Borrui Data

RapidsDB Wireline Protocol Specification

Protocol Version 2

Table of Contents

1. Introduction ... 3
2. Protocol Overview ... 3
3. Supported Languages and Dependencies ... 4
4. Protocol Versioning ... 4
5. Message Flows ... 5
   5.1 Introduction and Semantics ... 5
   5.2 Compatibility dialog ... 6
   5.3 Connection dialog ... 7
   5.4 Query Execute dialog ... 10
   5.5 Termination dialog ... 1
   5.6 Other Dialogs ... 2
6. Messages ... 2
   6.1 Nomenclature ... 2
   6.2 Compiling Message Definitions ... 2
   6.3 Message Header ... 3
   6.4 Message Types ... 4
   6.5 Thrift Files ... 5
7. Sessions ... 7
8. Limits And Constraints ... 7
9. Change History ... 8
   9.1 Version 1 ... 8
   9.2 Version 2 ... 8
10. Future Enhancements ... 8
   10.1 Asynchronous dialogs ... 8
      10.1.1 Server Shutdown Notification dialog ... 9
      10.1.2 Client Disconnect Notification dialog ... 9
   10.2 Sideband dialogs ... 10
      10.2.1 Query Cancellation dialog ... 10
      10.2.2 Query Progress dialog ... 11
      10.2.3 Session Monitoring dialog ... 12

 © Borrui Data Technology Co. Ltd 2017

1. Introduction

The RapidsDB Wireline Protocol is a platform-independent protocol that can be used for programmatically accessing RapidsDB from many different programming languages. It is a specification of the messages that flow between the client and server and sequencing of those messages. With this information, readers could build clients and APIs to interface with RapidsDB in almost any programming language.

This document describes this protocol.

2. Protocol Overview

The RapidsDB Wireline Protocol is a message-based protocol that uses TCP/IP sockets for transport between clients and the server. Unlike the rapids-shell, which connects to the DQC node, wireline protocol clients can connect to any DQC or DQE node in the cluster, since all nodes are functionally homogeneous. The node that a client connects to is then responsible for the parsing, planning and execution of the query, as well as collating responses from all nodes and sending them to the client. The port number of the server that the client is connecting to is designated by the "`clientPort`" key/value in the cluster.config file. This is typically set to 4333 unless it is overridden in this file.

Each TCP/IP connection to the server establishes a new session. Each session is capable of executing only one request from a client at a time. If multiple concurrent requests need to be executed then multiple connections must be used.

The wireline protocol follows a model of trying to keep the most common operations as simple as possible, at the expense of additional complexity for the less common operations. This makes it easier to implement the critical parts of the protocol, both for the client and server, and makes it much less error-prone. As such, the primary interaction between the client and RapidsDB is a synchronized request-response model: the client issues a request to the server and then waits for the server to send back one or more responses. We refer to this sequence within this document as a **dialog**. Occasionally a dialog between and client and server may contain a sub-dialog within it, but dialogs still follow the request-response pattern.

The Wireline Protocol can also support asynchronous messages sent from the server to the client. These types of messages are usually sent to inform the client of an event that is outside of any current request being processed. An example of this would be a message sent from the server to all clients to indicate that the server is shutting down. This is currently not implemented in the server, but the Wireline Protocol has allowed for it.

In the future, users will be able to track the progress of requests or cancel requests that are currently being executed. Due to the synchronous request-response model, and the desire to keep the most common aspects of the protocol as simple as possible, sending a second request on a session is not allowed until the first one is completed. Therefore the RapidsDB Wireline Protocol allows for this with out-of-band queries – i.e., allowing a request on a different session to track the progress or cancel a request on the original session, with appropriate authorization of course. This is currently not implemented in RapidsDB, but again the Wireline Protocol has allowed for it to be implemented in the future.

Since a request from the client can result in the server sending large amounts of data back to the client, it is important to understand flow control in the protocol. When executing a client's request,

© Borrui Data Technology Co. Ltd 2017

the server will try to send as much data as it has available to the client as soon as it can. The server typically batches the results into messages containing up to 10,000 rows to maximize throughput. The server will try to send each of these messages to the client when they are filled. If the client is not reading the messages as fast as the server is producing them (e.g., the client is processing the data returned) then the server will eventually get blocked from sending more data as the TCP/IP send window closes. As the client reads more data, the TCP/IP window will open and this will allow the server to send further messages.

Messages sent between clients and servers contain a fixed size header and a payload. The header identifies the type of payload and the overall size of the message. The payload itself is the serialization of a message described in an Interface Definition Language (IDL) – in this case a Thrift file. An IDL framework was chosen for the Wireline Protocol since it alleviates protocol implementers from writing error-prone and time consuming code to read and write messages correctly. Instead, the Wireline Protocol defines Thrift messages based on their content and these Thrift messages get compiled to code for the target programming language, allowing the protocol implementer to focus on message sequencing, knowing that the message contents will be formatted correctly. Thrift is widely available across all the popular programming languages.

3. Supported Languages and Dependencies

The RapidsDB Wireline Protocol is available as of RapidsDB version 3.0, which also supplies a JDBC driver based upon this protocol.

Clients wishing to develop their own implementation of this protocol require:

1. Access to TCP/IP sockets.
2. The ability to compile Thrift messages into code for their target language. As of version 0.10.1, Thrift supports the following programming languages:

AS3	Cocoa	C	C++	C#
D	Dart	Delphi	Erlang	Go
Haxe	Haskell	Java	Javascript	Lua
Ocaml	Perl	PHP	Python	Ruby
Smalltalk	Swift			

4. Protocol Versioning

The RapidsDB Wireline Protocol has a provision in it for versioning the protocol itself. Different versions of the protocol may exchange different messages, or they may exchange messages differently. RapidsDB has been built to be able to accept clients communicating over older protocol versions so that existing clients and their drivers may not need to be upgraded when RapidsDB is upgraded.

When a client connects to RapidsDB over the Wireline Protocol, the first interaction that takes place is for the client to announce what protocol version it supports. The server will then respond with a message to indicate whether it can accept this protocol version or not. From then on all communication will occur in a form that is specific to the requested protocol version.

 © Borrui Data Technology Co. Ltd 2017

There are specific Thrift message definitions for this compatibility check. It is expected that these messages themselves will not change, as that would imply breaking compatibility with previous protocol versions.

This compatibility check is further described in the 5. Message Flows section below.

The current version of the Wireline Protocol is **VERSION 2**.

5. Message Flows

5.1 Introduction and Semantics

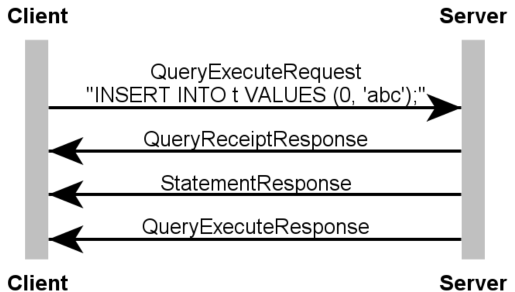
Dialogs represent the flow of messages between a client and a server, typically to fulfil some client request. Most dialogs are request-response based thus they are synchronous in nature, however some event messages from the server are sent asynchronously to convey important real time events. That is, the server only ever receives synchronous messages on a given connection (it should never receive another query on the same connection while it is still processing a prior query) however the server may send messages to the client asynchronously as needed. So the client needs to be aware that it could receive notification messages at any stage of any dialog.

Because the dialogs from the client to the server are synchronous by design, this makes things simpler to design and implement, however it precludes operations such as inspecting the state of a query or even cancelling a query from being asked on the same channel. Instead these operations will need to occur on an alternate channel. This is detailed further below.

The messages involved in the dialogs are therefore named according to their synchronicity for easier identification and understanding: request messages are named `XxxxRequest`, responses are named `XxxxResponse` and asynchronous messages are named `XxxxNotification`.

Request messages are always acknowledged with a Response message. A single Request message may cause multiple Response messages to be sent (e.g., the server sending multiple response messages containing row data), however a Request dialog is always terminated either by an ErrorResponse or another Response message that is similarly named to the Request. This indicates that the Request dialog has completed successfully. This can be seen in the example below, where the client sends a `QueryExecuteRequest` message to perform a SQL INSERT statement. This `QueryExecuteRequest` is ultimately terminated by a `QueryExecuteResponse` message being sent by the server back to the client. In this example the server sends additional responses back to the client before terminating the dialog, such as a unique statement ID and notification that the first statement within the batch (of which this request only has a batch of 1 SQL statement) has completed.

© Borrui Data Technology Co. Ltd 2017



A Request dialog may also involve a subdialog. For example, the client sends a Request to the server which then sends a Request back to the client for further information. The client would then send a Response to the server for that inner dialog which would result in the server sending a Response back for the client's original (outer) Request. This is analogous to being asked a question and having to ask a clarifying question in order to answer it.

All errors will be sent back using the ErrorResponse message, unless the error is such that the client can work around that specific error and try again some other way. i.e., it needs to be a non-fatal error that the client would want to explicitly catch.

The dialogs presented below use specifically named Response types for each Request message, rather than the approach of using a generic "ok" Response message. This is so that debugging a TCP dump or Wireshark trace is easier to follow, particularly for any dialogs involving sub-dialogs.

All dialogs described below are for the current version of the Wireline Protocol.

5.2 Compatibility dialog

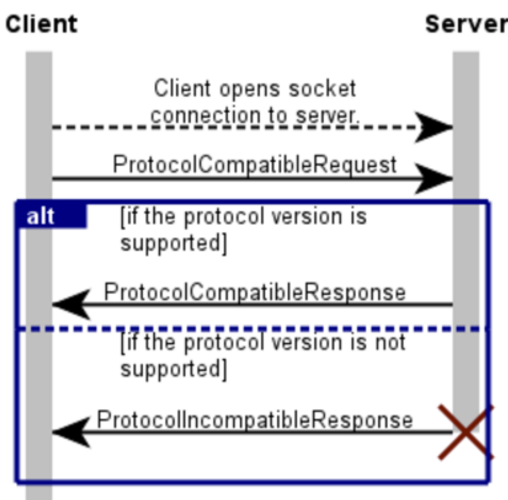
The Compatibility dialog represent the first interactions that occur between the client and server once the client connects. The interactions in this dialog involves checking to see if the client and server can talk to each other using a compatible protocol version that they both support.

In this dialog, the client tells the server which version number of the protocol it supports. The server returns a message indicating whether it supports that version of the protocol or not. If it does not support a protocol version then the server will also disconnect the socket after sending the incompatibility response message.

It is expected that this preliminary protocol compatibility interaction would remain largely unchanged for some significant period of time. This would allow the rest of the protocol to be updatable but still allow a client to easily determine if it is compatible with the protocol or not.

The Compatibility dialog is actually a subset of the overall Connection dialog, described next.

 © Borrui Data Technology Co. Ltd 2017



5.3 Connection dialog

The Connection dialog is the overall sequence of messages when a client connects to the server, before RapidsDB will accept any queries for execution. The Connection dialog actually includes the Compatibility dialog, which has been documented separately for clarity.

In the Connection dialog the concepts of protocol compatibility, user authentication, and session startup are clearly separated out.

After ensuring that both the client and server are talking a compatible protocol version (refer to the Compatibility dialog), the client negotiates the confidentiality of the connection it wants. At this stage RapidsDB only supports clear text tunnels (i.e., no encryption) however it is feasible in the future that RapidsDB may support SSL tunnels or other encryption mechanisms to provide confidentiality on the connection. The client sends the server a `ClearTextTunnelRequest` and the server duly acknowledges this by replying with a `TunnelSupportedResponse`. If the tunnel type was not supported then the server would respond with an `ErrorResponse` message and terminate the connection.

At this point the client and server are both talking a compatible protocol version over a tunnel with an appropriately agreed confidentiality. Now the connection then needs to be authenticated. Currently RapidsDB does not support users or their authentication, but the protocol has been designed to at least accommodate the fact that this may happen at a later stage. Having defined the basis for an authentication dialog will allow for easier backwards compatibility with clients that use an older version of the WLP. At this stage, since RapidsDB does not do any actual authentication the connection is allowed and an `AuthenticationOkResponse` is sent to the client.

After authenticating the connection, the client then needs to start a session with RapidsDB. A `SessionStartRequest` allows the protocol to send any initial session information (such as the default catalog and schema) to RapidsDB and for RapidsDB to validate the parameters and apply them when implemented. If the session parameters are validated successfully then the server will send back a `SessionChangeResponse` message to the client.

© Borrui Data Technology Co. Ltd 2017

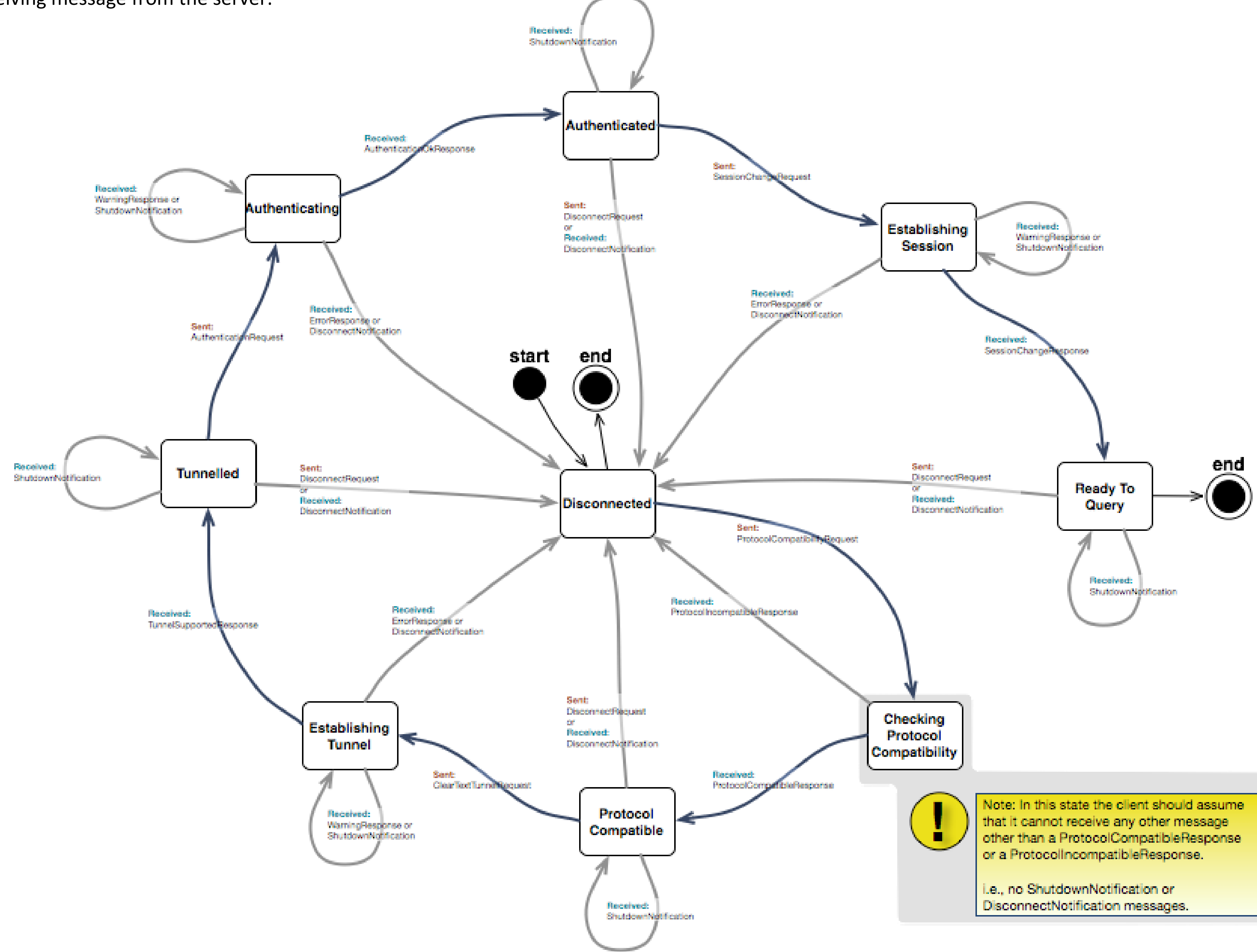
This completes the basic connection dialog. From this point the client is able to submit queries to the server for execution.

Connection Dialog

Client
Server
Client opens socket connection to server.
ProtocolCompatibleRequest
alt [if the protocol version is supported]
ProtocolCompatibleResponse
[if the protocol version is not supported]
ProtocolIncompatibleResponse
Server disconnects connection to Client.
ClearTextTunnelRequest
alt [if the tunnel type is supported]
opt
WarningResponse
TunnelSupportedResponse
[if the tunnel type is not supported]
ErrorResponse
Server disconnects connection to Client.
AuthenticationRequest
alt [if the server requires a password - NOT CURRENTLY IMPLEMENTED]
AuthenticatePasswordRequest
AuthenticatePasswordResponse
opt
WarningResponse
alt [if the user cannot be authenticated]
ErrorResponse
Server disconnects connection to Client.
AuthenticationOkResponse
SessionChangeRequest
alt [if the session parameters are valid]
opt
WarningResponse
SessionChangeResponse
[if the session parameters are invalid]
ErrorResponse
Server disconnects connection to Client.
Client
Server

 © Borrui Data Technology Co. Ltd 2017

The Connection dialog is also described from the client's perspective as a set of states and transitions below, where transitions occur as a result of sending a message to the server or receiving message from the server.



5.4 Query Execute dialog

The Query Execute dialog describes how a client interacts with RapidsDB in order to submit a query to be executed, and how it receives results and metadata. The Wireline Protocol allows for multiple statements to be batched in a single `QueryExecuteRequest` message, delimited by semicolons. RapidsDB will return responses for each individual statement in the batch.

Clients must have successfully completed the Connection dialog in order to start a Query Execute dialog.

The diagram that follows represents the generalised execution of SQL statement, and is henceforth described here.

After submitting a `QueryExecuteRequest`, the server parses the batch of SQL statements and will return an `ErrorResponse` if the batch cannot be parsed. This will end the request.

If the batch is parsed successfully then the server sends the client a `QueryReceiptResponse` message. This acknowledges the statements and gives the client a unique ID and secret key for each executable statement in the batch. This ID and key can later be used for side-band queries to track the progress or cancel the execution of a statement.

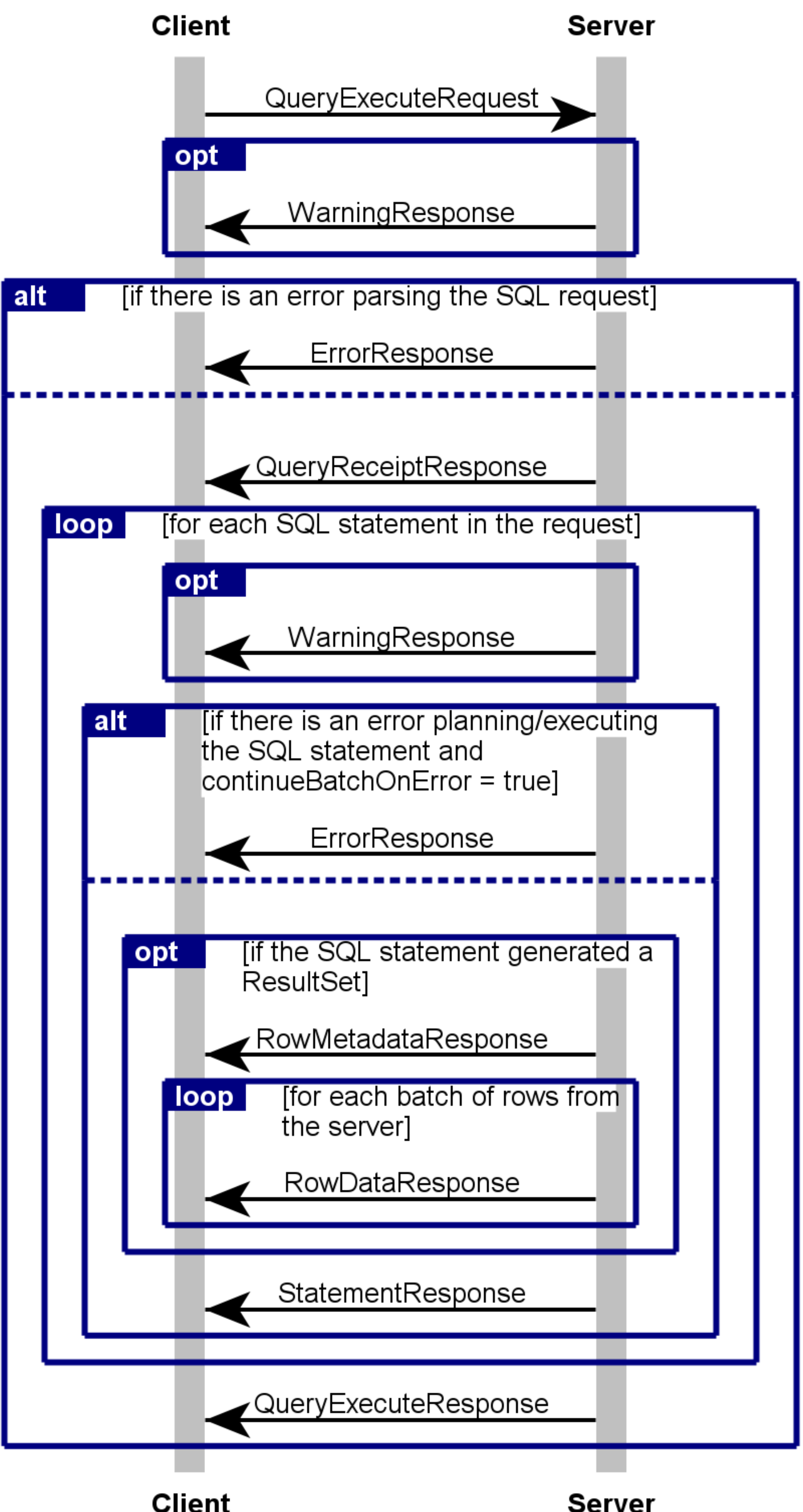
RapidsDB will then execute each statement in the batch in series. If a statement encounters an error during this planning or execution phase then the server will send the client an `ErrorResponse` message. If the *`continueBatchOnError`* field of the original `QueryExecuteRequest` message was set to false (the default) then this ErrorResponse marks the end of the entire Query Execute dialog, and the client is free to submit more requests. However, if this *`continueBatchOnError`* field is set to false then this error only marks the end of execution of this individual statement in the batch, and the server then starts to execute the next statement in the batch.

If a SQL statement generates a result set (e.g., a SELECT statement), even if the result set does not contain any rows, then the server will send the client a `RowMetadataResponse` message that describes the format of any rows to follow. After that, any rows will be batched and encapsulated in `RowDataResponse` messages, until all rows have been sent to the client. At that point the server will send the client a `StatementResponse` message to indicate that execution of the current statement in the batch is complete.

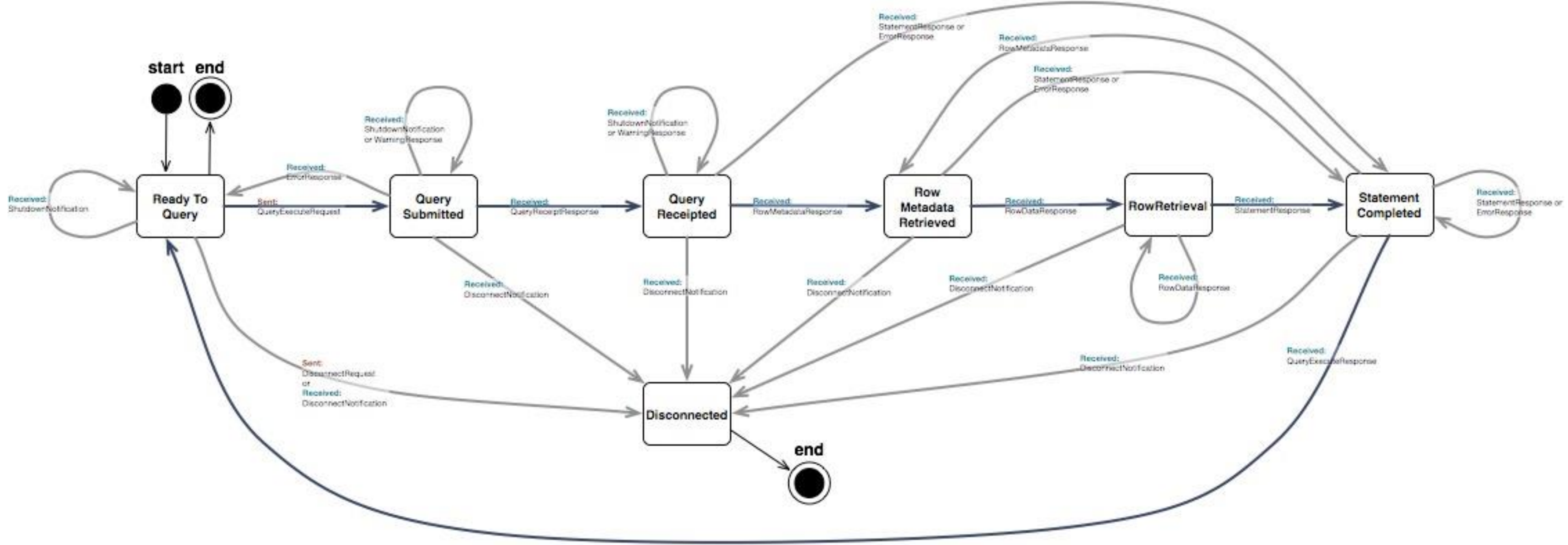
If a SQL statement does not generate a result set (e.g., an INSERT statement) then the server will simply execute the request and send the client a `StatementResponse` message when the execution has completed successfully.

After iterating and executing all statements in the batch, the server will finally send the client a `QueryExecuteResponse` message to indicate that all statements in the original `QueryExecuteRequest` message have been completed. The client is then free to execute subsequent requests.

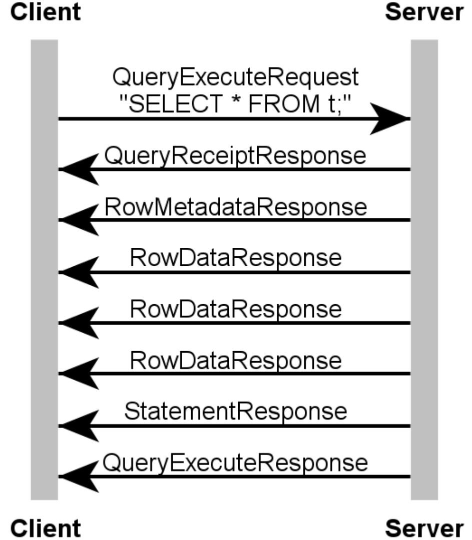
Query Execute Dialog



The Query Execute dialog is also described from the client's perspective as a set of states and transitions below, where transitions occur as a result of sending a message to the server or receiving message from the server.



As a simple, concrete example of executing a single SQL statement, consider the dialog below:



The client sends a `QueryExecuteRequest` message to the server containing a single, simple SQL statement. The server then responds with a `QueryReceiptResponse` message to provide tracking of this statement. It then executes the statement and sends the client a `RowMetadataResponse` message followed by three `RowDataResponse` messages. Each `RowDataResponse` messages can contain many individual rows. Finally the server sends the client a `StatementResponse` message to indicate that the execution of this `SELECT` statement is finished, and since there are no more statements in the batch it concludes the original `QueryExecuteRequest` by sending a `QueryExecuteResponse` message to the client.

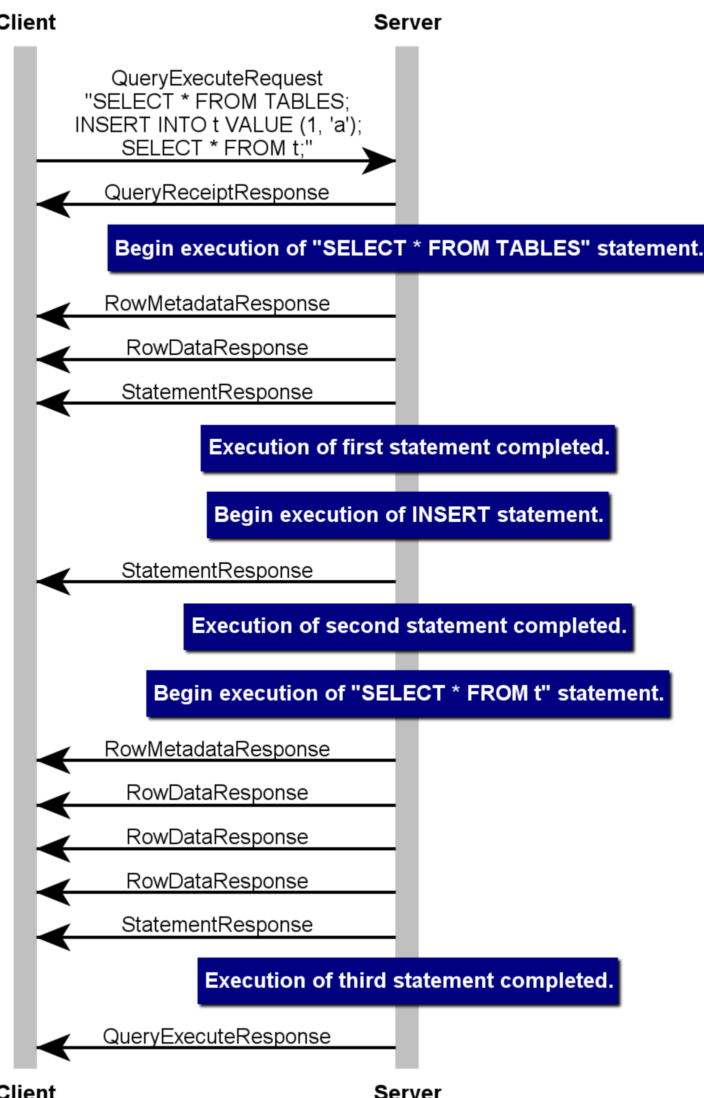
A concrete example of executing a batch of SQL statements in a single `QueryExecuteRequest` message is shown below. In this example the client has sent a single `QueryExecuteRequest` message with a string containing 3 SQL statements delimited by semicolons (a `SELECT`, `INSERT` and a `SELECT`).

The server acknowledges the request by sending the client a `QueryReceiptResponse` with a unique ID and secret key for each of the 3 statements in the batch. The server then begins executing the first statement. Since it is a SELECT statement the server sends a `RowMetadataResponse` message that describes the data that is to come, followed by a `RowDataResponse` message containing the row data. In this case, all the row data was able to fit into a single `RowDataResponse` message. It marks the end of executing this first statement by sending a `StatementResponse` message.

The server then starts to execute the second statement in the batch – an `INSERT` statement. Since there are no rows returned from this, the server simply indicates the successful completion of the statement by sending the client a `StatementResponse` message.

The server then executes the third statement in the batch – another `SELECT` statement. This occurs in the same way described for the first statement in the batch, except that it is a much larger data set and it requires three `RowDataResponse` messages to transport all the row data from the server to the client. A `StatementResponse` message indicates the end of executing this final statement in the batch.

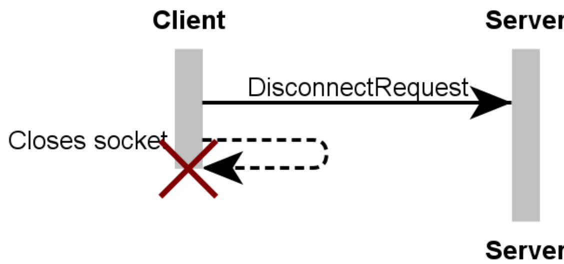
Finally, with all statements in the batch completed successfully, the server sends the client a `QueryExecuteResponse` message to indicate the end of the batch. The client is then able to submit further requests to the server.



5.5 Termination dialog

The termination dialog describes the tear down of the connection between the client and server. This occurs when the client wishes to disconnect from the server in a graceful or non-graceful manner. In any event, a client can simple close the socket that it established to the server to disconnect. Or if it wants to be nice it can send the server a `DisconnectRequest` to tell it that it is about the close the socket. The server does not acknowledge or respond to this request.

 © Borrui Data Technology Co. Ltd 2017



5.6 Other Dialogs

Refer to section 0 (10. Future Enhancements) for additional types of dialogs that may be supported in the future.

6. Messages

6.1 Nomenclature

As identified in section 0 (2. Protocol Overview), message names in the Wireline Protocol have a specific nomenclature that help to identify the origin of the message, the correlation to other messages and the synchronicity of the message. These are identified below:

Message ending with:	Description:
Request	Requests are typically sent from the client to the server (though also sometimes from the server to the client) to ask the server to perform some action. Requests are synchronous, meaning that a second request cannot be sent on the same connection until the first request is completed.
Response	Responses are typically sent from the server back to the client in reply to a Request message. A client may receive multiple responses to a request, however there is typically a single Response message with a similar name to the original Request that indicates that the Request has now been completed. E.g., a `QueryExecuteRequest` is terminated by a `QueryExecuteResponse` message.
Notification	Notification messages are an asynchronous message that can be sent by the server to the client at any time. They would typically be sent to inform the client that some signification action is about to occur that will affect the client. E.g., the server is being shut down, or the client is about to be disconnected for a given reason.

6.2 Compiling Message Definitions

Wireline Protocol messages are defined using the Thrift Interface Definition Language. An IDL was chosen to define messages because it alleviates developers from the tedious and error-prone work of serializing and deserializing messages manually. Instead, developers can focus on the sequencing of messages.

Thrift was chosen as the IDL because it performed better across a wide gamut of tests including serialization/deserialization time and the size of serialized messages. It is also available across a wide range of programming languages.

Thrift is available from https://thrift.apache.org/
The current version of Thrift used by RapidsDB as of Wireline Protocol version 2 is Thrift v0.10.1.

Once thrift is installed, it can be used to compile the Thrift message definition files into native code, for use in a client. A sample script that can be used to compile all Thrift message definition files to Java code is as follows:

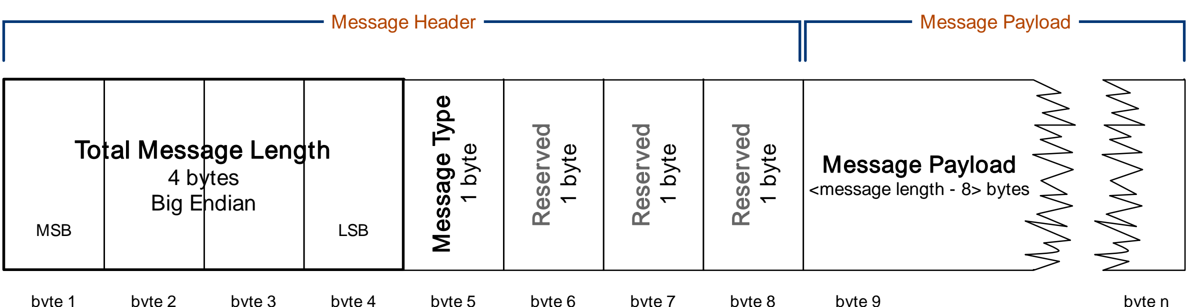
```
#!/bin/bash

DIR=$(dirname "$0")
THRIFT_FILES=$(find "${DIR}" -name *.thrift | xargs)
OUTPUT_DIR="${DIR}/../../java"

for f in $THRIFT_FILES
do
  echo -n "Compiling ${f}... "
  thrift -r --out $OUTPUT_DIR --gen java $f
  echo "Done."
done
```

Table 1 - Sample code to compile Thrift message definitions.

6.3 Message Header

When a Thrift message is serialized it is simply an opaque binary blob. It doesn't contain any information about what type of message it is, or the size of the overall message. As such, we need to prepend a header to message to include this information.

The message header is 8 bytes long. Of these 8 bytes, 4 bytes are used to represent the entire message length (header + payload) and 1 byte is used for the message type. The remaining 3 bytes are reserved for future use. The 8 byte length also ensures that the rest of the message is 64-bit aligned and as such will not incur any data re-alignment penalties on CPUs that prefer data to be aligned. The 4 byte message length will use big endian (or network byte) ordering as this is common on messaging protocols and it is also the default integer byte ordering for Java.



 © Borrui Data Technology Co. Ltd 2017

The interpretation of the message type field will be determined by an enumeration in each version of the Wireline Protocol. For ease of use, it will have the following properties:

- All client initiated requests and notifications will have an odd numbered message type.
- All server sent requests and notifications will have an even numbered message type.
- Any message types that can be sent by either client or server will have an even numbered message type.
- The `ProtocolCompatibleRequest` message will always have a type value of 1.
- The `ProtocolCompatibleRequest` message will always have a type value of 2.
- The `ProtocolIncompatibleRequest` message will always have a type value of 4.

The odd/even numbered assigning of message type IDs is simply to help any debugging efforts if one is looking at a network hex dump.

The message header is expected to remain unchanged across different versions of the Wireline Protocol.

Reading messages off a socket is as simple as reading the first 8 bytes (the message header), then extract the total message length from the first 4 bytes of the header, before reading the payload (`totalMessageLength – 8` bytes). By knowing the `messageType` ID and the version of the Wireline Protocol, the payload can then be deserialized.

6.4 Message Types

The `messageType` field from the header maps a simple 8-bit integer to a Thrift message definition. The interpretation of this integer value depends on the version of the Wireline Protocol being used. Different versions of the Wireline Protocol may add or remove messages (and therefore mappings), or they may even renumber existing messages with a new `messageType` value.

Because the Protocol Compatibility dialog is compatible across all versions of the Wireline Protocol, the messages involved in this dialog must also have fixed values for their `messageType` field. As such, the following messages will always have these values for the `messageType` field, regardless of the protocol version used:

ProtocolCompatibleRequest always has a `messageType` of 1,
ProtocolCompatibleResponse always has a `messageType` of 2,
ProtocolIncompatibleResponse always has a `messageType` of 4.

To determine what Thrift message is mapped to a particular value of the `messageType` field, consult the `MessageTypes` Thrift file. This contains an enumeration that holds these values. As an example, the mappings for version 2 of the Wireline Protocol are as follows:

Message Name	Type
`ProtocolCompatibleRequest`	1
`ProtocolCompatibleResponse`	2
`ProtocolIncompatibleResponse`	4
`ErrorResponse`	6
`WarningResponse`	8

 © Borrui Data Technology Co. Ltd 2017

ShutdownNotification	10
DisconnectNotification	12
DisconnectRequest	13
ClearTextTunnelRequest	21
TunnelSupportedResponse	22
AuthenticationRequest	23
AuthenticatePasswordRequest	24
AuthenticatePasswordResponse	25
AuthenticationOkResponse	26
SessionChangeRequest	27
SessionChangeResponse	28
QueryExecuteRequest	29
QueryReceiptResponse	30
RowMetadataResponse	32
RowDataResponse	34
StatementResponse	36
QueryExecuteResponse	38
QueryCancellationRequest	39
QueryCancellationResponse	40
QueryProgressRequest	41
QueryProgressResponse	42
SessionMonitoringRequest	43
SessionProtocolVersionResponse	44
SessionDataResponse	46
SessionDroppedMessagesResponse	48
SessionMonitoringResponse	50
ParseStatementRequest	51
ParseStatementResponse	52
StatementBindAndExecuteRequest	53
ClosePreparedStatementRequest	55
ClosePreparedStatementResponse	56

Table 2 - Mapping of messages to messageType values.

In the above example, you can also notice that messages typically sent from the client to the server have an odd numbering, whereas messages typically sent from the server to the client have an even message type number. This is done to aid protocol debugging with a packet capturing tool, such as Wireshark.

6.5 Thrift Files

Version 2 of the Wireline Protocol includes a number of Thrift files. Some of these are necessary for dialogs defined above and some are not yet used by RapidsDB, but may be included in the future. The table below shows which dialogs each message is used in.

Message Name	Refer to dialog(s)
AuthenticatePasswordRequest	Not yet used
AuthenticatePasswordResponse	Not yet used
AuthenticationOkResponse	Connection dialog
AuthenticationRequest	Connection dialog
ClearTextTunnelRequest	Connection dialog

 © Borrui Data Technology Co. Ltd 2017

ClosePreparedStatementRequest	Not yet used
ClosePreparedStatementResponse	Not yet used
DisconnectNotification	Not yet used
DisconnectRequest	Termination dialog
ErrorResponse	Connection, Query Execution dialogs
MessageTypes	Used to map message definitions to messageType integer values.
ParseStatementRequest	Not yet used
ParseStatementResponse	Not yet used
ProtocolCompatibleRequest	Compatibility dialog
ProtocolCompatibleResponse	Compatibility dialog
ProtocolIncompatibleResponse	Compatibility dialog
QueryCancellationRequest	Not yet used
QueryCancellationResponse	Not yet used
QueryExecuteRequest	Query Execute dialog
QueryExecuteResponse	Query Execute dialog
QueryProgressRequest	Not yet used
QueryProgressResponse	Not yet used
QueryReceiptResponse	Query Execute dialog
RowDataResponse	Query Execute dialog
RowMetadataResponse	Query Execute dialog
SessionChangeRequest	Connection dialog
SessionChangeResponse	Connection dialog
SessionDataResponse	Not yet used
SessionDroppedMessagesResponse	Not yet used
SessionMonitoringRequest	Not yet used
SessionMonitoringResponse	Not yet used
SessionProtocolVersionResponse	Not yet used
ShutdownNotification	Not yet used
StatementBindAndExecuteRequest	Not yet used
StatementResponse	Query Execute dialog
TunnelSupportedResponse	Connection dialog
WarningResponse	Connection, Query Execution dialogs

Table 3 - Which dialogs that each message is used in.

Also, please note that some Thrift files include other Thrift files because they reuse common definitions. Two examples of that are shown below, where the `RowMetadataResponse` file includes the `ColumnMetadata` file because it defines the data types that the Wireline Protocol supports. Similarly, the `RowDataResponse` file includes the `ColumnValue` file because the latter defines the structure for various composite column types, such as timestamps and intervals. These included Thrift files will be included by additional Thrift messages in the future.

 © Borrui Data Technology Co. Ltd 2017

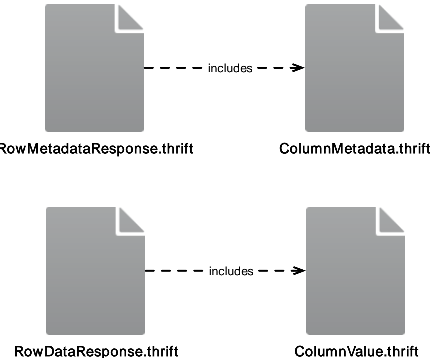


Table 4 - Thrift message inclusion.

7. Sessions

The RapidsDB Wireline Protocol includes stateful session management of clients connected this way. At present, the session holds information on the current catalog and current schema (if set), but it can later be expanded to hold any other useful stateful information.

At the time of writing, the rapids-shell does not connect to RapidsDB via the Wireline Protocol and thus can not make use of these session capabilities.

When a session is established, the client has the opportunity to set the default catalog and schema. This can be done with the `SessionChangeRequest` message, or it can be achieved explicitly with the `SET CATALOG` or `SET SCHEMA` commands. When a default catalog or schema is set, that catalog and/or schema name is used to help qualify any unqualified table names.

E.g., if the default catalog was `WEST` and the schema `SALES` then a query like:

```
SELECT * FROM CUSTOMERS;
```

would be translated to:

```
SELECT * FROM WEST.SALES.CUSTOMERS;
```

Tables that are fully qualified are not affected by the default catalog and schema.

8. Limits And Constraints

The RapidsDB Wireline Protocol has the following limitations and constraints:

1. The maximum size of any message is (2^32) – 8 bytes = 4,294,967,280 bytes. This is because there are 4 bytes allocated in the message header to the payload size, minus the size of the header itself (8 bytes).
   RowDataResponse messages with multiple rows that are larger than this will be split apart and re-serialized into multiple messages until they fit. Fundamentally, this means that a single row cannot be larger than this maximum size once it is serialized.

 © Borrui Data Technology Co. Ltd 2017

2. When a client sends a request to the server, the client cannot send a second request until the server completes the first one. If the client really needs to send a concurrent request then it should open a separate connection to the server.

9. Change History

9.1 Version 1

Version 1 was the initial version of the Wireline Protocol. Some beta versions of RapidsDB that included this were released in February 2017.

9.2 Version 2

Version 2 was released in March 2017 in RapidsDB version 3.1. It is functionally equivalent to protocol version 1 except that the QueryExecuteRequest message gained an extra optional field to set the maximum number of rows to return for any one query. When this value is set to a non-negative number then excess rows above this value are silently dropped on the server. This feature is used by the tool SQuirreL.

10. Future Enhancements

This section seeks to outline some potential future enhancements that can occur to the Wireline Protocol. There is no guarantee that any of the features documented here will appear in a future version, however this information is included so that developers using the Wireline Protocol can get a feel for these potential changes and better structure their implementation accordingly.

10.1 Asynchronous dialogs

This style of dialog is not currently implemented in the RapidsDB wireline protocol, but may be included at a later date. It is included here to help implementers think about how they will structure their implementation.

All of the dialogs previously presented in section 0 (5. Message Flows) have synchronous messaging. i.e., a Request from the client generates a known set of Responses from the server in a semi-defined order. However there are a class of messages that can be sent from the server to the client at any point in time after the client has established a socket connection and completed the Connection dialog. These messages are asynchronous to any dialogs that may happen to be occurring on a given client's connection, thus the client needs to be prepared to receive them at any time.

The purpose of these asynchronous messages is to convey important out-of-band status information about the server or cluster that is connected to, without the client having to request it first. This could include things like:

- Notification that the server or cluster is going to be shutting down.
- Notification of forced session termination from the server.
- Notification of changes to important server or cluster parameters.
- Notification of the health or operability of the server or cluster.

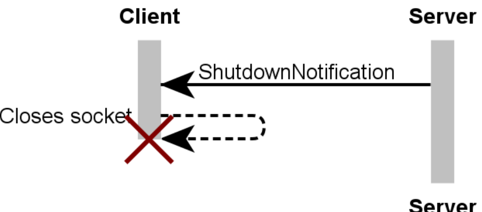
The client should not respond to these messages in any way. It is up to the client whether they wish to simply ignore the message or to do something about it (if possible).

Two asynchronous messages are defined below, relating to server shutdown and forceful client disconnection. These may or may not be implemented straight away in RapidsDB however they are being defined now for future compatibility. It is expected that other asynchronous messages may be defined in the future.

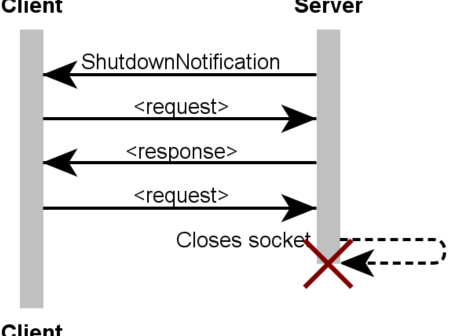
10.1.1 Server Shutdown Notification dialog

The server shutdown notification indicates to a client that the server is intending to gracefully shutdown very soon. The intent here is that the server is giving notice to the clients so that they can finish any open operations and then terminate their connections to the server gracefully.

Server Shutdown Notification

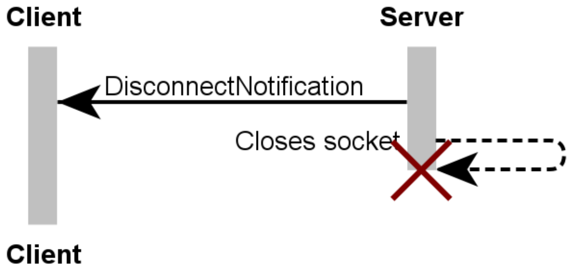


After waiting for some period of time, if the server still has clients connected then it will forcefully disconnect them before completing its shutdown process. This is shown below:



10.1.2 Client Disconnect Notification dialog

The `DisconnectNotification` is an asynchronous message sent from the server to a client to indicate that the server is forcefully disconnecting the client for a given reason. This could be because of a lack of resources or from some direct administrator action. After sending this message, the server immediately disconnects its socket to the client and then cleans up the client's session state.



10.2 Sideband dialogs

This style of dialog is not currently implemented in the RapidsDB wireline protocol, but may be included at a later date. It is included here to help implementers think about how they will structure their implementation.

The semantics of the client's interaction with the server are such that the client can only start a new request after the previous one has been completed. However there would be use-cases whereby the client would wish to interact with a request that is already in flight. These include the ability to cancel a query that is currently executing, the ability to inspect the progress of an executing query or even the ability to monitor in real time the queries and results on another connection (e.g., for logging/debugging purposes).

The way this is done is via sideband requests, that is, a request on a new connection. This approach was chosen because sideband requests would be relatively rare in comparison to the regular Request-Response dialogs. Therefore the majority of communication dialogs should not be overcomplicated by this rarely used feature. Instead, the implementation of sideband requests should incur the cost of any additional complexities.

Whilst RapidsDB has not implemented any of the use cases defined above (cancelling an existing query, inspecting the progress of a query or session monitoring), these are likely to come up in the future, hence these dialogs are described now for greater future compatibility. In the future, other sorts of sideband requests may also be defined.

10.2.1 Query Cancellation dialog

This dialog relates to being able to request the cancellation of a query that is currently executing. While there are no guarantees that a query can be cancelled in a timely manner, this interaction could be crucial to being able to terminate a continuous streaming query.

Because an active query would be utilising a client's connection, and the client cannot submit another request until the current one is finished, the query cancellation must be sent on a new connection. But how does a Request on one connection affect the operation on another connection, and how do we prevent a malicious user from abusing this? This is achieved by the use of a unique ID and secret key for each `QueryExecuteRequest` that the client submits. This unique identifier and secret key is returned

 © Borrui Data Technology Co. Ltd 2017

to the client in a `QueryReceiptResponse` message immediately after the client submits the `QueryExecuteRequest` message (refer to the 5.4 Query Execute dialog). The client would need to extract out the query identifier and secret key and hold on to it for potential future use in a query cancellation dialog.

When the client wishes to request cancellation of the query, the client opens a new connection to the server, completes the Connection dialog as usual and then sends a `QueryCancellationRequest` message to the server with the query ID and secret query key previously returned. The server is then able to attempt to cancel the query at its leisure.

An example of this is shown below. Since this is not yet implemented, this dialog is subject to change.

Query Cancellation Dialog

Client1
Server
QueryExecuteRequest
QueryReceiptResponse
RowMetadataResponse
RowDataResponse
RowDataResponse
Create new connection
Client2
Client2 completes the connection dialog.
QueryCancellationRequest
QueryCancellationResponse
ErrorResponse
QueryExecuteResponse
Supplies query ID and key from the QueryReceiptResponse.
Error code indicates that the query has been cancelled.
Client1
Client2
Server

10.2.2 Query Progress dialog

Similar to the Query Cancellation dialog, the `QueryProgressRequest` message is sent on a sideband connection in order to obtain some information about the progress of a long running query. RapidsDB does not support such a functionality at this current time, however it is foreseeable that such a feature could be asked for. Therefore the protocol should allow such a request to be made.

The `QueryProgressRequest` operates in the same way as the `QueryCancellationRequest` by using the unique and secret identifier returned from a `QueryExecuteRequest`. An example of its usage is shown below. Since this is not yet implemented, this dialog is subject to change.

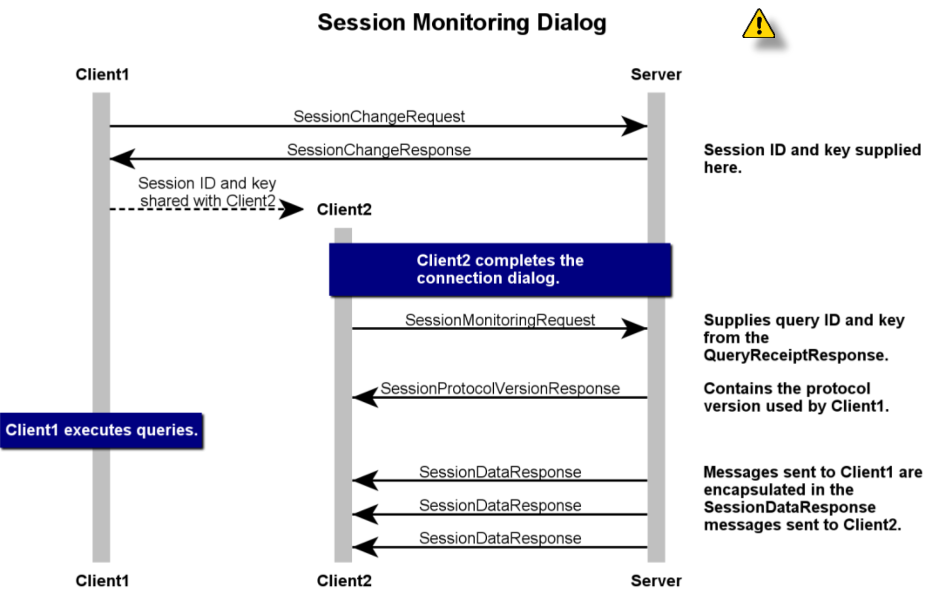
Query Progress Dialog

Client1
Server
QueryExecuteRequest
QueryReceiptResponse
Query ID and key supplied here.
RowMetadataResponse
RowDataResponse
RowDataResponse
Create new connection
Client2
Client2 completes the connection dialog.
QueryProgressRequest
Supplies query ID and key from the QueryReceiptResponse.
QueryProgressResponse
QueryProgressRequest
QueryProgressResponse
Some time later...
StatementResponse
QueryExecuteResponse
Client1
Client2
Server

10.2.3 Session Monitoring dialog

Session monitoring is the ability to snoop on an existing session and see all the requests and responses that are flowing across that session. This would be useful for logging all input and output of a session independently of that session for the purposes of offline analysis, debugging or general monitoring.

While this presents a leak of potentially privileged information, it is still protected by the same security mechanism that is used for cancelling queries or inspecting the progress of queries. This feature is not currently implemented in RapidsDB but is being described here to allow for user's to factor this in to any usage of the Wireline Protocol. This dialog is subject to significant change.



 © Borrui Data Technology Co. Ltd 2017