

## Brief introduction of SocketPro continuous SQL-stream sending and processing system (Part 2: MySQL)

### 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 for your complex needs. For reduction of learning complexity, I recommend you study the SQL-stream sample for SQLite ([sqlstream SQLite.pdf](#)) first before playing these MySQL sample projects as SQLite and MySQL samples share the same client API functions.

MySQL is currently the most popular open-source client-server distributed database management system. After studying MySQL source code, UDAParts has applied SocketPro SQL-stream technology onto MySQL and developed a server plug-in. UDAParts has compared SQL-stream technology with MySQL Connector/Net. Our performance study shows that SQL-stream technology can be up to one thousand times faster than MySQL Connector/Net on WAN.

### 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 *mysql* inside the directory *socketpro/stream\_sql*. You can see these samples are created from .NET, C/C++, Java and Python development environments. However, we use C# code (*socketpro/stream\_sql/mysql/test\_csahrp*) for client and C++ code (*socketpro/stream\_sql/mysql/smssql*) for server side development at this article for explanations.

In additon to the above samples, you can find performance study samples by use of MySQL sample database sakila at the directory *socketpro/stream\_sql/mysql/DBPerf*. The sub directory contains three performance study projects, cppperf, netperf and mysqlperf, which are written with C++/SocketPro SQL streaming, .NET/SocketPro SQL streaming and ADO.NET provider technologies, respectively.

Further, SocketPro MySQL server plugin supports data table update events (DELETE, INSERT and UPDATE) through triggers. You can use this feature to push update events of selected tables onto clients. The sample project is located at the directory *socketpro/stream\_sql/mysql/test\_cache*.

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*.

### Enabling MySQL prepare statements within server plug-in

Although MySql server plug-in does support general SQL statements, it does not support prepare statements at all. After digging into MySQL source code, UDAParts has figured out how to enable MySQL prepare statements within server plug-in. To add support of prepare statements within MySQL server plug-in, it is required to modify the four implementation files, *protocol\_callback.cc*, *protocol\_callback.h*, *protocol\_classic.cc* and *sql\_prepare.cc* before compiling the MySQL application *mysqld*. It must be noted that this is a **temporary** solution! Once MySQL server plug-in supports prepare statements in the future, UDAParts will use MySQL native implementation instead as expected. UDAParts has pre-compiled MySQL server application *mysqld* located at the directory *socketpro/stream\_sql/mysql/(mysql-5.7.18|mysql-5.7.19)* in case you are not familiar with C/C++ and want to skip the following paragraphs directly to the paragraph beginning with **Restart *mysqld*:**.

***protocol\_callback.cc* and *protocol\_callback.h*:** At the very beginning, add a forward class declaration (*class Item\_param;*) at the header of the file *protocol\_callback.h*. Next, it is required that the class *Protocol\_callback* must be added with two public methods (*send\_out\_parameters* and *init*) and one protected member *m\_thd*. These members have already been implemented within the two files at the directory *socketpro/stream\_sql/mysql/mysql-5.7.18*.

***protocol\_classic.cc*:** The method *Protocol\_classic::flush* has to be modified as shown in the below Figure 1 because the member *vio* could be null for MySQL server side plug-in.

```

1016     bool Protocol_classic::flush()
1017     {
1018     #ifndef EMBEDDED_LIBRARY
1019         bool error = 0;
1020         m_thd->get_stmt_da()->set_overwrite_status(true);
1021         if (m_thd->net.vio)
1022             error= net_flush(&m_thd->net);
1023         m_thd->get_stmt_da()->set_overwrite_status(false);
1024         return error;
1025     #else
1026         return 0;
1027     #endif
1028     }

```

Figure 1: The implementation of the method `Protocol_classic::flush` for enabling MySQL prepare statements within server plug-in

**sql\_prepare.cc:** At the very beginning, add one include for referring the file `protocol_callback.h`. Afterwards, find the method `mysql_stmt_execute`, and use the below code as shown at the below Figure 2 to replace this call `thd->set_protocol(&thd->protocol_binary);`.

```

2546     if (thd->protocol_binary.get_vio())
2547         thd->set_protocol(&thd->protocol_binary);
2548     else {
2549         ((Protocol_callback*) save_protocol)->init(thd);
2550     }

```

Figure 2: Modification of the method `mysql_stmt_execute` within the file `sql_prepare.cc`

Next, find the method `Prepared_statement::execute`, navigate to its end, and find this if statement (`if (error == 0 && this->lex->sql_command == SQLCOM_CALL)`). Use the code snippet as shown in the below Figure 3 to replace all its internal braced code.

```

3979     if (is_sql_prepare())
3980         thd->protocol_text.send_out_parameters(&this->lex->param_list);
3981     else if (thd->active_vio)
3982         thd->get_protocol_classic()->send_out_parameters(&this->lex->param_list);
3983     else {
3984         Protocol_callback *pc = (Protocol_callback*)thd->get_protocol();
3985         pc->send_out_parameters(&this->lex->param_list);
3986     }

```

Figure 3: Modification of the method *Prepared\_statement::execute* within the file *sql\_prepare.cc*

You can see the above code to check if active *vio* is available at run time. If it is not available for server plug-in prepared statements, we use callback protocol instead.

**Restart *mysqld*:** Before starting the newly compiled *mysqld* application on your system, you should **explicitly** set *plugin\_dir* to the directory containing MySQL plugin libraries (for example, *plugin\_dir = /usr/lib/mysql/plugin*) within the section *mysqld* of MySQL configuration file. Also, it is better to increase *thread\_stack* to 512K (*thread\_stack = 512K*) by changing the MySQL configuration file. Next, copy MySQL SQL-stream plugin *libsmysql.so* (smysql.dll on window platforms) into the MySQL plugin directory. After resetting all these configuration settings and replacing *mysqld* with new one, restart MySQL service or daemon.

### Register SocketPro MySQL SQL-streaming plugin and its configuration database

As described at [this site](#), register MySQL SQL-stream plugin by calling statement ***INSTALL PLUGIN UDAParts\_SQL\_Streaming SONAME 'libsmysql.so'*** from the application *mysql*. If successful, you should see a new database *sp\_streaming\_db* created as shown in the below Figure 4.

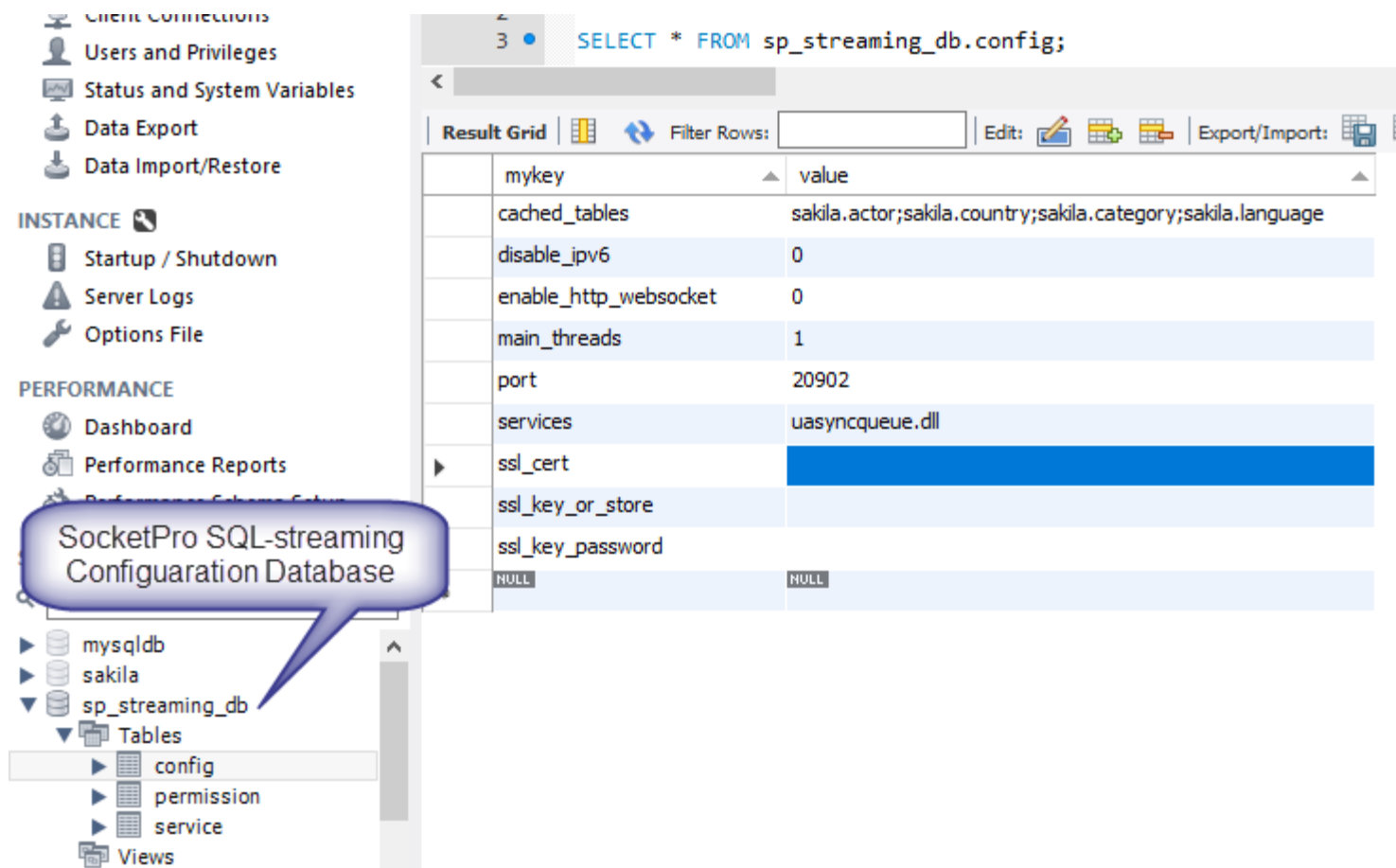
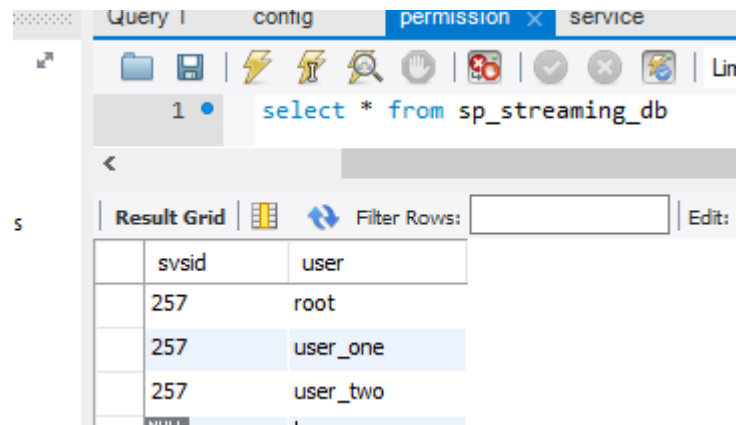


Figure 4: SocketPro SQL-streaming configuration database `sp_streaming_db` and table `config`

The configuration database has three simple tables, `config`, `service` and `permission` as shown in the above Figure 4. It is expected that SocketPro MySQL SQL-streaming plugin supports industrial security standard SSL3/TLSv1.x to secure communication between client and server. By default, a SocketPro client can use either IP v4 or v6 to access MySQL database at port number 20902. Pay attention to the record `cached_tables`. If you set its value properly, all connected SocketPro clients can see data changes within these tables (for example, table `actor`, `country`, `category` and `language` within database `sakila`) in real time. Referring the sample `test_cache` at directory `socketpro/stream_sql/mysql`, you can use the real-time cache feature to improve your middle tier performance and scalability by reducing data trips between middle tier and database.

One SocketPro server is capable to support many services at the same time by use of one TCP port. If you like, you can enable websocket from SocketPro MySQL SQL-streaming plugin by setting value to '1' for record *enable\_http\_websocket*. Further, you can also embed other services by setting value properly of record *services* as shown in the above Figure 4. Once changing any one or more values within the table *config*, you should restart MySQL. Otherwise, the changes will not function correctly.

In regards to the table *permission*, SocketPro MySQL SQL-streaming technology uses its records to authenticate clients for embedded services as shown in the following Figure 5. MySQL SQL-streaming plugin uses the two tables *mysql.user* and *sp\_streaming\_db.permission* to authenticate all clients for all services. However, its SQL-streaming service does not use records within the table *sp\_streaming\_db.permission* for authentication.



svsid	user
257	root
257	user_one
257	user_two

Figure 5: Three users (*root*, *user\_one* and *user\_two*) allowed for SocketPro asynchronous persistent message queue service (service id=257)

Under most cases, you are not required to touch the table *service*.

## 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 with this sample at client side as shown in the below Figure 6.

```

20 static void Main(string[] args)
21 {
22     Console.WriteLine("Remote host: "); string host = Console.ReadLine();
23     CConnectionContext cc = new CConnectionContext(host, 20902, "root", "Smash123");
24     using (CSocketPool<CMySQL> spMySQL = new CSocketPool<CMySQL>()) {
25         if (!spMySQL.StartSocketPool(cc, 1, 1)) {
26             Console.WriteLine("Failed in connecting to remote async mysql server and press any key to close the application .....");
27             Console.Read(); return;
28         }
29         CMySQL mysql = spMySQL.Seek();
30         bool ok = mysql.Open("", dr);
31         List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>> ra = new List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>>();
32         CMySQL.DRows r = (handler, rowData) => { //rowset data come here
33             int last = ra.Count - 1; KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = ra[last];
34             item.Value.AddRange(rowData);
35         };
36         CMySQL.DRowsetHeader rh = (handler) => { //rowset header comes here
37             KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = new KeyValuePair<CDBColumnInfoArray, CDBVariantArray>(handler.ColumnInfo, new CDBVariantArray());
38             ra.Add(item);
39         };
40         TestCreateTables(mysql);
41         ok = mysql.Execute("delete from employee;delete from company", er);
42         TestPreparedStatements(mysql);
43         InsertBLOBByPreparedStatement(mysql);
44         ok = mysql.Execute("SELECT * from company;select * from employee;select curtime()", er, r, rh);
45         CDBVariantArray vPData = new CDBVariantArray();
46         //first set
47         vPData.Add(1); vPData.Add(1.4); vPData.Add(0);
48         //second set
49         vPData.Add(2); vPData.Add(2.5); vPData.Add(0);
50         TestStoredProcedure(mysql, ra, vPData);
51         ok = mysql.WaitAll();
52         Console.WriteLine(); Console.WriteLine("There are {0} output data returned", mysql.Outputs * 2);
53         int index = 0; Console.WriteLine(); Console.WriteLine("++++ Start rowsets +++");
54         foreach (KeyValuePair<CDBColumnInfoArray, CDBVariantArray> it in ra) {
55             Console.WriteLine("Statement index = {0}", index);
56             if (it.Key.Count > 0)
57                 Console.WriteLine("rowset with columns = {0}, records = {1}.", it.Key.Count, it.Value.Count / it.Key.Count);
58             else
59                 Console.WriteLine("no rowset received.");
60             ++index;
61         }
62         Console.WriteLine("++++ End rowsets +++"); Console.WriteLine();
63         Console.WriteLine("Press any key to close the application .....");Console.Read();
64     }
65 }

```

Figure 6: Main function for demonstration of SocketPro MySQL SQL-stream system at client side

**Starting one socket pool:** The above Figure 6 starts one socket pool which only has one worker thread that only hosts one non-blocking socket at line 25 for demonstration clarity by use of one instance of connection context. However, it is noted that you can create multiple pools within one client application if necessary. Afterwards, we get one asynchronous MySQL handler at line 29.

**Opening database:** We can send a request for opening a MySQL server database at line 30. If the first input is an empty or null string as shown at this example, we are opening one default database for a connected user, for example. If you like to open a specified database, you can simply give a non-empty valid database name 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 line 30. 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 31.

**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 can easily stream all SQL statements as well as others as shown at lines 40 through 50. All SocketPro SQL-stream services support this unique feature for the best network efficiency, which will significantly improve 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 in the [previous article](#).

**Waiting until all processed:** Since SocketPro only supports asynchronous data transferring, SocketPro must have a way to wait until all requests and returning results are sent, processed and returned. SocketPro does come with this method WaitAll at client side to serve this purpose as shown at line 51. If you like, you can use this method to convert all asynchronous requests into synchronous ones.

### **TestCreateTables, TestPreparedStatements and InsertBLOBByPreparedStatement**

As shown at lines 40, 42 and 43 in the above Figure 6, we could ignore them because they are truly the same as ones in the [previous article](#). Let's focus executing MySQL stored procedures with input-output and output parameters.

### **TestStoredProcedure**

MySQL fully supports stored procedures. SocketPro SQL-stream technology does too. Further, SocketPro SQL-stream technology supports executing multiple sets of MySQL stored procedures with input-output and output parameters in one call as shown at lines 45 through 50 in the above Figure 6. The below Figure 7 shows how to call a MySQL stored procedure which may have input, input/output and output parameters and return multiple sets of records.



```

157 static void TestStoredProcedure(CMysql mysql, List<KeyValuePair<CDBColumnInfoArray, CDBVariantArray>> ra, CDBVariantArray vPData) {
158     bool ok = mysql.Prepare("call sp_TestProc(?,?,?)", dr);
159     CMysql.DRows r = (handler, rowData) => { //rowset data come here
160         int last = ra.Count - 1;
161         KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = ra[last];
162         item.Value.AddRange(rowData);
163     };
164     CMysql.DRowsetHeader rh = (handler) => { //rowset header comes here
165         KeyValuePair<CDBColumnInfoArray, CDBVariantArray> item = new KeyValuePair<CDBColumnInfoArray, CDBVariantArray>(handler.ColumnInfo, new CDBVariantArray());
166         ra.Add(item);
167     };
168     ok = mysql.Execute(vPData, er, r, rh);
169 }

```

Figure 7: Call MySQL stored procedure which returns multiple sets of records and output parameters

It is very simple to call stored procedure through SocketPro SQL-stream technology at line 168 as shown in the above Figure 7. It is noted that all output parameter data will be directly copied into the passing parameter data array *vPData*. The callback *rh* at lines 164 through 167 is called when record set meta data comes if available. Whenever an array of record data comes, the callback *r* at lines 159 through 163 will be called. You can populate all queried meta and record data into an arbitrary container like *ra*, for example, from the two callbacks.

## Performance study

SocketPro SQL-stream technology has excellent performance in database data accessing for both query and update. You can see two MySQL performance test projects (cppperf and netperf) available at [socketpro/stream\\_sql/mysql/DBPerf/](http://socketpro/stream_sql/mysql/DBPerf/). The first sample is written by C++ and the other by C#. A sample project mysqlperf writtern from C# is provided for you to compare SocketPro SQL-stream technology with MySQL .NET provider in performance.

See the performance study data of the below Figure 8, which is obtained from three cheap Google cloud virtual machines with solid state drive for free evaluation. All data are times required in millisecond for executing 10,000 queries and 50,000 inserts. The performance study is also focused on influence of network latency on MySQL 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	
5670	1540	17600	2000	SocketPro Streaming + Async
14300	3905	357000	349000	SocketPro + Sync
15200	3850	354000	348000	MySQL.NET Provider
SQL-INSERT	1170	SQL-INSERT	2840	SocketPro Streaming/Async
	10400		1726000	MySQL.NET Provider
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 8: MySQL 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 6,500 (10,000/1.54) times per second and socket connection. For inserting records, you can easily get the speed like 43,000 (50,000/1.17) inserts per second for MySQL on local area network (LAN, cross-machine, 0.2 ms/2.0 Gbps). On LAN, SocketPro streaming could improve 150% in performance over traditional non-streaming approach (SocketPro + Sync) for query. For SQL inserts, the improvement would be more than seven times (10,400/1,170 = 8.9). SocketPro streaming and in-line batching features make network efficiency superiorly high, which leads to the significantly improvement in comparison to existing MySQL socket communication approach.

Let's consider wide area network (WAN, cross-region, 34 ms/40 Mbps). SocketPro SQL streaming query speed could be 5,000 (10,000/2.00) times per second and socket connection. For inserting records, the speed could easily be 17,600 records (50,000/2.84) per second. Contrarily, the query speed will be as low as 30 queries per second on WAN if a client uses traditional communication way (SocketPro+Sync/MySQL.NET Provider) for database accessing because of high latency as shown in the above Figure 8. SocketPro SQL streaming can be more than 170 (349000/2000 = 174.5) times in query faster than non-streaming technology, assuming database backend processing time is ignorable on high latency WAN (cross-region). If we consider SQL inserts, the improvement could be over 600 times (1,726,000/2840 = 607).

After analyzing the performance data in Figure 8, you will find SocketPro streaming technology is truly great for speeding up not only local but also remoting database accessing. Second, performance data for WAN would be much better if the test WAN have better network bandwidth. Further, SocketPro supports inline compression but this test study doesn't use it. 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 will be better if dedicated machines are used for testing.

### **Executing SQLs 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 9.

```

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<CMySQL> sp = new CSocketPool<CMySQL>()) {
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], 28902, "root", "Smash123");
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", (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            CMySQL mysql = sp.SeekByQueue(); //get one handler for querying one record
45            ok = mysql.Execute(sql, er, r); ok = mysql.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                mysql = sp.SeekByQueue(); ok = mysql.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 9: Demonstration of SocketPro parallel computation and fault auto recovery features

As shown in Figure 9, 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* at line 25 for each of connections. First, 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 can 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 must 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. In fact, 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 unplugged, 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 MySQL servers 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

At last, SocketPro MySQL SQL-stream plugin doesn't support cursors at all, but it does provide all required basic client/server database features. Further, the SQL-stream plugin does have 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 required
3. **Superior performance and scalability** because of powerful SocketPro communication architecture
4. **Real-time cache for table update, insert and delete** as shown at the sample project *test\_cache* at the directory *socketpro/stream\_sql/mysql/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 automatically for processing in case a server is down for anyone of reasons – **fault auto recovery**