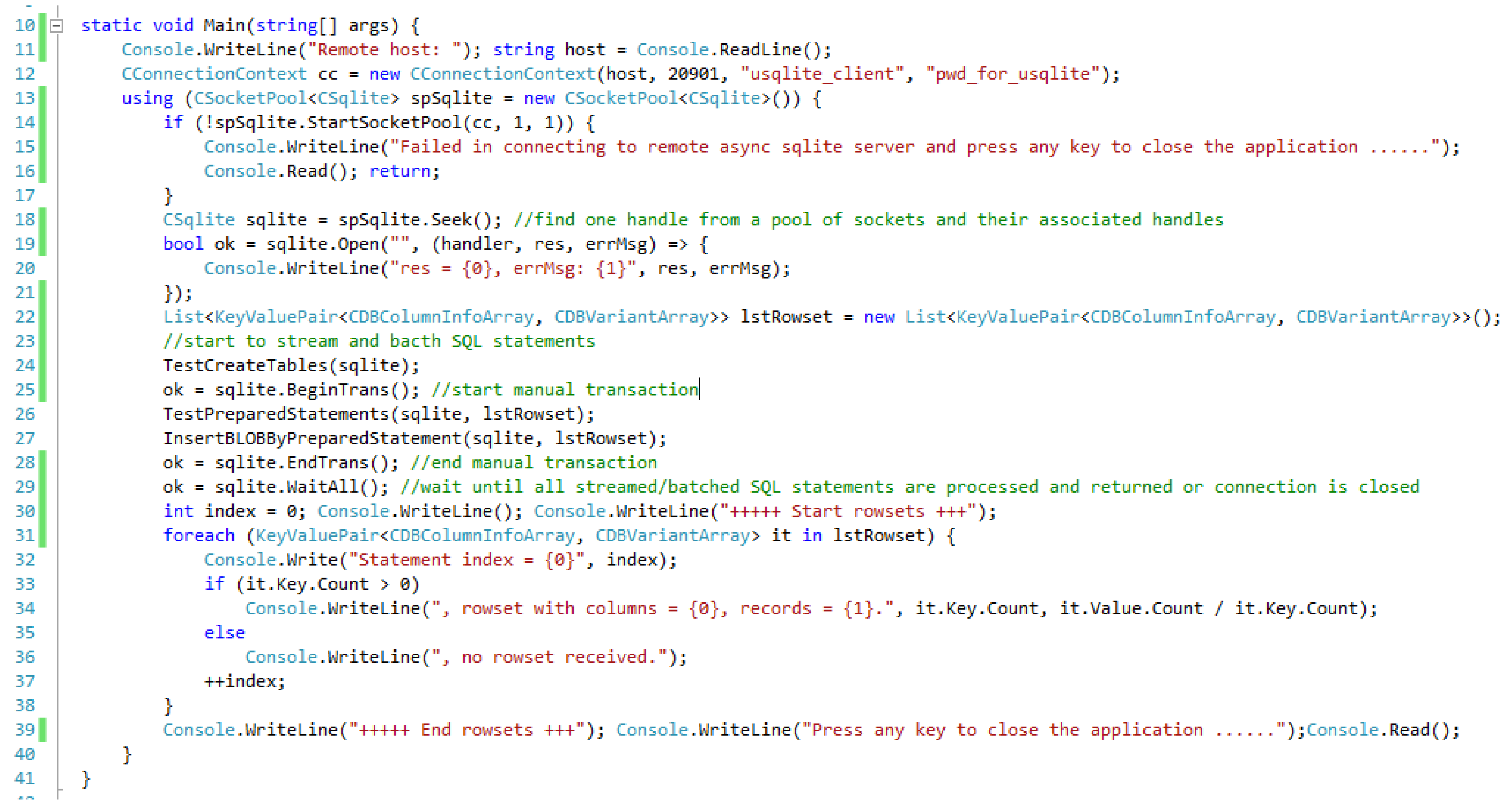
You can see these samples are created from .NET, C/C++, Java and Python development environments. They can be compiled and run on either Linux or windows platforms. In case you are not used to C/C++ development, UDAParts also distributes pre-compiled test applications, test_ssqlite for server and test_csqlite for client inside the directory socketpro/bin/(win|linux) because these test applications are written from C/C++. In addition, you can figure out how to load a SocketPro 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# code (socketpro/samples/module_sample/usqlite/test_csahrp) at this article for explanations.

You should distribute system libraries inside the directory of 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. It is noted that we just use one pool for clear demonstration here, and the pool is made of one thread and one socket for this sample at client side as shown in the below Figure 1.



```
static void Main(string[] args) {
    Console.WriteLine("Remote host: "); string host = Console.ReadLine();
    CConnectionContext cc = new CConnectionContext(host, 20901, "usqlite_client", "pwd_for_usqlite");
    using (CSocketPool<CSqlite> spSqlite = new CSocketPool<CSqlite>()) {
        if (!spSqlite.StartSocketPool(cc, 1, 1)) {
            Console.WriteLine("Failed in connecting to remote async sqlite server and press any key to close the application ......");
            Console.Read(); return;
        }
        CSqlite sqlite = spSqlite.Seek(); //find one handle from a pool of sockets and their associated handles
        bool ok = sqlite.Open("", (handler, res, errMsg) => {
            Console.WriteLine("res = {0}, errMsg: {1}", res, errMsg);
        });
        List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>> lstRowset = new List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>>();
        //start to stream and bacth SQL statements
        TestCreateTables(sqlite);
        ok = sqlite.BeginTrans(); //start manual transaction
        TestPreparedStatements(sqlite, lstRowset);
        InsertBLOBByPreparedStatement(sqlite, lstRowset);
        ok = sqlite.EndTrans(); //end manual transaction
        ok = sqlite.WaitAll(); //wait until all streamed/batched SQL statements are processed and returned or connection is closed
        int index = 0; Console.WriteLine(); Console.WriteLine("+++++ Start rowsets +++");
        foreach (KeyValuePair<CDBColumnInfoArray, CDBVariantArray> it in lstRowset) {
            Console.Write("Statement index = {0}", index);
            if (it.Key.Count > 0)
                Console.WriteLine(", rowset with columns = {0}, records = {1}.", it.Key.Count, it.Value.Count / it.Key.Count);
            else
                Console.WriteLine(", no rowset received.");
            ++index;
        }
        Console.WriteLine("+++++ End rowsets +++"); Console.WriteLine("Press any key to close the application ......");Console.Read();
    }
}
```

Figure 1: Main function for demonstration of use of SocketPro SQL-stream system 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 14 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 SQLite handler at line 18.

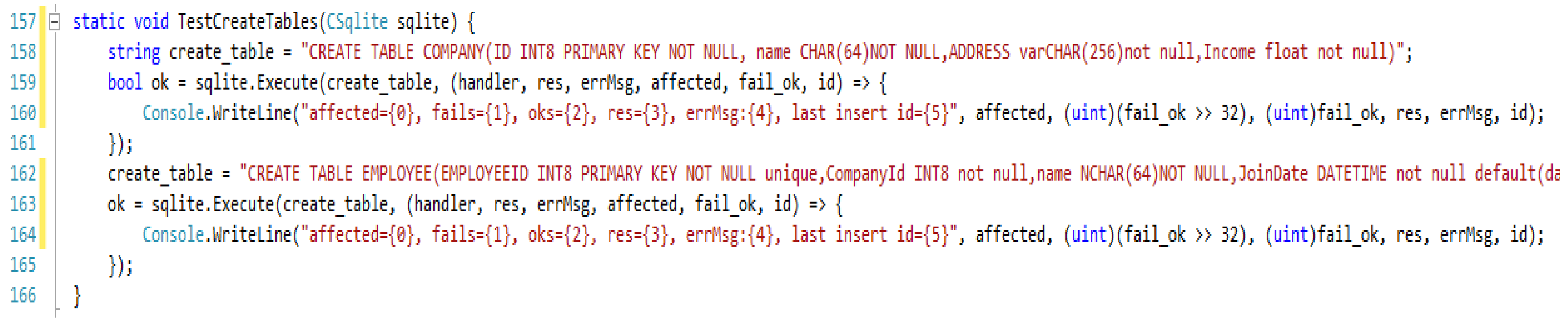
Opening database: We can send a request to open a SQLite server database at line 19. If the first input is an empty or null string as shown at this example, we are opening one instance of server global database *usqlite.db*, for example. If you like to create an own database, you can simply give a non-empty valid string. In addition, you need to set a callback or Lambda expression for tracking returning error message from server side if you like as shown at lines 19 through 21. It is noted that SocketPro supports only asynchronous data transferring between client and server so that a request could be inputted with one or more callbacks for processing returning data. This is completely different from synchronous data transferring. In addition, we create an instance of container that is used to receive all sets of records in coming queries at line 22.

Streaming SQL statements: Keep in mind that SocketPro supports streaming all types of any number of requests on one non-blocking socket session effortlessly by design. Certainly, we are able to stream all SQL statements as well as others as shown at lines 24 through 28. All SocketPro SQL-stream services support this particular feature for the best network efficiency, which significantly improves data accessing performance. As far as we know, you cannot find such a wonderful feature from other technologies. If you find one, please let us know. Like normal database accessing APIs, SocketPro SQL-stream technology supports manual transaction too as shown at line 25 and 28. We are going to elaborate the three functions, *TestCreateTables*, *TestPreparedStatements* and *InsertBLOBByPreparedStatement* in successive sections.

Waiting until all processed: Since SocketPro only supports asynchronous data transferring, SocketPro must provide a way to wait until all requests and returning results are sent, returned and processed. SocketPro comes one unique method *WaitAll* at client side to serve this purpose as shown at line 29. If you like, you can use this method to convert all asynchronous requests into synchronous ones.

TestCreateTables

This function is internally made of sending two SQL DDL statements for creating two tables, *COMPANY* and *EMPLOYEE*, as shown in the below Figure 2.



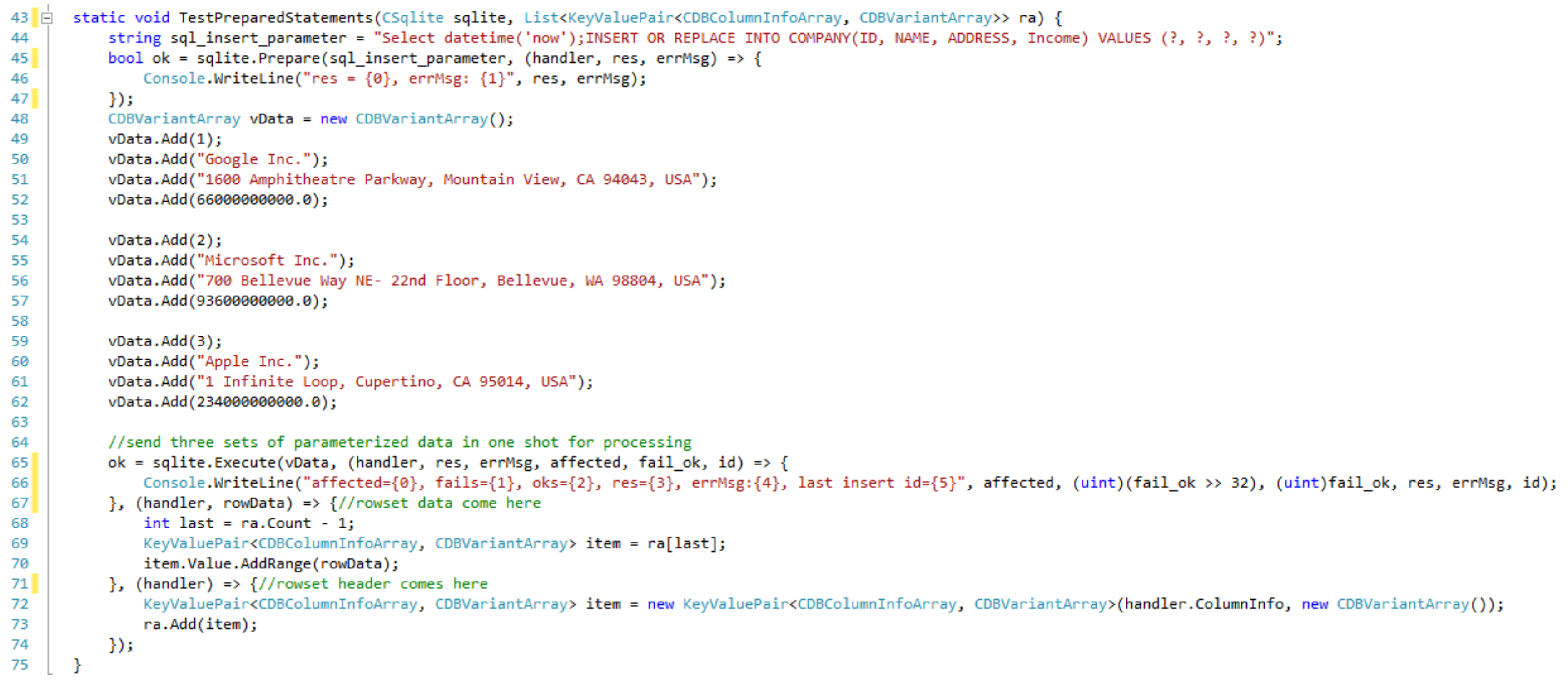
```
static void TestCreateTables(CSqlite sqlite) {
    string create_table = "CREATE TABLE COMPANY(ID INT8 PRIMARY KEY NOT NULL, name CHAR(64)NOT NULL,ADDRESS varCHAR(256)not null,Income float not null)";
    bool ok = sqlite.Execute(create_table, (handler, res, errMsg, affected, fail_ok, id) => {
        Console.WriteLine("affected={0}, fails={1}, oks={2}, res={3}, errMsg:{4}, last insert id={5}", affected, (uint)(fail_ok >> 32), (uint)fail_ok, res, errMsg, id);
    });
    create_table = "CREATE TABLE EMPLOYEE(EMPLOYEEID INT8 PRIMARY KEY NOT NULL unique,CompanyId INT8 not null,name NCHAR(64)NOT NULL,JoinDate DATETIME not null default(da
    ok = sqlite.Execute(create_table, (handler, res, errMsg, affected, fail_ok, id) => {
        Console.WriteLine("affected={0}, fails={1}, oks={2}, res={3}, errMsg:{4}, last insert id={5}", affected, (uint)(fail_ok >> 32), (uint)fail_ok, res, errMsg, id);
    });
}
```

Figure 2: Creating two tables in streaming by SocketPro SQL-stream technology

You can execute any number of SQL statements in stream as shown in the Figure 2. Each of requests consists of one input SQL statement and one optional callback (or Lambda expression) for tracking expected returning results. Again, this is different from common database accessing approach as SocketPro uses asynchronous data transferring for communication.

TestPreparedStatements

SocketPro SQL-stream technology supports preparing SQL statement just like common database accessing APIs. Particularly, SocketPro SQL-stream technology even supports preparing multiple SQL statements at one shot for SQLite server database as shown in the below Figure 3.



```
static void TestPreparedStatements(CSqlite sqlite, List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>> ra) {
    string sql_insert_parameter = "Select datetime('now');INSERT OR REPLACE INTO COMPANY(ID, NAME, ADDRESS, Income) VALUES (?, ?, ?, ?)";
    bool ok = sqlite.Prepare(sql_insert_parameter, (handler, res, errMsg) => {
        Console.WriteLine("res = {0}, errMsg: {1}", res, errMsg);
    });
    CDBVariantArray vData = new CDBVariantArray();
    vData.Add(1);
    vData.Add("Google Inc.");
    vData.Add("1600 Amphitheatre Parkway, Mountain View, CA 94043, USA");
    vData.Add(66000000000.0);

    vData.Add(2);
    vData.Add("Microsoft Inc.");
    vData.Add("700 Bellevue Way NE- 22nd Floor, Bellevue, WA 98804, USA");
    vData.Add(93600000000.0);

    vData.Add(3);
    vData.Add("Apple Inc.");
    vData.Add("1 Infinite Loop, Cupertino, CA 95014, USA");
    vData.Add(234000000000.0);

    //send three sets of parameterized data in one shot for processing
    ok = sqlite.Execute(vData, (handler, res, errMsg, affected, fail_ok, id) => {
        Console.WriteLine("affected={0}, fails={1}, oks={2}, res={3}, errMsg:{4}, last insert id={5}", affected, (uint)(fail_ok >> 32), (uint)fail_ok, res, errMsg, id);
    }, (handler, rowData) => {//rowset data come here
        int last = ra.Count - 1;
        KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = ra[last];
        item.Value.AddRange(rowData);
    }, (handler) => {//rowset header comes here
        KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = new KeyValuePair<CDBColumnInfoArray, CDBVariantArray>(handler.ColumnInfo, new CDBVariantArray());
        ra.Add(item);
    });
}
```

Figure 3: Sending multiple sets of parameters for processing multiple SQL statements in one shot by SocketPro SQL-stream technology

It is noted that the sample preparing SQL statement consists of one query and one insert statements. When the function is called, a client will expect three sets of records returned and three records inserted into the table *COMPANY*. The sample is designed for demonstrating the power of SocketPro SQL-stream technology. In reality, you probably don't prepare a combined SQL statement having multiple basic SQL statements. If you use a parameterized statement, you are required to send a prepare request as shown at line 45. After obtaining an array of data as shown at lines 49 through 62, you can send multiple sets of parameter data for processing from client to server in one single shot as shown at line 65. If you have a large amount of data, you could call the request at line 65 repeatedly without needing to prepare a statement again.

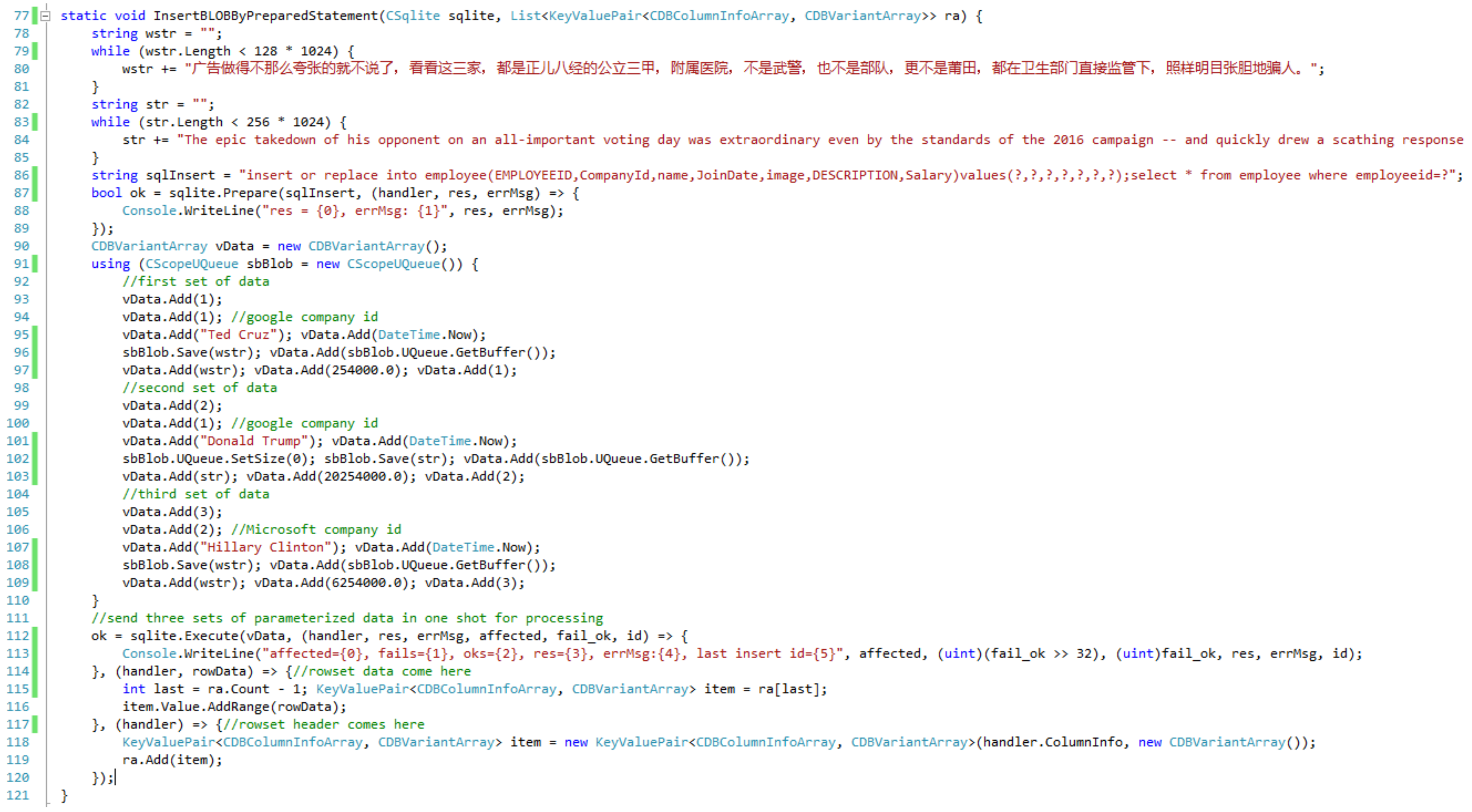
Next, we need more details for how to handle returning record sets. The request at line 65 has three callbacks or Lambda expressions for the second, third and fourth input parameters except the first input for parameter data array. Whenever a record set is coming, the third callback will be automatically called by SQLite client handler for record set column information. If actual records are available, the second callback will be called and you can populate data into a container as shown at lines 67 through 70. At the end, the first callback will be called for you to track the number of affected records and last insert identification number if successful. If we take the Figure 3 as a sample, the third callback will be called three times and the first callback will be called one time only, but it is expected that the times of calling the second callback is dependent on both the number of records and the size of one record.

InsertBLOBByPreparedStatement

Now, you can see SocketPro SQL-stream technology provides all required features for accessing a backend database. Before the end of this article, we are going to use the sample to show how to handle large binary and text objects within SocketPro-stream technology. Usually it is difficult to access large objects inside databases efficiently. However, it is truly very simple with SocketPro SQL-stream technology for both development and efficiency as shown at the below Figure 4.

After looking through the code snippet in Figure 4, you would find that this code snippet is really the same as one in the previous Figure 3 although this code snippet is longer. Therefore, this approach is really a good thing for a software developer to reuse SocketPro SQL-stream technology for handling all types of database table fields in the same coding style for easy development.

SocketPro always divides a large binary or text object into chunks first at both client and server sides. Afterwards, SocketPro sends these smaller chunks to the other side. At end, SocketPro will reconstruct the original large binary or text object from collected smaller chunks. This happens silently at run time for reduction of memory foot print.



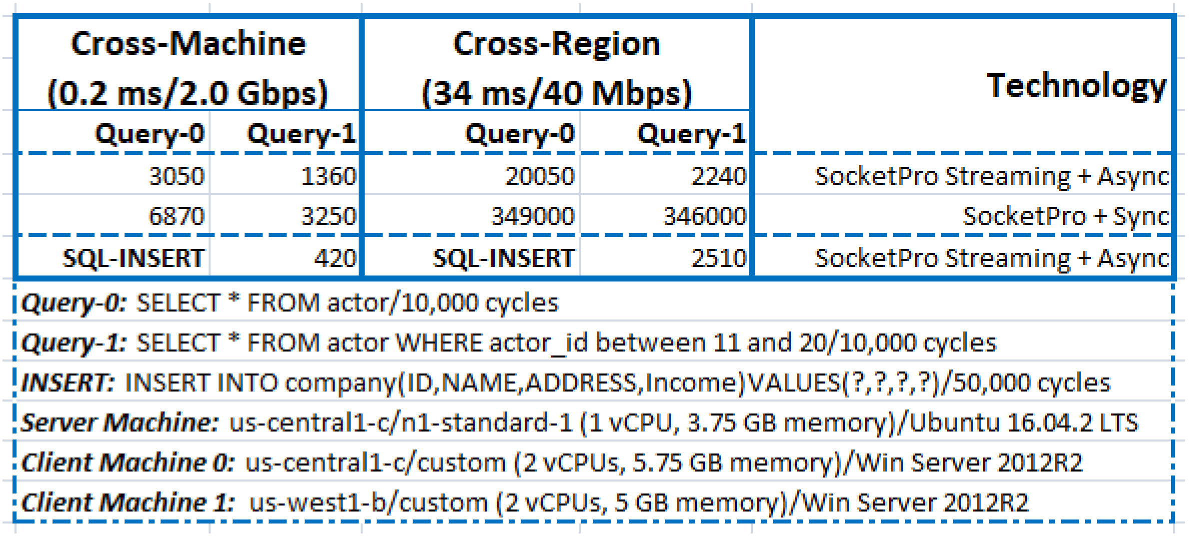
```
static void InsertBLOBByPreparedStatement(CSqlite sqlite, List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>> ra) {
    string wstr = "";
    while (wstr.Length < 128 * 1024) {
        wstr += "广告做得不那么夸张的就不说了，看看这三家，都是正儿八经的公立三甲，附属医院，不是武警，也不是部队，更不是莆田，都在卫生部门直接监管下，照样明目张胆地骗人。";
    }
    string str = "";
    while (str.Length < 256 * 1024) {
        str += "The epic takedown of his opponent on an all-important voting day was extraordinary even by the standards of the 2016 campaign -- and quickly drew a scathing response
    }
    string sqlInsert = "insert or replace into employee(EMPLOYEEID,CompanyId,name,JoinDate,image,DESCRIPTION,Salary)values(?,?,?,?,?,?,?);select * from employee where employeeid=?";
    bool ok = sqlite.Prepare(sqlInsert, (handler, res, errMsg) => {
        Console.WriteLine("res = {0}, errMsg: {1}", res, errMsg);
    });
    CDBVariantArray vData = new CDBVariantArray();
    using (CScopeUQueue sbBlob = new CScopeUQueue()) {
        //first set of data
        vData.Add(1);
        vData.Add(1); //google company id
        vData.Add("Ted Cruz"); vData.Add(DateTime.Now);
        sbBlob.Save(wstr); vData.Add(sbBlob.UQueue.GetBuffer());
        vData.Add(wstr); vData.Add(254000.0); vData.Add(1);
        //second set of data
        vData.Add(2);
        vData.Add(1); //google company id
        vData.Add("Donald Trump"); vData.Add(DateTime.Now);
        sbBlob.UQueue.SetSize(0); sbBlob.Save(str); vData.Add(sbBlob.UQueue.GetBuffer());
        vData.Add(str); vData.Add(20254000.0); vData.Add(2);
        //third set of data
        vData.Add(3);
        vData.Add(2); //Microsoft company id
        vData.Add("Hillary Clinton"); vData.Add(DateTime.Now);
        sbBlob.Save(wstr); vData.Add(sbBlob.UQueue.GetBuffer());
        vData.Add(wstr); vData.Add(6254000.0); vData.Add(3);
    }
    //send three sets of parameterized data in one shot for processing
    ok = sqlite.Execute(vData, (handler, res, errMsg, affected, fail_ok, id) => {
        Console.WriteLine("affected={0}, fails={1}, oks={2}, res={3}, errMsg:{4}, last insert id={5}", affected, (uint)(fail_ok >> 32), (uint)fail_ok, res, errMsg, id);
    }, (handler, rowData) => {//rowset data come here
        int last = ra.Count - 1; KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = ra[last];
        item.Value.AddRange(rowData);
    }, (handler) => {//rowset header comes here
        KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = new KeyValuePair<CDBColumnInfoArray, CDBVariantArray>(handler.ColumnInfo, new CDBVariantArray());
        ra.Add(item);
    });
}
```

Figure 4: Insert and query tables having multiple large binary and text objects with SocketPro SQL-stream technology

Performance study

SocketPro SQL-stream technology has excellent performance in database data accessing for both query and update too. You can see two performance test projects (cppperf and netperf) available at socketpro/samples/module_sample/usqlite/DBPerf/. The first sample is written by C++ and the other by C#. In addition, MySQL *sakila* sample database, which is located at the directory socketpro/samples/module_sample/usqlite/DBPerf, is used for you to play after running the sample test_csqlite for creating a global SQLite database *usqlite.db*.

See the performance study data of the below Figure 5, which is obtained from three cheap Google cloud virtual machines with solid state drive for free evaluation. All data are time required in millisecond for executing 10,000 queries and 50,000 inserts. The performance study is also focused on influence of network latency on SQL accessing speed.



Cross-Machine (0.2 ms/2.0 Gbps)		Cross-Region (34 ms/40 Mbps)		Technology
Query-0	**Query-1**	**Query-0**	**Query-1**	
3050	1360	20050	2240	SocketPro Streaming + Async
6870	3250	349000	346000	SocketPro + Sync
SQL-INSERT	420	**SQL-INSERT**	2510	SocketPro Streaming + Async

Query-0: SELECT * FROM actor/10,000 cycles

Query-1: SELECT * FROM actor WHERE actor_id between 11 and 20/10,000 cycles

INSERT: INSERT INTO company(ID,NAME,ADDRESS,Income)VALUES(?,?,?,?)/50,000 cycles

Server Machine: us-central1-c/n1-standard-1 (1 vCPU, 3.75 GB memory)/Ubuntu 16.04.2 LTS

Client Machine 0: us-central1-c/custom (2 vCPUs, 5.75 GB memory)/Win Server 2012R2

Client Machine 1: us-west1-b/custom (2 vCPUs, 5 GB memory)/Win Server 2012R2

Figure 5: SQLite streaming performance study data of SocketPro SQL-stream technology on three cheap Google cloud virtual machines

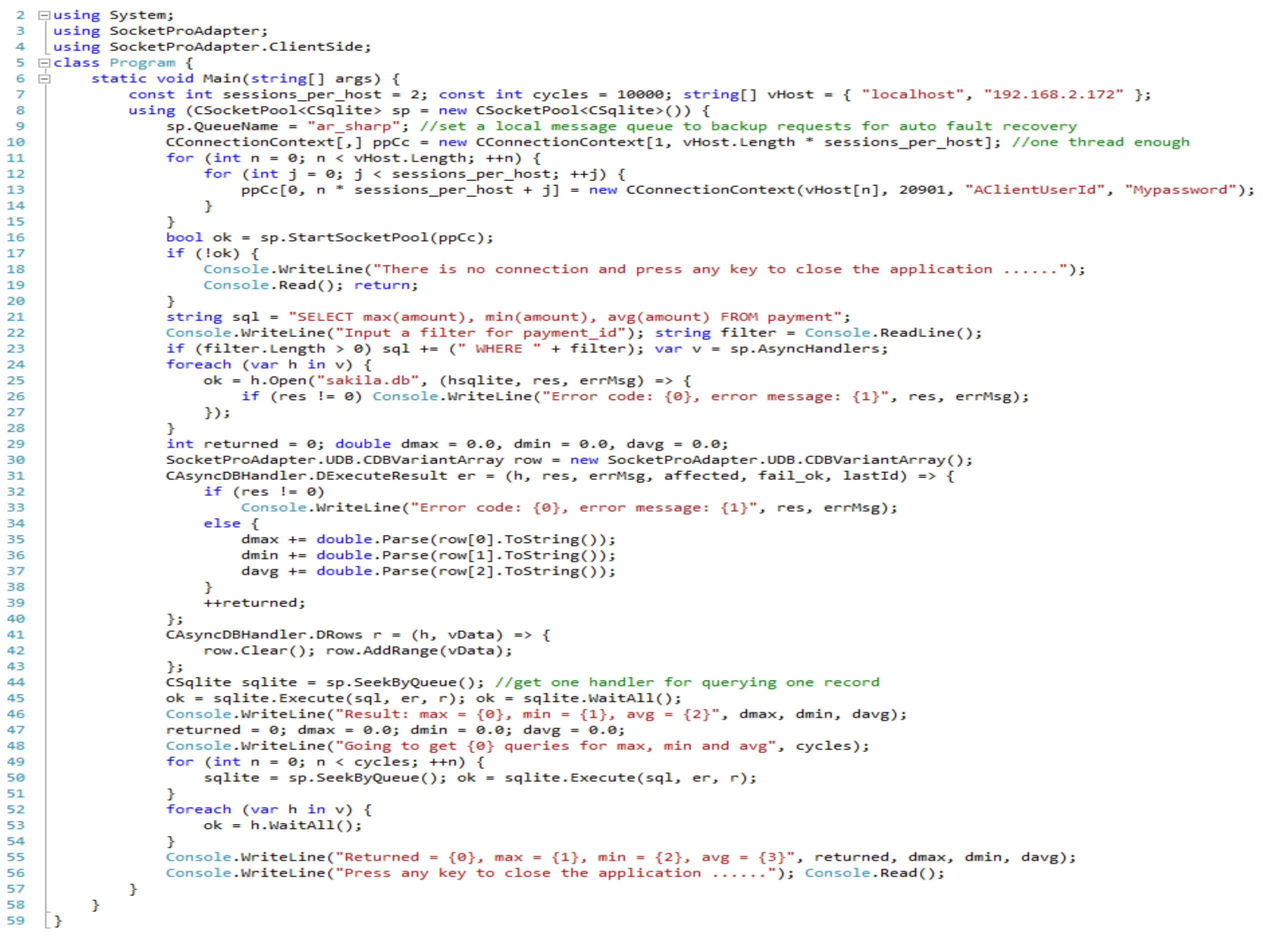
Our performance study shows that it is easy to get query executed at the speed of 7,400 (10,000/1.36) times per second and socket connection. For inserting records, you can easily get the speed like 120,000 (50,000/0.42) inserts per second for SQLite on local area network (LAN, cross-machine). SocketPro streaming could improve 140% in performance over traditional non-streaming approach (SocketPro + Sync).

In regards to wide area network (WAN, cross-region), the query speed could be 4,000 (10,000/2.24) times per second and socket connection. For inserting records, the speed could easily be 20,000 records (50,000/2.51) per second. Contrarily, the query speed will be as low as 30 queries per second on WAN if a client uses traditional communication way (non-streaming) for database accessing because of high latency. SocketPro SQL streaming can be more than 150 (346000/2240) times faster over non-streaming technology if database backend processing time is ignorable in comparison to IO communication time on WAN (cross-region) having a high latency. After analyzing the performance data at the above Figure 5, you will find SocketPro streaming technology is truly great for speeding up not only local but also remoting database accessing.

The above performance study was completed on WAN having bandwidth around 30 Mbps for cross-region communication. It is imagined that performance data for WAN would be much better if the test WAN have better network bandwidth. Further, SocketPro supports inline compression too, but this test study doesn't use this feature. If SocketPro inline compression feature is employed, its streaming test data will be further improved on WAN. At last, the performance study is completed on cheap virtual machines with one or two CPUs only. The performance data would be considerably improved if dedicated machines are used for testing.

Executing SQL statements in parallel with fault auto recovery

Parallel computation: After studying the previous two simple examples, it is time to study the coming third sample at the directory socketpro/samples/auto_recovery/(test_cplusplus|test_java|test_python|test_sharp). SocketPro is created from the bottom to support parallel computation. You can distribute multiple SQL statements onto different backend databases for processing concurrently. This feature is designed for improvement of application scalability as shown at the Figure 6.



```
 2  using System;
 3  using SocketProAdapter;
 4  using SocketProAdapter.ClientSide;
 5  class Program {
 6      static void Main(string[] args) {
 7          const int sessions_per_host = 2; const int cycles = 10000; string[] vHost = { "localhost", "192.168.2.172" };
 8          using (CSocketPool<CSqlite> sp = new CSocketPool<CSqlite>()) {
 9              sp.QueueName = "ar_sharp"; //set a local message queue to backup requests for auto fault recovery
10              CConnectionContext[,] ppCc = new CConnectionContext[1, vHost.Length * sessions_per_host]; //one thread enough
11              for (int n = 0; n < vHost.Length; ++n) {
12                  for (int j = 0; j < sessions_per_host; ++j) {
13                      ppCc[0, n * sessions_per_host + j] = new CConnectionContext(vHost[n], 20901, "AClientUserId", "Mypassword");
14                  }
15              }
16              bool ok = sp.StartSocketPool(ppCc);
17              if (!ok) {
18                  Console.WriteLine("There is no connection and press any key to close the application ......");
19                  Console.Read(); return;
20              }
21              string sql = "SELECT max(amount), min(amount), avg(amount) FROM payment";
22              Console.WriteLine("Input a filter for payment_id"); string filter = Console.ReadLine();
23              if (filter.Length > 0) sql += (" WHERE " + filter); var v = sp.AsyncHandlers;
24              foreach (var h in v) {
25                  ok = h.Open("sakila.db", (hsqlite, res, errMsg) => {
26                      if (res != 0) Console.WriteLine("Error code: {0}, error message: {1}", res, errMsg);
27                  });
28              }
29              int returned = 0; double dmax = 0.0, dmin = 0.0, davg = 0.0;
30              SocketProAdapter.UDB.CDBVariantArray row = new SocketProAdapter.UDB.CDBVariantArray();
31              CAsyncDBHandler.DExecuteResult er = (h, res, errMsg, affected, fail_ok, lastId) => {
32                  if (res != 0)
33                      Console.WriteLine("Error code: {0}, error message: {1}", res, errMsg);
34                  else {
35                      dmax += double.Parse(row[0].ToString());
36                      dmin += double.Parse(row[1].ToString());
37                      davg += double.Parse(row[2].ToString());
38                  }
39                  ++returned;
40              };
41              CAsyncDBHandler.DRows r = (h, vData) => {
42                  row.Clear(); row.AddRange(vData);
43              };
44              CSqlite sqlite = sp.SeekByQueue(); //get one handler for querying one record
45              ok = sqlite.Execute(sql, er, r); ok = sqlite.WaitAll();
46              Console.WriteLine("Result: max = {0}, min = {1}, avg = {2}", dmax, dmin, davg);
47              returned = 0; dmax = 0.0; dmin = 0.0; davg = 0.0;
48              Console.WriteLine("Going to get {0} queries for max, min and avg", cycles);
49              for (int n = 0; n < cycles; ++n) {
50                  sqlite = sp.SeekByQueue(); ok = sqlite.Execute(sql, er, r);
51              }
52              foreach (var h in v) {
53                  ok = h.WaitAll();
54              }
55              Console.WriteLine("Returned = {0}, max = {1}, min = {2}, avg = {3}", returned, dmax, dmin, davg);
56              Console.WriteLine("Press any key to close the application ......"); Console.Read();
57          }
58      }
59  }
```

Figure 6: Demonstration of SocketPro parallel computation and fault auto recovery features

As shown in Figure 6, we could start multiple non-blocking sockets to different machines (*localhost*, *192.168.2.172*), and each of the two database machines has two sockets connected at line 16. The code opens a default database *sakila.db* at line 25 for each of connections. First of all, the code executes one query *'SELECT max(amount), min(amount), avg(amount) FROM payment …'* at line 45 for one record. At last, the code sends the query 10,000 times onto the two machines for parallel processing at line 50. Each of records will be summed at lines 35 through 38 inside a Lambda expression as a callback for method *Execute*. It is noted that you can create multiple pools for different services hosted on different machines. As you can see, SocketPro socket pool can be used to significantly improve application scalability.

Auto fault recovery: SocketPro is able to open a file locally, and save all request data into it before sending these requests to a server through network. The file is called local message queue or client message queue. The idea is simple to back up all requests for automatic fault recovery. To use this feature, you have to set a local message queue name as shown at line 9. When we develop a real application, it is very common to write lots of code to deal with various communication errors properly. Actually, it is usually a challenge to software developers. SocketPro client message queue makes communication error handling very simple. Suppose the machine 192.168.2.172 is not accessible for one of whatever reasons like machine power off, unhandled exception, software/hardware maintenance and network unplug, and so on, the socket close event will be notified either immediately or sometime later. Once the socket pool finds a socket is closed, SocketPro will automatically merge all requests associated with the socket connection onto another socket which is not closed yet for processing.

To verify this feature, you can brutally down one of SQLite server (test_ssqlite) during executing the above query at line 50, and see if the final results are correct.

It is noted that UDAParts has applied this feature to all SocketPro SQL-stream services, asynchronous persistent message queue service and remote file exchange service to simplify your development.

Points of interest

SocketPro SQLite SQL-stream service provides all required basic client/server database features, but it does deliver the following unique features.

1. Continuous inline request/result batching and real-time SQL-stream processing for **the best network efficiency** especially on WAN
2. Bi-directional asynchronous data transferring between client and server, but all asynchronous requests can be converted into synchronous ones if necessary
3. **Superior performance and scalability** because of powerful SocketPro communication architecture
4. **Real-time cache for table update, insert and delete**. You can set a callback at client side for tracking table record add, delete and update events as shown at the sample project *test_cache* at the directory *socketpro/samples/module_sample/usqlite/test_cache*
5. All requests are **cancelable** by executing the method *Cancel* of class *CClientSocket* at client side
6. Both windows and Linux are supported
7. Simple development for all supported development languages

8. Both client and server components are **thread-safe**. They can be easily reused within your multi-threaded applications with much fewer thread related issues
9. All requests can be backed up at client side and resent to another server for processing in case a server is down for anyone of reasons – **fault auto recovery**