**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 protocols that require 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 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 popular opened source databases such as Sqlite and MySQL as well as others through ODBC drivers to support continuous SQL-stream sending and processing. At the last, these components for databases are **totally free forever** to the public. For reduction of learning complexity, I recommend you study the SQL-stream sample for Sqlite first before playing these sample projects as both of them 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 hundred times faster than MySQL Connector/Net.

**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. 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.

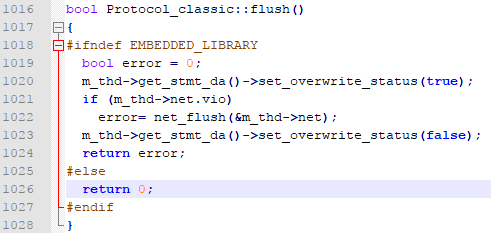
In regards to SocketPro communication framework, you may also refer to its development guide documentation at socketpro/doc/**SocketPro development guide.pdf.**

**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 is 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.**

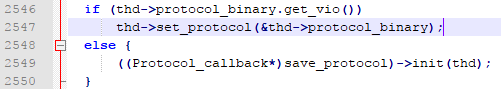
***protocol\_callback.cc and protocol\_callback.h:* At the very beginning, add a forward class declaration (*class Item\_param;*) at the header of the file *protocol\_callback.h*. Next, it is required that the class *Protocol\_callback* has to be added with two public methods (*send\_out\_parameters* 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.**



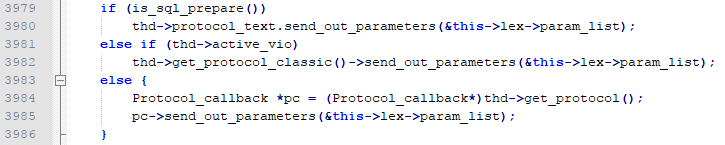
***Figure 1: The implementation of the method Protocol\_classic::flush for enabling MySQL prepare statements within server plug-in***

***sql\_prepare.cc:* At the very beginning, add one include for referring the file *protocol\_callback.h*. Afterwards, find the method *mysqld\_stmt\_execute*, and use the below code as shown at the below Figure 2 to replace this call *thd->set\_protocol(&thd->protocol\_binary);*.**



***Figure 2: Modification of the method mysqld\_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 below code snippet as shown in the below Figure 3 to replace all internal braced code.**

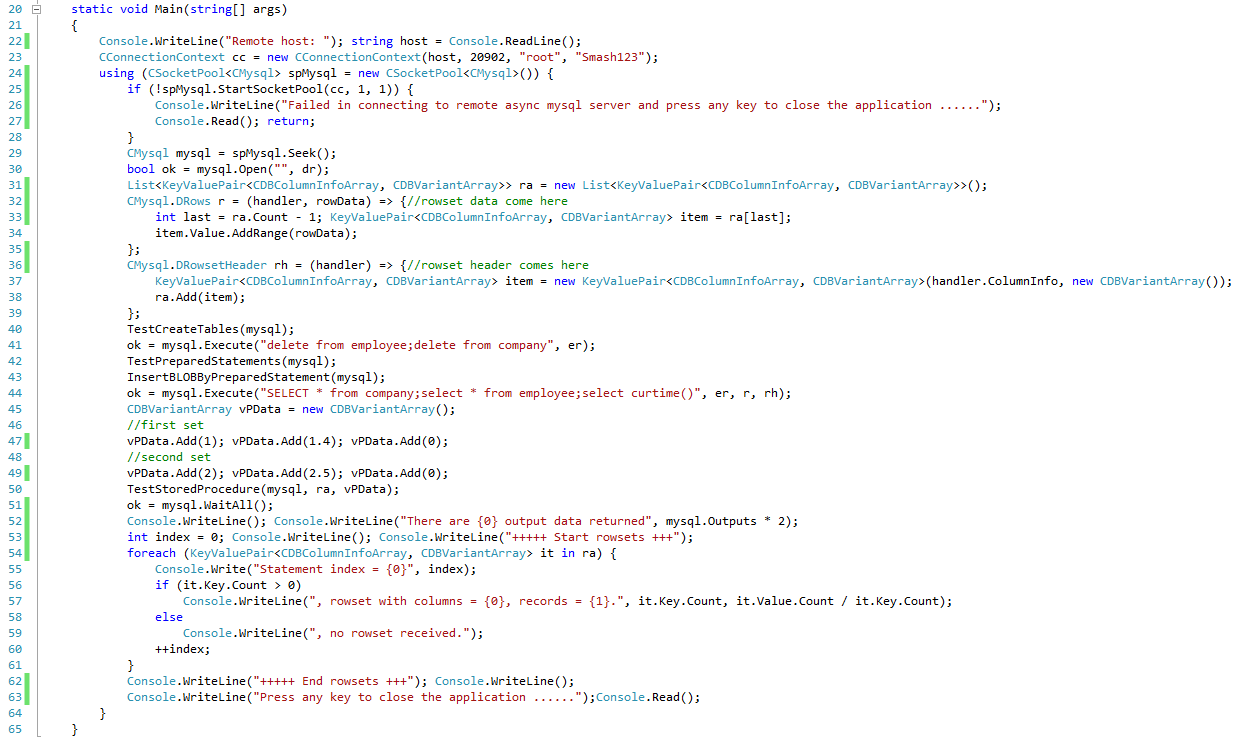


***Figure 3: Modification of the method Prepared\_statement::execute within the file sql\_prepare.cc***

**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.**

**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 is made of one or more threads, and each of threads hosts one or more non-blocking sockets at client side. However, 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 4.



*Figure 4: Main function for demonstration of SocketPro MySQL 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 25 for demonstration clarity by use of one instance of connection context. However, you can create multiple pools within one client application if necessary. Afterwards, we get one asynchronous MySQL handler at line 29.

***Opening database:*** We are able to 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 are able to stream all SQL statements as well as others as shown at lines 40 through 50. 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 in the previous article.

***Waiting until all processed:*** Since SocketPro only supports asynchronous data transferring, SocketPro has to provide a way to wait until all requests and returning results are sent, processed and returned. SocketPro comes one particular 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**

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



*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.



*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.



*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. 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.

Our performance study shows that it is easy to get query executed at the speed of 12,000 times per second and socket connection. For insert, you can easily get the speed like 50,000 inserts per second for Sqlite.