VoltDB Client Wire Protocol

VoltDB Team - 01/26/2016 - VoltDB Client Wire Protocol Version 1

Protocol Version

The current version of the protocol is version 1. VoltDB servers using wire protocol version 1 will interpret and respond appropriately to messages sent using either version 0 or version 1 of the wire protocol.

Overview

A client connection to a VoltDB instance consists of a TCP connection on port 21212. After the initial login process the only exchange between the client library and the VoltDB server is the invocation of and response to stored procedures. All messages in the VoltDB wire protocol are in network byte order and all integers are signed. Every message is length preceded with a 4 byte integer and the length does not include the length value. The first value after the length preceding value is the version number of the wire protocol represented as a byte. These two items precede all messages.

The login message is the first message the client library sends to the VoltDB server and it is required even if authentication is disabled in the server's configuration. The login message consists of a service name string, a username string and a 256-bit SHA-256 hash of the password. The response from the server consists of a response byte. A value of 0 indicates successful authentication and all other values indicate failure. If authentication fails the server will close the connection. The client can safely send invocations before receiving an authentication response.

The procedure invocation request contains the procedure to be called by name, and the serialized parameters to the procedure. The message also includes an opaque 8 byte client data value which will be returned with the response, and can be used by the client to correlate requests with responses.

The returned response contains

- a byte status code,
- an integer measuring intra-cluster latency,
- a serialized exception if an error occurred and the exception was serializable,
- an array of VoltTables, which may be of length 0, and is never null,
- a string value containing any extra information the server included, and
- the 8 byte client data value contained in the originating procedure invocation.

The following sections describe how values are serialized. The wire protocol is still under development and will be adapted to support new authentication methods and new data types as necessary.

Basic Data Types

Binary fields do not have a wire type associated with them but they are present in certain messages. Binary fields, such as opaque client data or hashes, do not have any byte order, sign, or size other then what is specified in the message format.

Integer Types:

All integer types are signed, twos-compliment and in network byte order.

- Byte - 1 Byte
- Short - 2 Bytes
- Integer - 4 Bytes
- Long - 8 Bytes

Floating Point Type

Only 8-Byte Double floating point values are supported using the byte representation in IEEE 754 "double format." Positive and negative infinity, as well as NaN, are supported on the wire, but not guaranteed to work as SQL values.

String Type

Strings begin with a 4-byte integer storing the number of bytes of character data, followed by the character data. UTF-8 is the only supported character encoding. The NULL string has a length preceded value of -1 (negative one) followed by 0 (zero) bytes of string data. The empty string is represented with a length preceding value of 0.

A String encoded into a parameter set has to be deserialized as a Java String before being passed to a stored procedure and this is a relatively slow operation. If the stored procedure does not need the Java String representation it can specify a byte array as its argument instead of String. The String should then be presented in the parameter set as a byte array (not preceded with the String wire type).

If an array of bytes is passed as a parameter to a SQL statement in a stored procedure, and that parameter expects a string, it will automatically be converted to a string on the native side without any Java deserialization. SQL statements do not support array parameters so this is not ambiguous. In the current implementation this can have significantly better performance, especially with large parameter sets with many strings.

The following table presents the byte by byte serialization of the string "foo".

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	String Length	3	String is 3 bytes long
01	0			
02	0			
03	3			
04	102	Characters	f	String data is "foo"
05	111		o	
06	111		o	

Varbinary Type

Varbinary is a effectively a binary string, using the same serialization and storage.

Like strings, varbinary begin with a 4-byte integer storing the number of bytes of raw data, followed by the raw data itself. The NULL varbinary has a length preceded value of -1 (negative one) followed by 0 (zero) bytes of data. The value of zero bytes is represented with a length preceding value of 0.

The native java type for varbinary is byte[]. Stored procedures and SQL statements will accept byte[] as input for varbinary parameters. For compatibility with textual SQL and less binary-friendly clients, varbinary parameters may also be passed as hexadecimal strings using standard string serialization and the string type code indicator. For example, the string "aa" would represent a single byte of value 170. The hex-encoding is case-insensitive and will fail on invalid input, such as odd-length strings. The string prefix, "0x", is not used and will cause exceptions in VoltDB code.

Geography Point Value

It should be noted that, although a description of the GeographyPointValue structure is being provided here for completeness, in most cases the client interface does *not* need to interpret the structure. Generally the client passes the point representation unchanged between the server and the client application.

A GeographyPointValue represents a point on the surface of the earth. These points are represented by a <longitude, latitude> pair. GeographyPointValues have fixed size, so they are not preceded by their length.

A GeographyPointValue's wire protocol representation is simply two double precision numbers in sequence. The first is the longitude, and must be in the range -180 $\leq$ longitude $\leq$ 180. The second is the latitude, and must be in the range -90 $\leq$ latitude $\leq$ 90. The null GeographyPoint value has longitude and latitude both equal to 360.0.

Geography Point Example

The following table represents the coordinates of Santa Cruz, CA.

Byte Offset	Field Length	Field Value	Field Description	Meaning
0	8	-122.0264	Longitude	Longitude is 122.0266W
8	8	36.90719	Latitude	Latitude is 36.90719N

Geography Value

It should be noted that, although a description of the GeographyValue structure is being provided here for completeness, in most cases the client interface does *not* need to interpret the structure. Generally the client passes the GeographyValue representation unchanged between the server and the client application.

A GeographyValue data item represents a region of the earth. In version 6.0 we only support polygons, with and without holes. In the future we may support other kinds of geography values. The wire protocol data format for polygons is not identical to the Java representation. There are coordinate transformations and ring transformations, which we detail now.

Execution Engine Geography Value Coordinate and Ring Transformations

The wire protocol representation is similar to the S2 representation which is used in the Execution Engine. This format is different from the Java longitude/latitude format, so some explanation is in order. In Java a point is represented by a longitude and latitude. In the wire protocol a point is represented by a three dimensional point on the unit sphere. These three dimensional points are called `XYZPoint`s. Each dimension is a double precision IEEE floating point number. The Euclidean length of each `XYZPoint` must be 1.0. The formula to convert a longitude/latitude pair to `XYZPoint` is given by the following pseudo-code. Let `longitude` and `latitude` be double precision floating point numbers measured in degrees.

```
double radiansPerDegree = (Math.PI/180.0);        // A conversion factor.
double latRadians = latitude * radiansPerDegree;  // latitude is in degrees.
double lngRadians = longitude * radiansPerDegree; // longitude is in degrees.
double cosPhi = Math.cos(latRadians);
double x = Math.cos(lngRadians) * cosPhi;
double y = Math.cos(lngRadians) * cosPhi;
double z = Math.sin(latRadians);
```

The three dimensional coordinates of the `XYZPoint` are `x`, `y` and `z` above. To convert from `XYZPoint` to longitude/latitude pairs use the formula from this pseudocode. Let `x`, `y` and `z` be the coordinates of an `XYZPoint`.

```
double degreesPerRadian = (180.0/Math.PI);
double lngRadians = Math.atan2(y, x);
double latRadians = Math.atan2(z, Math.sqrt(x * x + y * y)
double longitude = lngRadians * degreesPerRadian;
double latRadians = latRadians * degreesPerRadian;
```

The longitude and latitude are given by the variables `longitude` and `latitude` above.

In addition to coordinate transformations, the rings which comprise the polygons are transformed. In the Java representation the first ring is the exterior boundary, must be counter clockwise and must start and end with the same point. Subsequent rings are holes, must be clockwise and must start and end with the same point. In the wire protocol the first ring is still the exterior boundary and subsequent rings are holes. However, in the wire protocol all rings, exterior and hole alike, must be counter clockwise, and the last point should not be equal to the first point. To transform a ring from Java representation to wire protocol representation one must:

- Remove the last vertex, which is the same as the first vertex,
- Transform the coordinates to `XYZPoint` values, and
- Reverse the order of the rings from the second to the end.

It's best to reverse the rings by reversing the points from the second to the last. This leaves the first point invariant.

Transforming from the wire protocol value to Java values requires reversing the three steps above.

Geography Value Wire Representation

Like strings, GeographyValue begins with a 4-byte integer storing the number of bytes of raw data, followed by the raw data itself. The NULL GeographyValue has a length of -1 followed by 0 (zero) bytes of data. Unlike strings, there are no zero byte GeographyValue data values.

The raw data has a particular format. Some of this format is taken from the S2 format, to avoid copying. Consequently some of the fields will be filled in by the Execution Engine, but are represented in the protocol. These fields need to be given initial values if a polygon is created by an application, but their values should be maintained if the application is reading a polygon from the server. We use the phase *initially K but maintained on read* for such values when their initial value is *K*.

1. The first byte, byte 0, is an encoding version, which tells whether certain fields need to be initialized by the Execution Engine. This is initially zero (0) and should be maintained on read.
2. The next byte, byte 1, is internal. It should be initially 1, and should be maintained on read.
3. The next byte, byte 2, is 1 if the polygon has holes and 0 if it does not.

4. The next four bytes, bytes 3, 4, 5 and 6, comprise a 32 bit integer which gives the number of rings. Call this value `NRINGS`

5. Next follows NRINGS ring representations. Each ring representation is variable sized, and is described below.

   a. The first byte of a ring tells if the ring is initialized. It is initially zero (0) and should be maintained on read.

   b. The next 4 bytes are a 32-bit integer containing the number of vertices in the ring. Call this number `NVERTS`.

   c. The next `NVERTS*3*8` bytes are `NVERTS` triples of double precision floating point numbers, in the order `X`, `Y` and `Z`.

   d. The next 38 bytes contain a bounding box and some internal fields. They should all be initially zero (0) and should be maintained on read.

6. The next 33 bytes, after all the vertices, should be initially zero (0) and should be maintained on read.

Geography Value Wire Protocol Example

Table WirePoly below is an example of the the initial representation of the polygon 'polygon((0 0, 1 0, 1 1, 0 1, 0 0), ((0.1 0.1, 0.1 0.9, 0.9 0.9, 0.9 0.1, 0.1 0.1))'. This is a polygon with one somewhat large hole in the center. Some fields will be given values by the Execution Engine, so they are given their initial values here.

There are 8 interesting points in this polygon. We show their `XYZPoint` coordinates here for reference.

`XYZPoint` values for points in Table WirePoly.

Longitude	Latitude	x	y	z
0.000000	0.000000	1.000000	0.000000	0.000000
1.000000	0.000000	0.999848	0.017452	0.000000
1.000000	1.000000	0.999695	0.017450	0.017452
0.000000	1.000000	0.999848	0.000000	0.017452
0.100000	0.100000	0.999997	0.001745	0.001745
0.100000	0.900000	0.999875	0.001745	0.015707
0.900000	0.900000	0.999753	0.015705	0.015707
0.900000	0.100000	0.999875	0.015707	0.001745

This is the initial representation of the polygon given in well known text above. Note that the second ring has been reversed.

Table WirePoly: Representation of a polygon.

Byte Offset	Length in Bytes	Value	Data Type	Meaning
0	1	byte	0	IsValid. Initially zero (0)
1	1	byte	1	Internal. Initially zero (0)
2	1	byte	1	Polygon has holes.
3	4	32 bit int	2	Number of Rings

Byte Offset	**Length in Bytes**	**Value**	**Data Type**	**Meaning**
Vertices follow here.				
Ring 1				
7	1	byte	0	Is initialized. Initially zero (0)
8	4	32 bit int	4	Number Vertices in ring 1
12	8	double	1.000000	X Coordinate for ring 1, vertex 1
20	8	double	0.000000	Y Coordinate for ring 1, vertex 1
28	8	double	0.000000	Z Coordinate for ring 1, vertex 1
36	8	double	0.999848	X Coordinate for ring 1, vertex 2
44	8	double	0.017452	Y Coordinate for ring 1, vertex 2
52	8	double	0.000000	Z Coordinate for ring 1, vertex 2
60	8	double	0.999695	X Coordinate for ring 1, vertex 3
68	8	double	0.017450	Y Coordinate for ring 1, vertex 3
76	8	double	0.017452	Z Coordinate for ring 1, vertex 3
84	8	double	0.999848	X Coordinate for ring 1, vertex 4
92	8	double	0.000000	Y Coordinate for ring 1, vertex 4
100	8	double	0.017452	Z Coordinate for ring 1, vertex 4
108	38	0	blob of zeros	Internal plus the bounding box of the ring. Initially zero (0).
Ring 2				
146	1	byte	0	Is initialized. Initially zero (0)
147	4	32 bit int	4	Number Vertices in ring 2
151	8	double	0.999997	X Coordinate for ring 2, vertex 1
159	8	double	0.001745	Y Coordinate for ring 2, vertex 1
167	8	double	0.001745	Z Coordinate for ring 2, vertex 1

Byte Offset	Length in Bytes	Value	Data Type	Meaning
175	8	double	0.999875	X Coordinate for ring 2, vertex 2
183	8	double	0.015707	Y Coordinate for ring 2, vertex 2
191	8	double	0.001745	Z Coordinate for ring 2, vertex 2
199	8	double	0.999753	X Coordinate for ring 2, vertex 3
207	8	double	0.015705	Y Coordinate for ring 2, vertex 3
215	8	double	0.015707	Z Coordinate for ring 2, vertex 3
223	8	double	0.999875	X Coordinate for ring 2, vertex 4
231	8	double	0.001745	Y Coordinate for ring 2, vertex 4
239	8	double	0.015707	Z Coordinate for ring 2, vertex 4
247	38	0	blob of zeros	Internal plus the bounding box of the ring. Initially zero (0).
285	33	0	blob of zeros	Internal fields plus the bounding box of the polygon. Initially zero(0).

Date Type

All dates are represented on the wire as Long values. This signed number represents the number of microseconds before or after Jan. 1 1970 00:00:00 GMT, the Unix epoch. Note that the units are microseconds, not milliseconds.

Decimal Type

VoltDB implements a fixed precision and scale DECIMAL(38,12) type. This type is serialized as a 128 bit signed twos complement integer representing the un-scaled decimal value. The integer must be in network byte order. Null is serialized as the smallest representable value which is "-170141183460469231731687303715884105728." Serializing values (Null excluded) that are greater than "99999999999999999999999999999999999999" or less than "-99999999999999999999999999999999999999" will result in an error response.

Note that the serialization is given above in terms of a 128 bit integer. Since the logical value represented is actually scaled by 12 decimal places, the logical maximum value, minimum value and null values are "99999999999999999999999999.999999999999", "-99999999999999999999999999.999999999999" and "-170141183460469231731687303.715884105728" respectively.

The following table presents the byte by byte serialization of the number "-23325.23425"

Byte Offset	Byte Value (dec)
00	-1

Byte Offset	Byte Value (dec)
01	-1
02	-1
03	-1
04	-1
05	-1
06	-1
07	-1
08	-1
09	-83
10	33
11	-46
12	-78
13	57
14	-39
15	-128

Application Specific Data Types

Wire Type Info

A Byte value specifying the type of a serialized value on the wire.

- ARRAY = -99
- NULL = 1
- TINYINT = 3
- SMALLINT = 4
- INTEGER = 5
- BIGINT = 6
- FLOAT = 8
- STRING = 9
- TIMESTAMP = 11
- DECIMAL = 22
- VARBINARY = 25
- GEOGRAPHY_POINT = 26
- GEOGRAPHY = 27

Procedure Call Status Code

A Byte value specifying the success or failure of a remote stored procedure call.

- SUCCESS = 1
- USER_ABORT = -1
- GRACEFUL_FAILURE = -2
- UNEXPECTED_FAILURE = -3
- CONNECTION_LOST = -4

Compound Data Types

Array Types

Arrays are represented as Byte value indicating the wire type of the elements and a 2 byte Short value indicating the number of elements in the array, followed by the specified number of elements. The length preceding value for the TINYINT (byte) type is length preceded by a 4 byte integer instead of a 2 byte short. This important exception allows large quantities of binary or string data to be passed as a byte array, and allows the serialization of and array of TINYINTs (bytes) to match the serialization of VARBINARY. Each array is serialized according to its type (Strings as Strings, VoltTables as VoltTables, Integers as Integers). Arrays are only present as parameters in parameter sets. Note that it is possible to pass an array of arrays of bytes if they are serialized as an array of VARBINARY types.

- Size is limited to 32,767 values due to the signed short length with the exception of TINYINT (byte) arrays which use a 4 byte integer length.
- All values must be homogeneous with respect to type.

The following example serialization shows an array with two String elements ("foo1", "foo2").

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	9	Element Type	9	Array Elements are Strings
01	0	Element Count	2	Array contains two elements
02	2			
03	0	String Length	4	String is 4 bytes long
04	0			
05	0			
06	4			
07	102	Characters	f	String data is "foo1"
08	111		o	
09	111		o	
10	49		1	
11	0	String Length	4	String is 4 bytes long
12	0			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
13	0			
14	4			
15	102	Characters	f	String data is "foo2"
16	111		o	
17	111		o	
18	50		2	

Complex API Data Types

VoltTable

On the wire a VoltTable is serialized as a header followed by tuple data. VoltTables, like all VoltDB serialized structures are stored in network byte order.

> (*Note:* In the following description, the term "sequence" means a sequence of objects of the specified type, not a VoltDB array object as described in the previous section. Because the number of columns is fixed and is already part of the VoltTable descriptor, we know how many objects are in each sequence and a full array descriptor is not needed.)

It should also be noted that although a description of the VoltTable structure is being provided here for completeness, in most cases the client interface does *not* need to interpret the structure, but rather passes it unchanged between the server and the client application.

Name	**Type**	**Length (bytes)**	**[Basic | Compound | Complex]**
Total table length	Integer	4	Basic
Table Metadata Length	Integer	4	Basic
Status Code	Byte	1	Basic
Column Count	Short	2	Basic
Column Types	Sequence of Bytes	variable	Compound
Column Names	Sequence of Strings	variable	Compound
Row Count	Integer	4	Basic
Row Data...			

Notes on the Header Format:

- The "Table Metadata Length" stores the length in bytes of the contents of the table from byte 8 (the end of the metadata length field) all the way to the end of the "Column Names" sequence. NOTE: It does not include the row count value. See below for an example.
- The size of the "Column Types" and "Column Names" sequences is expected to equal the value stored in "Column Count".
- Column names are limited to the ASCII character set. Strings in row values are still UTF-8 encoded.
- Values with 4-byte (integer) length fields are signed and are limited to a max of 1 megabyte.

Row Data Format:

Each row is preceded by a 4 byte integer that holds the length of the row not including the length. For example, if a row is a single 4-byte integer column, the value of this length prefix will be 4. Row size is artificially restricted to 2 megabytes.

The body of the row is packed array of values. The value at index i is is of type specified by the column type field for index i. The values are serialized according to the serialization rules in "Basic Data Types" above.

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Row Length	Integer	4	Basic
Single Row Value Array			

Example VoltTable Serialization:

The following is a 35-byte long serialized table containing one column named "Test" of type BIGINT with one row containing the number 5.

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	Total Table Length	31	Table size is 31 bytes
01	0			
02	0			
03	31			
04	0	Table Metadata Length	12	Header size is 12 bytes
05	0			
06	0			
07	12			
08	0	Status Code	0	Status code is 0
09	0	Column Count	1	Table contains 1 column
10	1			
11	6	Column 1 Type	VoltType.BIGINT	Column 1 is a BigInt
12	0	Col 1 Name Length	4	The name of column 1 has 4 chars (ASCII)
13	0			
14	0			
15	4			
16	84	Column 1 Name Value	T	Column 1 is named "Test"
17	101		e	
18	115		s	
19	116		t	
20	0	Row Count	1	Table has 1 row
21	0			
22	0			
23	1			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
24	0	Row 1 Length	8	First row is 8 bytes long
25	0			
26	0			
27	8			
28	0	Row 1 Col 1 Value	5	Value in first row/first col is 5
29	0			
30	0			
31	0			
32	0			
33	0			
34	0			
35	5			

Serializable exceptions

Currently serializable exceptions are not a part of the wire protocol although they are present in the invocation response message. The length of every serialized exception as a 4 byte integer is part of the invocation response message. It can be used to skip the exception. It is possible to retrieve the Byte ordinal that follows the exception's length. The ordinal will not be present if the exception's length is 0.

Exception Ordinal Name	Exception Ordinal Value	Description
EEException	1	This is a generic failure in Volt. Should indicate a failure in the server and not the application code. These should not occur in normal operation.
SQLException	2	This is the base class for all exceptions that can occur during normal operation. This includes things like constraint failures (unique, string length, not null) that are caught and handled correct by Volt.
ConstraintFailureException	3	This is a specialization of SQLException for constraint failures during the execution of a stored procedure.

In the future a set of serializable exceptions and their serialization format will be added to the wire protocol.

Parameter Set

A parameter set contains all the parameters to be passed to a stored procedure and it is one of the structures bundled inside a stored procedure invocation request. The first value of a parameter set is a Short indicating the number of parameters that follow. The following values are a series of <wire type, value> pairs. Each value is preceded by its wire type represented as a Byte. NULL is a valid wire type and value and it is not followed by any additional value. Arrays are preceded by the wire type -99 and the array value contains the type of the array elements as well as the number of elements (see Array type). A parameter set cannot contain a nested parameter set (there is no wire type for parameter set).

Note that varbinary values using type number 25 and arrays of bytes using type number -99, followed by type 3 are effectively interchangeable.

Parameter set

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Parameter count	Short	2	Basic
Parameters...			

Parameter

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Parameter type	Byte	1	Basic
Parameter	Any Basic | Compound | Complex excluding parameter sets	variable	Basic | Compound | Complex

The following example parameter set serialization shows a parameter set consisting of an array of two strings ("foo1, "foo2") and a decimal.

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	Parameter Count	2	Parameter set contains two parameters
01	2			
02	-99	Parameter Type	-99	Next parameter is an array
03	9	Element Type	9	Array Elements are Strings
04	0	Element Count	2	Array contains two elements
05	2			
06	0	String Length	4	String is 4 bytes long
07	0			
08	0			
09	4			
10	102	Characters	f	String data is "foo1"
11	111		o	
12	111		o	
13	49		1	
14	0	String Length	4	String is 4 bytes long
15	0			
16	0			
17	4			
18	102	Characters	f	String data is "foo2"
19	111		o	

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
20	111		o	
21	50		2	
22	22	Parameter Type	22	Next parameter is a decimal
23	-1	Decimal data	-23325.23425	Decimal value "-23325.23425"
24	-1			
25	-1			
26	-1			
27	-1			
28	-1			
29	-1			
30	-1			
31	-1			
32	-83			
33	33			
34	-46			
35	-78			
36	57			
37	-39			
38	-128			

Message formats

Message header

The header that is included at the beginning of all messages. The length value includes the protocol version byte but not the 4 byte length value.

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Message length	Integer	4	Basic
Protocol version	Byte	1	Basic

The following table shows and example header for a 140,000 byte message.

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	Message Length	140000	The message is 140,000 bytes long
01	2			
02	34			
03	-32			
04	0	Protocol version	1	This messages is a Volt Wire Protocol version 1 message

Login message

The login message is the first message a client can send to a server after opening a connection. A client does not need to wait for a response to the login message to begin sending invocation requests. The login message identifies a service to authenticate to. There are currently two supported services: the "database" service authenticates stored procedure callers; the "export" service authenticates export connector callers. Export connectors are extensible and future plugin connectors may handle other service string values.

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Message Header			
Password hash version	Byte	0 or 1	Basic
Service	String	variable	Basic
Username	String	variable	Basic
SHA-1 or SHA-2 password hash	Binary	20 or 32	Basic

The size and type of the password hash depends on the hash version number. If the version is 0, the hash is a 20-byte SHA-1 hash. If the version number is 1, the hash is a 32-byte SHA-256 hash.

The following table shows an example login message for username "scooby" password "doo" using the recommended SHA-256 hashing.

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	Message Length	81	The message is 80 bytes long
01	0			
02	0			
03	81			
04	0	Protocol version	1	This messages is a Volt Wire Protocol version 1 message
05	1	Password Hash version	1	The password is being sent as a SHA-256 hash
06	0	Service length	8	Service string is 8 bytes long
07	0			
08	0			
09	8			
10	100	Characters	d	Service string is "database"
11	97		a	
12	116		t	
13	97		a	
14	98		b	
15	97		a	
16	115		s	
17	101		e	

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
18	0	Username length	6	Username is 6 bytes long
19	0			
20	0			
21	6			
22	115	Characters	s	Username is "scooby"
23	99		c	
24	111		o	
25	111		o	
26	98		b	
27	121		y	
28	119	SHA-256 Hash	SHA-256("doo")	Password has is SHA-256("doo")
29	-116			
30	85			
31	62			
32	-6			
33	0			
34	-45			
35	-60			
36	36			
37	14			
38	109			
39	-96			
40	79			
41	82			
42	90			
43	60			
44	-123			
45	-24			
46	35			
47	38			
48	12			
49	126			
50	-59			
51	-98			
52	-86			
53	-76			
54	-118			
55	64			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
56	-84			
57	-23			
58	110			
59	3			

Login response

A response is generated to a login request and success is indicated with a result code of 0. Any other value indicates authentication failure and will be followed by the server closing the connection. A response code of 1 indicates that the there are too many connections. A response code of 2 indicates that authentication failed because the client took too long to transmit credentials. A response code of 3 indicates a corrupt or invalid login message. If the response code is 0 the response will also contain additional information following the result code. A 4 byte integer specifying the host id of the Volt node . An 8 byte long specifying a connection id that is unique among connections to that node. An 8 byte long timestamp (milliseconds since Unix epoch) and a 4 byte IPV4 address representing the time the cluster was started and the address of the leader node. These two values uniquely identify a Volt cluster instance. And finally a string containing a textual description of the build the node being connected to is running.

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Message Header			
Authentication result code	Byte	1	Basic
Server Host ID	Integer	4	Basic
Connection ID	Long	8	Basic
Cluster start timestamp (milliseconds since Unix epoch)	Long	8	Basic
Leader IPV4 address	Integer	4	Basic
Build string	String	variable	Basic

The following table shows a login response indicating success

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	Message Length	82	The message is 82 bytes long
01	0			
02	0			
03	82			
04	0	Protocol version	0	This messages is a Volt Wire Protocol version 0 message
05	0	Result code	0	Authentication succeded
06	0	Server Host ID	0	The Host ID of the server is 0
07	0			
08	0			
09	0			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
10	0	Connection ID	12	The ID of the connection is 12
11	0			
12	0			
13	0			
14	0			
15	0			
16	0			
17	12			
18	0	Cluster start timestamp	105	The cluster was started 105 milliseconds after the Unix epoch
19	0			
20	0			
21	0			
22	0			
23	0			
24	0			
25	105			
26	192	Leader IPV4 address	192.168.0.1	The IPV4 address of the leader that started the cluster was 192.168.0.1
27	168			
28	0			
29	1			
30	0	Build string length	52	The length of the build string is 52
31	0			
32	0			
33	52			
34	48 ("0")	Build string	0.7.01 https://svn.voltdb.com/eng/trunk?revision=443	Build is version 0.7.01 off the trunk revision 443/td>
35	46 (".")			
36	55 ("7")			
37	46 (".")			
38	48 ("0")			
39	49 ("1")			
40	32 (" ")			
41	104 ("h"			
42	116 ("t")			
43	116 ("t")			
44	112 ("p")			
45	115 ("s")			
46	58 (":")			
47	47 ("/")			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
48	47 ("/")			
49	115 ("s")			
50	118 ("v")			
51	110 ("n")			
52	46 (".")			
53	118 ("v")			
54	111 ("o")			
55	108 ("l")			
56	116 ("t")			
57	100 ("d")			
58	98 ("b")			
59	46 (".")			
60	99 ("c")			
61	111 ("o")			
62	109 ("m")			
63	47 ("/")			
64	101 ("e")			
65	110 ("n")			
66	103 ("g")			
67	47 ("/")			
68	116 ("t")			
69	114 ("r")			
70	117 ("u")			
71	110 ("n")			
72	107 ("k")			
73	63 ("?")			
74	114 ("r")			
75	101 ("e")			
76	118 ("v")			
77	105 ("i")			
78	115 ("s")			
79	105 ("i")			
80	111 ("o")			
81	110 ("n")			
82	61 ("=")			
83	52 ("4")			
84	52 ("4")			
85	51 ("3")			

Invocation request

A request to invoke a stored procedure identifies the procedure to invoke by name, the parameters to pass to the procedure, and an 8 byte piece of client data that will be returned with the response to the invocation request. A client does not need to wait for a response to a request to continue sending requests. The server will use TCP backpressure to avoid running out of memory when a client sends too many invocations for the server to handle.

Name	**Type**	**Length (bytes)**	**[Basic | Compound | Complex]**
Message Header			
Procedure name	String	variable	Basic
Client data	Binary	8	Basic
Parameters	Parameter Set	variable	Complex

The following table shows the serialization for invoking a procedure called "proc" with a parameter set containing an array of two strings ("foo1", "foo2") and a decimal value "-23325.23425"

Byte Offset	**Byte Value (dec)**	**Field Desc**	**Field Value**	**Meaning**
00	0	Message Length	56	The message is 56 bytes long
01	0			
02	0			
03	56			
04	0	Protocol version	0	This messages is a Volt Wire Protocol version 0 message
05	0	Procedure name length	4	Procedure name is 4 bytes long
06	0			
07	0			
08	4			
09	112	Characters	p	Stored procedure name is "proc"
10	114		r	
11	111		o	
12	99		c	
13	0	Client Data		Opaque client data
14	1			
15	2			
16	3			
17	4			
18	5			
19	6			
20	7			
21	0	Parameter Count	2	Parameter set contains two parameters
22	2			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
23	-99	Parameter Type	-99	Next parameter is an array
24	9	Element Type	9	Array Elements are Strings
25	0	Element Count	2	Array contains two elements
26	2			
27	0	String Length	4	String is 4 bytes long
28	0			
29	0			
30	4			
31	102	Characters	f	String data is "foo1"
32	111		o	
33	111		o	
34	49		1	
35	0	String Length	4	String is 4 bytes long
36	0			
37	0			
38	4			
39	102	Characters	f	String data is "foo2"
40	111		o	
41	111		o	
42	50		2	
43	22	Parameter Type	22	Next parameter is a decimal
44	-1	Decimal data	-23325.23425	Decimal value "-23325.23425"
45	-1			
46	-1			
47	-1			
48	-1			
49	-1			
50	-1			
51	-1			
52	-1			
53	-83			
54	33			
55	-46			
56	-78			
57	57			
58	-39			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
59	-128			

Invocation response

An invocation response contains the results of the server's attempt to execute the stored procedure. The response includes optional fields and the first byte after the header is used to indicate which optional fields are present. The status string, application status string, and serializable exception are all optional fields. Bit 7 indicates the presence of a serializable exception, bit 6 indicates the presence of a status string, and bit 8 indicates the presence of an app status string. The serializable exception that can be included in some responses is currently not a part of the wire protocol. The exception length value should be used to skip exceptions if they are present. The status string is used to return any human readable information the server or stored procedure wants to return with the response. The app status code and app status string can be set by application code from within stored procedures and is returned with the response.

Name	Type	Length (bytes)	[Basic | Compound | Complex]
Message Header			
Client data	Binary	8	Basic
Fields present	Byte (bit field)	1	Basic
Status	Byte	1	Basic
Status string	String (optional field)	variable	Basic
Application Status	Byte	1	Basic
Application Status string	String (optional field)	variable	Basic
Cluster Round Trip Time	Integer	4	Basic
Serialized exception length	Integer (optional field)	4	Basic
Serialized exception	Serializable exception (optional field)	variable	Complex
Result count	Short	2	Basic
Result tables	Series of VoltTables	variable	Compound (containing Complex)

The following table shows a response to a previous invocation. For demonstrational purposes two tables are returned, but in a real failure case there would be no tables returned.

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
00	0	Message Length	109	The message is 109 bytes long
01	0			
02	0			
03	109			
04	0	Protocol version	0	This messages is a Volt Wire Protocol version 0 message
05	0	Client Data		Opaque client data
06	1			
07	2			
08	3			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
09	4			
10	5			
11	6			
12	7			
13	-32	Fields present	-32	The optional status string, app status string, and serializable exception fields are present
14	2	Status code	2	Status code is graceful failure (2)
15	0	Status String length	4	Status string length is 4
16	0			
17	0			
18	4			
19	102	Status String	f	The status string was "fail"
20	97		a	
21	105		i	
22	108		l	
23	99	Application status code	99	Application status code is 99
24	0	Application status string length	4	Application status string length is 4
25	0			
26	0			
27	4			
28	118	Application status string	v	The application status string was "volt"
29	111		o	
30	108		l	
31	116		t	
32	0	Cluster Round Trip Time	1	Milliseconds from invocation receipt to response on the cluster
33	0			
34	0			
35	1			
36	0	Serialized Exception length	5	Serialized exception is 5 bytes long
37	0			
38	0			
39	5			
40	1	Exception ordinal	1	Exception is a SQL exception
41	0	Exception body	Opaque	Opaque

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
42	0			
43	0			
44	0			
45	0	Result table count	2	Two result tables follow
46	2			
47	0	Total Table Length	32	Table size is 32 bytes
48	0			
49	0			
50	32			
51	0	Table Metadata Length	12	Header size is 12 bytes
52	0			
53	0			
54	12			
55	0	Status Code	0	Status code is 0
56	0	Column Count	1	Table contains 1 column
57	1			
58	6	Column 1 Type	VoltType.BIGINT	Column 1 is a BigInt
59	0	Col 1 Name Length	4	The name of column 1 has 4 chars (ASCII)
60	0			
61	0			
62	4			
63	84	Column 1 Name Value	T	Column 1 is named "Test"
64	101		e	
65	115		s	
66	116		t	
67	0	Row Count	1	Table has 1 row
68	0			
69	0			
70	1			
71	0	Row 1 Length	8	First row is 8 bytes long
72	0			
73	0			
74	8			
75	0	Row 1 Col 1 Value	5	Value in first row/first col 5
76	0			
77	0			
78	0			
79	0			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
80	0			
81	0			
82	5			
83	0	Total Table Length	32	Table size is 32 bytes
84	0			
85	0			
86	32			
87	0	Table Metadata Length	12	Header size is 12 bytes
88	0			
89	0			
90	12			
91	0	Status Code	0	Status code is 0
92	0	Column Count	1	Table contains 1 column
93	1			
94	6	Column 1 Type	VoltType.BIGINT	Column 1 is a BigInt
95	0	Col 1 Name Length	4	The name of column 1 has 4 chars (ASCII)
96	0			
97	0			
98	4			
99	84	Column 1 Name Value	T	Column 1 is named "Test"
100	101		e	
101	115		s	
102	116		t	
103	0	Row Count	1	Table has 1 row
104	0			
105	0			
106	1			
107	0	Row 1 Length	8	First row is 8 bytes long
108	0			
109	0			
110	8			
111	0	Row 1 Col 1 Value	5	Value in first row/first col 5
112	0			
113	0			
114	0			
115	0			
116	0			
117	0			

Byte Offset	Byte Value (dec)	Field Desc	Field Value	Meaning
118	5			