SQL networking protocol

1. Requirements.

1.1 ***Text-based***. Should be possible to use simple command-line applications like "terminal" to run test sessions against SQL server.

1.2 ***HTTP-like***. Each protocol command starts with a set of headers, optionally followed by text or binary data associated with the commands. Headers are separated from the body by double CRLF (see below). Headers should be limited to ASCII characters. Anything out of that range should be encoded with Base-64 and marked accordingly.

1.3 ***Synchronous.*** Initially the protocol will be synchronous – requests are executed in the order received and responses are sent in the order executed. Normally, a client posts a command and then waits for a response. However, clients should be able to post multiple commands and then wait for multiple responses. Should be easily convertible to asynchronous mode with requests processed and responses sent out of order; with as little changes to protocol as possible.

1.4 ***No tunneling.*** Initially the protocol will not support tunneling - there will be only one client session per connection. Should be easily convertible to tunneling mode with as little changes to protocol as possible.

1.5 ***Two-way communication.*** Must support server-initiated requests, such as but not limited to asynchronous execution status, logging, status updates, etc… Current version does not support server-initiated requests.

2. Specification.

2.1 White spaces, caret returns and line feeds are considered formatting symbols and can not be used inside protocol tokens. The following symbols are defined to display protocol usage examples:

- SP any combination of white space characters (space and tab)
- CR caret return
- LF line feed
- TOKEN Capitalized words are actual protocol tokens as they will be used in protocol.
- *Italic* Words in italic contain description of what they represent.

2.2 Each protocol command has a set of required attributes that are passed from the client to the server and back. Some of them are ***Command ID, Session ID, Authentication token, etc…***

- SQL server will return Command ID along with the response so that the client can identify the response. Client's Command IDs must be easily differentiable from Server initiated Command IDs. Other than that, there are no major restrictions on Command ID format.
- Session ID can be used to handle tunneling of multiple sessions within one socket connection.

- Protocol header values may be Base-64 encoded. In this case, header name ‘XXX’ changes to ‘XXX-BASE64’. This is applicable to both – request and response headers.

2.3 Server response must always contain result code. Error messages might also be provided in case of error.

2.4 Commands. The following list is incomplete at the moment, but is enough to produce a working version of ODBC and JDBC drivers with approximately 80% of the features implemented.

- LOGIN
- PREPARE-STATEMENT
- EXECUTE-STATEMENT
- CLOSE-STATEMENT
- FETCH-RESULT
- BULK-MODIFY
- LOGOUT
- QUIT

3. Command description.

3.1 A protocol session starts with **LOGIN** command, which can optionally perform strong authentication. To avoid unnecessary network roundtrips, information about client’s environment should be included in LOGIN command, such as client’s version, OS name and version, language preferences, preferred data encoding, etc… In response to accepted LOGIN, the server will return metadata information about itself, current database, and current connection. Here is an hypothetical example:

Command-ID SP LOGIN CRLF
USER-NAME-BASE64 SP : SP *Base-64-Encoded-User-Name* CRLF
USER-PASSWORD-BASE64 SP: SP *Base-64-Encoded-User-Password* CRLF
REPLY-WITH-BASE64-TEXT: [*Y* | *N*]
VERSION SP : SP *Version-String* CRLF
OS-NAME SP : SP *OS-Name* CRLF
OS-VERSION SP : SP *OS-Version* CRLF
CRLF

There should also be plain USER-NAME and USER-PASSWORD headers to enable command-line terminal testing.

User password can be MD5 hashed, in which case the corresponding header is named USER-PASSWORD-MD5.

Here is a real-life example of the current LOGIN command:
001 LOGIN CRLF
USER-NAME: UserName CRLF
USER-PASSWORD: UserPassword CRLF

REPLY-WITH-BASE64-TEXT: Y
PROTOCOL-VERSION: 0.1a CRLF
CRLF

3.2 A protocol session ends with **LOGOUT** command which has a simple form. Here it is:

Command-ID LOGOUT CRLF
CRLF

LOGOUT command does not close the socket. A new session may be started with new LOGIN command.

3.3

Commands are executed using **EXECUTE-STATEMENT** protocol command. The specification given below is not final as it contains several workarounds to deal with some of the missing pieces of functionality in either XToolBox or SQL server itself. The command looks like this:

Command-ID SP EXECUTE-STATEMENT CRLF
STATEMENT SP : SP *Statement text* CRLF
OUTPUT-MODE SP: SP Debug *or* Release CRLF
PREFERRED-IMAGE-TYPES : *A space-separated list of target image formats*
FIRST-PAGE-SIZE : *Number of records returned with response (default is 100)* CRLF
PARAMETER-TYPES: SP *A space-separated list of XTollBox types* SP CRLF
FULL-ERROR-STACK: SP Y *or* N SP CRLF
CRLF
Parameter values in binary mode as described below.

For example:
003 Execute-Statement
Statement: select * from mytable
Output-Mode: release

A response looks like this:

Command-ID SP *Status(OK or Error)* CRLF
STATEMENT-ID SP : SP *A server-assigned statement id CRLF*
COMMAND-COUNT SP : SP *Amount of commands in the statement* CRLF
RESULT-TYPE SP : SP *Result-Set or Update-Count* CRLF
COLUMN-COUNT SP : SP *Column count in the result set* CRLF
ROW-COUNT SP : SP *Row count in the result set* CRLF
COLUMN-TYPES SP : SP *A space-separated list of XTollBox types* CRLF
COLUMN-ALIASES SP : SP *Space-separated aliases in optional [and]* CRLF
COLUMN-UPDATEABILITY SP : SP *Space-separated* Y *or* N CRLF

ROW-COUNT-SENT SP : SP *An amount of rows sent in this initial response* CRLF
CRLF

For example:

003 OK
Statement-ID:2
Command-Count:1
Result-Type:Result-Set
Column-Count:3
Row-Count:16
Column-Types:VK_LONG VK_STRING VK_STRING
Column-Aliases:[EMPNO] [ENAME] [JOB]
Column-Updateability: Y Y Y
Row-Count-Sent:16

1?1SMITH1CLERK1?1?
?w1???S??@1ק=
???11?
?w11K1ALLENSALESMAN11?
?w1??K7 ?@1ק=
???11?
?w11a1WARDSALESMAN11?
?w1-???'??@1ק=
???11?
?w11?1JONES1MANAGER1?1?
?w1?n?>?@1ק=

PREFERRED-IMAGE-TYPES is an optional header that allows to choose formats in which a server will send images. For example, if this header's value is *.gif .jpeg .png* then a server will send gif, jpeg and png images without any modification, but images of other types will be converted to gif. This is a way for a client to say "Here are the image formats that I understand and if the server finds an image of another type, please convert it to the first type in my list".

After the double CRLF goes the binary data in release mode or tab-delimited data in debug mode of the first command within the statement. The format of binary output is discussed below.

Rows are output from first to last. If result-set is updateable then each row starts with a 4 byte ID of a record, otherwise record ID is omitted. Column values within a row are output from left to right. Each value is preceded by a status byte. If the value was successfully calculated then the status byte is character '1' for non-null values and

character ‘0’ for null values. If there was an error during value calculation then the status byte is character ‘2’.

If the value is null then immediately after ‘0’ status byte goes the data for the next value. If the status byte is ‘2’ then the status byte is followed by the VLong8 error code and no more data is sent for this command (TODO: Should also send the error message). The ‘1’ status byte is followed by the binary representation of one of the VValueSingle subclass instances. The exact type of the instance can be determined using “Column-Types” header. Binary data for VValueSingle should be read using ReadFromStream() method with the appropriate sub-class of VStream whenever possible. The only exception from this case is binary values for columns of type VK_IMAGE. VK_IMAGE values are streamed as VBlob and thus should be read as such. Binary data should never be read manually unless XToolBox streaming classes are not available.

Binary data ends with the last value in the last row. There is no command terminator of any kind.

3.4 EXECUTE-STATEMENT protocol command returns only first few records of the large result set. **FETCH-RESULT** command allows retrieving arbitrary sub-range of result-set. Here is the definition of the fetch command:

Command-ID SP FETCH-RESULT CRLF
STATEMENT-ID SP : SP *Statement ID* CRLF
COMMAND-INDEX SP : SP *Command index* CRLF
FIRST-ROW-INDEX SP : SP *Index of the first row to fetch* CRLF
LAST-ROW-INDEX SP : SP *Index of the last row to fetch* CRLF
OUTPUT-MODE SP : SP RELEASE CRLF
CRLF *Binary data*
Note, that all indexes are 0-based, if not noted otherwise. First-Last row interval is inclusive on both sides.
For example:

123 FETCH-RESULT
STATEMENT-ID : 23
COMMAND-INDEX : 3
FIRST-ROW-INDEX : 200
LAST-ROW-INDEX : 299
OUTPUT-MODE : RELEASE

Response to FETCH-RESULT command is defined as following:

Command-ID SP *Status(OK or Error)* CRLF
CRLF
Binary response exactly as in EXECUTE-STATEMENT command response.

3.5 **BULK-MODIFY** command allows making modifications to individual records and individual values within those records in a given result set. Here is how it is

defined:

Command-ID SP BULK-MODIFY CRLF
STATEMENT-ID SP : SP *Statement ID* CRLF
COMMAND-INDEX SP : SP *Command index* CRLF
ROW-COUNT SP : SP *Amount of records to be modified* CRLF
CRLF *Binary data*

Modifications are described on a row-by-row basis. First goes modification type – 1 byte; its values are '1' for update, '2' for insert and '3' for delete. Subsequent data differs depending on the modification byte.

For updates: numeric long value (VK_LONG streamed) – ID of the record to be updated; numeric word value (VK_WORD streamed) - amount of updated fields in a given record. Then the list of actual values in the following format – first, updated column's 0-based index (VK_WORD streamed), then "nullness" byte and then the actual VValue streamed, if it is not null.

For inserts: numeric word value (VK_WORD streamed) - amount of inserted fields in the new record (not all fields may be specified). Then the list of new values in the following format – first, inserted column's 0-based index (VK_WORD streamed), then "nullness" byte and then the actual VValue streamed, if it is not null.

For deletes: numeric long value (VK_LONG streamed) – ID of the record to be deleted.

Successful reply looks like this:

Command-ID SP OK CRLF
CRLF
Binary data

Binary data consists of a list of streamed VLong without preceding 'nullness' byte. Each VLong specifies new ID of the corresponding new record. There are as many VLong values as there are new records.

3.6 **PREPARE-STATEMENT** command parses, validates and provides detailed information about results of SQL commands within a given statement without actually executing any of them. Here is a definition of this command:

Command-ID SP PREPARE-STATEMENT CRLF
STATEMENT SP : SP *Statement text* CRLF
PARAMETER-TYPES: SP *A space-separated list of XTollBox types* SP CRLF
CRLF
Parameters in binary form are sent exactly as in EXECUTE-STATEMENT command.

PARAMETER-TYPES header is optional. It is required only if there are parameters

in the statement.
For example:

00x PREPARE-STATEMENT
STATEMENT: select * from mytable; insert into mytable values (1, ‘one’);

A response looks like this:

Command-ID SP *Status(OK or Error)* CRLF
COMMAND-COUNT SP : SP *Amount of commands in the statement* CRLF
RESULT-TYPE SP : SP *Space-separated Result-Set or Update-Count* CRLF
COLUMN-COUNT SP : SP *Space-separated column count in the result set* CRLF
COLUMN-TYPES SP : SP *A space-separated list of XTollBox types* CRLF
COLUMN-ALIASES SP : SP *Space-separated aliases in optional [and]* CRLF
COLUMN-UPDATEABILITY SP : SP *Space-separated* Y *or* N CRLF
CRLF

For example:

00x OK
COMMAND-COUNT: 2
RESULT-TYPE: Result-Set Update-Count
COLUMN-COUNT: 2 1
COLUMN-TYPES: VK_LONG, VK_STRING, VK_LONG8
COLUMN-ALIASES: [ID] [NAME] [Update Count]
COLUMN-UPDATEABILITY: Y Y N

3.7 **CLOSE-STATEMENT** command is used to close one of the current opened statements within a current client connection. Here is the definition

Command-ID SP CLOSE-STATEMENT CRLF
STATEMENT-ID SP : SP *Statement ID* CRLF
CRLF

The OK response is defined like this:

Command-ID SP OK CRLF
CRLF

The error response looks like the following:

Command-ID SP ERROR CRLF
ERROR-CODE : *Error Code*
ERROR-DESCRIPTION : *Error description* CRLF

CRLF

3.8 **QUIT** command closes the actual socket connection. It must be called when the user is not logged in (after LOGOUT command or before LOGIN).

It looks like this:
Command-ID QUIT CRLF
CRLF

The OK response is defined like this:

Command-ID SP OK CRLF
CRLF

The error response looks like the following:

Command-ID SP ERROR CRLF
ERROR-CODE : *Error Code*
ERROR-DESCRIPTION : *Error description* CRLF
CRLF

3.9 Continue…

4. Binary format for values:

Boolean	Signed short int (2 bytes); 0 for FALSE, non-0 for TRUE
Byte	Signed short int (2 bytes); Numeric value itself – C++
Word	Signed short int (2 bytes); Numeric value itself – C++
Long	Signed long int (4 bytes); Numeric value itself – C++
Long8	Signed long long int (8 bytes); Numeric value itself – C++
Real	double (8 bytes); Numeric value itself – C++
Float	Custom, non-primitive (varying length); 4 bytes (exponent) + 1 byte (sign) + 4 bytes (data length)
Timestamp	Custom, non-primitive (8 bytes); 2 bytes (year) + 1 byte (month) + 1 byte (day) + 4 bytes (milliseconds)
Duration	Signed long long int (8 bytes); Numeric value itself (milliseconds) – C++
String	Custom, non-primitive (varying length); 4 bytes (negative value of length) + UniCode value in UTF-16 (2 bytes per char)
Blob	Custom, non-primitive (varying length); 4 bytes (length) + binary data
Image	Custom, non-primitive (varying length); raw image bytes, image type depends on the value of PREFERRED-IMAGE-TYPES header of the EXECUTE-STATEMENT command.

Almost any language will let you build a value of primitive type from a binary stream or a binary array as is. However, if such functionality is not available then you may need to

reconstruct values yourself – byte by byte. Note that some byte swapping may be necessary depending on the implementation language, networking libraries and the OS.

5. Notes and TODOs:

4.1 How will the server accept and store large chunks of data? All in one go? How to store and transfer it efficiently?